Presentation Manager Programming Guide The Basics

IBM

OS/2® WARP

Version 3

Presentation Manager Programming Guide The Basics

IBM

OS/2® WARP

Version 3

Note

Before using this information and the product it supports, be sure to read the general information under Appendix A, "Notices" on page A-1.

First Edition (October 1994)

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM authorized reseller or IBM marketing representative.

Contents

Presentation Manager Programming Guide - The Basics

Figures

Tables

About This Book

This book provides information and code examples to enable you to start writing source code, using the functions in the Presentation Manager application programming interface (API) of the OS/2 operating system.

Who Should Read This Book

The *Presentation Manager Programming Guide - The Basics* is intended for application programmers who want to create programs using the windowed, message-based Presentation Manager user interface. The basic concepts and features of this interface are discussed in this guide, including: windows, messages, controls, input/output, and system features such as the clipboard and timers. Details of messages, programming functions, and data structures used in application development are also included.

How This Book Is Organized

All chapters of this book, except the Introduction and Resource Files chapter, are divided into five main sections:

- *About* the topic

  Covers basic concepts, terminology, and general information about the topic.

- *Using* the topic

  Introduces many of the functions and structures related to the topic and provides examples in the form of code fragments.

- Functions

  Provides more details of the functions relevant to the topics in the chapter.

- Messages

  Provides details of the messages related to the topics discussed in the chapter.

- Data Structures

  Provides details of data structures related to the chapter topics.

- Summary

  Provides brief descriptions of the functions, messages, and data structures mentioned in the chapter.

To illustrate use of the functions and messages, this guide makes extensive use of code fragments. Sample applications also are available with the *Developer's Toolkit for OS/2 Version 3* (Toolkit). You may find it useful to execute the samples and examine the C files, resource files, makefiles, and other files provided by the Toolkit.

For information on how to compile and link your programs, refer to the compiler publications for the programming language you are using.

Prerequisite Publications

This guide is intended for application designers and programmers who are familiar with the following:

- Information contained in the *Control Program Programming Guide*
- Information contained in the *Presentation Manager Programming Reference* materials
- C Programming Language

Programming experience on a multitasking operating system would also be helpful.

Related Publications

The following diagram provides an overview of the OS/2 Technical Library.

Books can be ordered by calling toll free 1-800-342-6672 weekdays between 8:00 a.m. and 8:00 p.m. (EST). In Canada, call 1-800-465-4234.

OS/2 Warp, Version 3 Technical Library G25H-7116

- Control Program Programming Guide G25H-7101
- Control Program Programming Reference G25H-7102
- Graphics Programming Interface Programming Guide G25H-7106
- Graphics Programming Interface Programming Reference G25H-7107
- Information Presentation Facility Programming Guide G25H-7110
- Multimedia Application Programming Guide G25H-7112
- Multimedia Programming Reference G25H-7114
- Multimedia Subsystem Programming Guide G25H-7113
- Presentation Manager Programming Guide - Advanced Topics G25H-7104
- Presentation Manager Programming Guide - The Basics G25H-7103
- Presentation Manager Programming Reference G25H-7105
- REXX Reference S10G-6268
- REXX User's Guide S10G-6269
- Tools Reference G25H-7111
- Workplace Shell Programming Guide G25H-7108
- Workplace Shell Programming Reference G25H-7109

IBM Device Driver Publications for OS/2

- Display Device Driver Reference 71G1896
- Input/Output Device Driver Reference 71G1898
- MMPM/2 Device Driver Reference 71G3678
- Pen for OS/2 Device Driver Reference 71G1899
- Physical Device Driver Reference 10G6266
- Presentation Driver Reference 10G6267
- Printer Device Driver Reference 71G1895
- Storage Device Driver Reference 71G1897
- Virtual Device Driver Reference 10G6310

Presentation Manager Programming Guide - The Basics

Chapter 1. Introduction to Presentation Manager Programming

This chapter introduces the fundamental concepts and features of Presentation Manager* (PM*) and summarizes topics covered in this guide.

Presentation Manager Fundamentals

Presentation Manager (PM) provides a message-based, event-driven, graphical user interface (GUI) for the Operating System/2 * (OS/2*) environment. Some of the major features of PM are:

- The window environment
- The user interface
- Input management
- Application resource management
- Data exchange
- The Information Presentation Facility
- Presentation drivers

PM enables programmers to build applications that conform to Systems Application Architecture * (SAA *) guidelines. For more information on SAA requirements, see the *Systems Application Architecture: Common User Access Guide to User Interface Design* and the *Systems Application Architecture: Common User Access Advanced Interface Design Reference*.

The Window Environment

The PM user interface is based on *windows*, an area of the screen through which interaction is presented to the user. A large number of application programming interface (API) functions (which begin with the prefix Win) are available for controlling windows. These functions enable an application to create, size, move, and control windows and their contents. This guide describes common programming techniques for managing the window environment.

Defining Window Relationships

A window is an area of the screen where an application displays output and receives input from the user. A screen can have more than one window. The common analogy is that multiple windows on the screen are like many pieces of paper on a Desktop. In the analogy, the Desktop is the area that comprises the background of the screen. Windows, like papers, can be arranged to lie on top of one another and to overlap. If they overlap, the bottom papers can be either partially or completely hidden. Windows can be defined in the window hierarchy using the API.

Figure 1-1 on page 1-2 illustrates the window hierarchy as it appears on the screen.

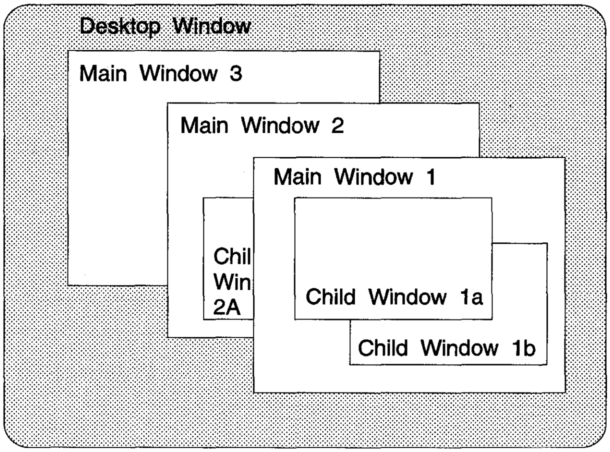


Figure 1-1. Windows on the Screen

The *Desktop window* is at the top of the hierarchy. Below the Desktop window are the top-level windows, called *main windows*. Main windows can overlap one another and, at times, a main window can be completely hidden. Operations on one main window do not affect those on other main windows. Figure 1-2 illustrates the hierarchical arrangement of windows created by the application.

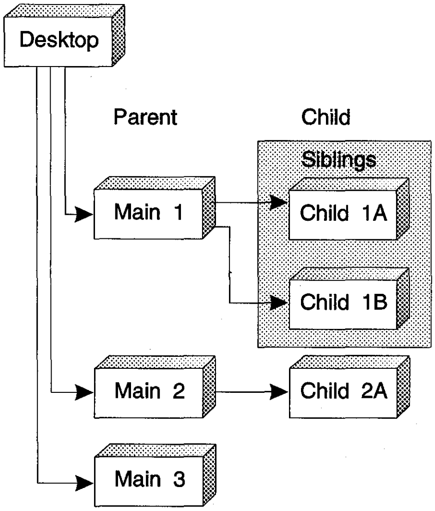


Figure 1-2. Window Hierarchy

Main windows can create subordinate windows in a parent-child order of descendancy. A child window is always *clipped* to its parent window, meaning that only the part of a child window that lies within the parent window is visible.

Windows that share the same parent are called *sibling windows*. Like main windows, sibling windows can overlap one another. Every sibling window has a *z-order* position that specifies where it lies in the stack of overlapping windows.

The application can define another relationship in addition to the hierarchical one. When a window is created, an *owner window* can be defined. The two windows must be created by the same thread. The owner relationship varies at different levels of the hierarchy. A child window can send messages to its owner window. If one main window owns other main windows and the owner window is hidden, minimized, or closed, all the owned main windows also are hidden, minimized, or closed.

A window can be visible, hidden, or partly hidden on the screen. When a window is hidden or partly hidden, its size, position, and hierarchical and owner relationships remain the same. However, when the window becomes visible again, any area of the window that was previously hidden is redrawn. A window can also be disabled, meaning that it is still visible but unable to respond to mouse input.

Creating and Classifying Windows

A window and its associated *window procedure* are considered to be a *program object*. A window procedure "represents" a window in the sense that the window procedure controls all aspects of the window, such as what it looks like, how it responds to changes, and how it processes input.

A *window class* is a set of windows that has the same window procedure to implement them. Many windows can belong to a window class. The windows can differ from one another only in the data they process. If multiple applications have need for the same type of window, implementing common window classes is an efficient way of using system resources.

The OS/2 operating system provides many *preregistered* window classes. The windows specified in these classes are designed specifically to meet the needs for a graphics-based standard user interface. If a preregistered window class is not provided, the application must register the class at the process level. Several API functions are available for applications to reserve a small area of memory (the *window words* area) for windows in classes registered by the application. If a window is expected to handle large amounts of data, the data should be held in memory and referred from the window words area.

A window class can be defined as *private* or *public*. Windows created in either class can be used by any process in the system. "Public" and "private" refer only to the window class at the time the window is created.

Only the process with which a private class is registered can create a window for that class. The class name must be unique to the process. However, other processes can register private classes with the same class name.

Any process can create a window in a public class. Window procedures for windows in public classes must be available to all processes. Thus, such classes should be defined in dynamic link libraries (DLLs). Public-class names must be unique for each process.

All windows have certain attributes. Each window is identified by a *window handle*. Each window represents a *rectangle* describing the size and position of the window on the screen. The size of a window is defined in picture elements (pels) relative to the origin of the parent window. The origin of a window, the lower-left corner, is the 0,0 coordinate in a set of x and y axes. The x and y coordinates define the top, bottom, and sides of the window. The coordinates range from –32768 to +32767 pels in each direction, so the maximum size that can be specified in any direction is 65,535 pels. The application also can position a window by defining its relative distance from the point of origin (0,0) of its parent window. If the application positions a child window outside its parent window, which is permissible, only the part of the child window within the parent window will be visible on the screen.

Using a set of API functions, the application can modify the behavior of a window in a window class, or create a new class from an existing one. This process, called *sub-classing*, enables the application to modify the behavior of a single window without rewriting its complete window procedure.

Providing the User Interface

The Common User Access* (CUA*) is a set of guidelines for designing and writing the application's user interface. The guidelines cover standard menu-bar items, interaction techniques, and window types. Use the CUA guidelines when deciding how to design the user interface for your application's user interface.

Many PM services, if allowed to default, help to enable a consistent user interface among applications designed and written according to the CUA guidelines. Consistency also is enabled by the selection of appropriate options. Ensuring consistency is the responsibility of the application developer.

Standard and Control Windows

Information is displayed on the screen through the use of windows. PM supports a *standard window*, whose elements generally conform to CUA guidelines.

The standard window, which the application controls using the API functions, can have all or some of the following elements:

- Title-bar icon
- Window borders
- Window sizing buttons
- Menu bar
- Scroll bars
- Window title
- Information area

Figure 1-3 on page 1-5 illustrates a standard window and its elements.

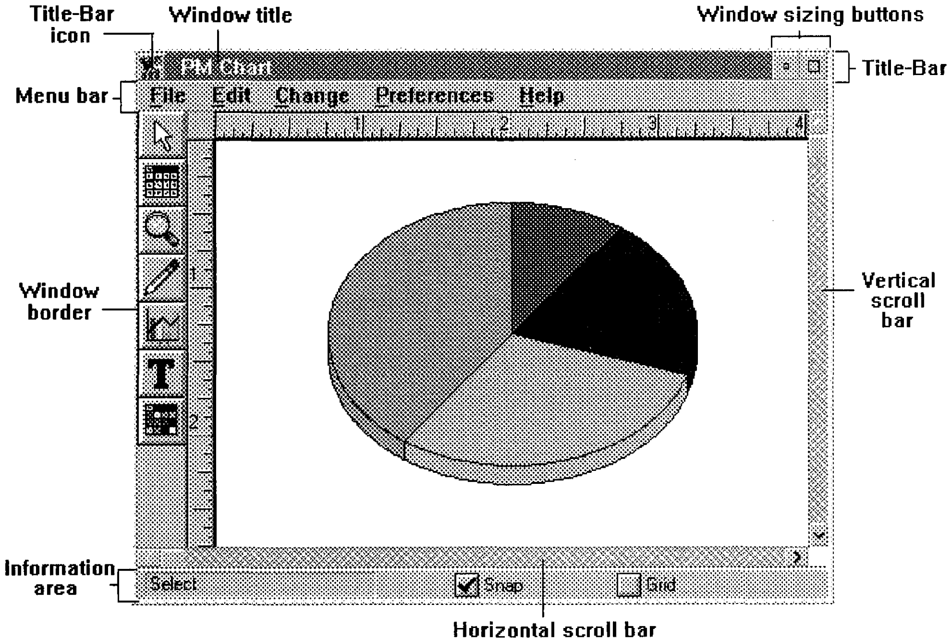


Figure 1-3. Standard Window

The title-bar icon, window border, and window sizing buttons enable a user to change the size and position of a window. The menu bar and scroll bars enable a user to work with the window's contents. The window title indicates the name of the object seen in the window, and it also indicates which kind of *view* is displayed. A *view* is a way of looking at an object's information. Different views display information in different forms, which mimics the way information is presented in the real world. The information area displays brief messages to a user about the object or choice that the cursor is on. Information about the normal completion of a process can also appear in the information area. For example, if a user copies several objects from one container to another, the information area in a container's window might display a brief message to tell the user when the copying has been completed.

The standard window is created using a standard *frame window*. The elements of the standard window, such as the title bar and the menu bar, are child windows of the standard frame window. The child windows are called *control windows*. The system maintains a set of preregistered control windows that any application can use to perform I/O.

From the application's perspective, control windows are no different from other windows in the system, and the application can manage them using window-management functions. Each control window has its own window identifier and a specific set of messages. The application can query the system to determine the control window's parent. Control windows can be used as a part of a *dialog window*. A dialog window can be created using a *dialog template*, which defines the position, appearance, and identifier of the dialog window and each of its child windows. A template can be loaded as a resource or created dynamically in memory. It can be used to create dialog windows of all window classes. The window classes can contain control windows of all window classes. Also, the application can create its own dialog controls by creating and preregistering its own control-window class.

A dialog window is controlled by a window procedure called a *dialog procedure*. The dialog procedure is responsible for responding to all messages sent to the dialog window, either by sending them to the control windows or returning them to the default dialog procedure. A set of API functions enables the application to create, load, process, and cancel dialog windows. The dialog procedure can obtain the handle of its child windows, send messages to them, and process messages and text strings itself.

The standard frame window and the control windows are implemented with standard preregistered window classes. The standard frame window manages the control windows and the client window as the user interacts with them. The frame window also is responsible for routing messages to the appropriate control and client windows.

Primary and Secondary Windows

CUA guidelines define two types of windows: *primary* and *secondary*. In a PM program, a primary window is a standard window, while a secondary window is a control window or the child of the main window.

A primary window is the main interface point between an object and the user. It appears when a user opens an object, and is used to present a *view* of an object or group of objects when the information displayed about the object or group of objects is not dependent on other objects.

Object information is presented in the area of the window below the menu bar. A user can control the size and position of primary windows on the screen.

A secondary window looks very much like a primary window. For example, both have window borders and title bars. The important distinction between a primary window and a secondary window is based on how they are used. A secondary window is always associated with a primary window and contains information that is dependent on an object in the primary window. A secondary window is used, for example, to allow a user to further clarify action requests. A secondary window is always removed when the primary window is closed or minimized and redisplayed when the primary window is opened or restored.

Dialog Box

A *dialog box* extends a dialog between a user and a primary or secondary window. It usually appears when the user selects a choice from the menu bar, thereby generating a pull-down menu. Selecting one of the choices in the pull-down menu generates the dialog box. The dialog box can contain buttons, entry fields, icons and text, list boxes, and title bars. A dialog box and its supporting window enable the application to gather input from the user. A temporary dialog window usually is created for special-purpose input and is then canceled.

There are two types of dialog boxes: *modal* and *modeless*. A modal dialog box retains control until the application issues a call to cancel it. Users cannot activate other windows belonging to the application until they finish interacting with the modal dialog box. A modeless dialog box enables windows in other applications to be activated after it has been created.

Handling Mouse and Keyboard Input

The session manager in the operating system manages applications running in the PM environment, including their input and output operations. However, PM handles PM applications, including their input and output operations. PM handles all input as *messages*, which are packets of data.

PM supports user input from the keyboard and *mouse pointer*. The mouse pointer is the symbol associated with the mouse pointing device. Mouse input is provided by pressing a button and usually is directed at the window under the mouse pointer. The precise position on the screen that is activated is called the *hot spot*. The mouse pointer also can be moved across the screen, and the operating system provides support for that activity. The application can direct all mouse input to a single window called a *mouse capture window*. A mouse capture window enables the application to track all input from the mouse pointer no matter where the mouse pointer is moved on the screen.

Keyboard input is sent when any key on the keyboard is pressed. All keyboard input is directed to one window at a time. The window receiving keyboard input is called the *active window*. A main window, or one of its child windows, is responsible for keeping the window receiving the input visible on the screen.

The *cursor* is a symbol displayed within a window that indicates where characters entered from the keyboard will be placed. The cursor can be moved to any location within a window. Its size and position are defined in coordinates relative to the window in which the cursor is located. The application can create, display, move, and cancel the system cursor.

Processing Messages

The Common Programming Interface (CPI) defines an application structure that uses a system of queues and application window procedures to process messages. The PM message system conforms to the CPI and is fundamental to the smooth operation of the PM environment. A complete set of CPI reference manuals is provided in the SAA Library.

Figure 1-4 illustrates the PM message-processing system.

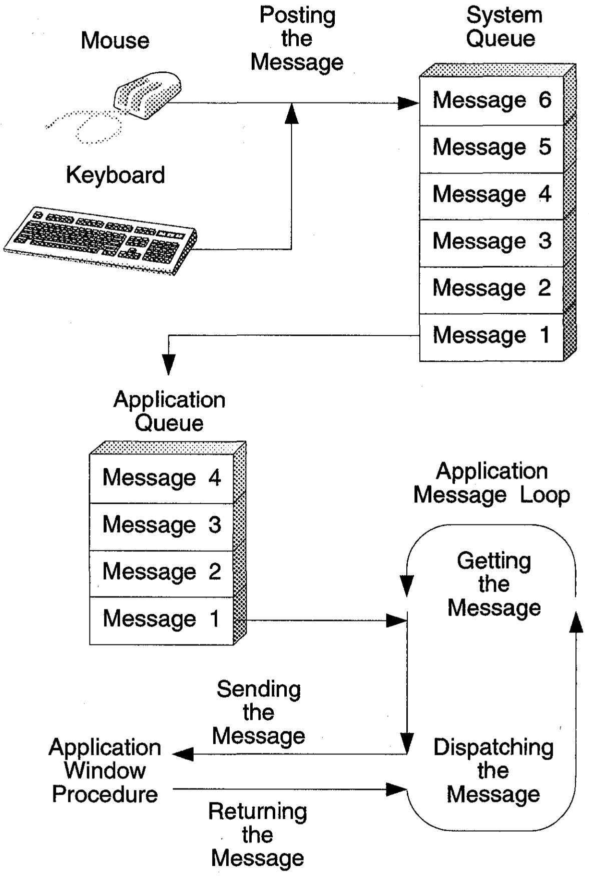


Figure 1-4. Message-Processing System

In the PM environment, each input from the mouse or keyboard is delivered to an application as a message. A message cannot be processed before previous input, because the specific application and window for which input is intended are not known until all preceding input has been processed.

All input is first placed in a single queue called the *system queue*. The system queue, which is shared by all applications in the system, receives messages generated by the user from the mouse and keyboard. The system queue can hold the input from approximately 60 key presses and mouse clicks. The system queue can receive input from many sources, including the system itself, the timer, and other applications. However, only user input is processed synchronously.

The system queue temporarily stores user input so that nothing is lost if the user enters data faster than the application can process it. Generally, input is processed in the order in which it appears in the queue, but the application can change the order by *filtering* the input. Filtering is performed with functions, but should be performed with discretion, because the processing of one input often changes the context for the next.

Each thread that receives input has an *application queue* allocated by an API function. This queue does not receive user input directly, but can receive other messages directly, such as messages from the system or timer. Messages are removed from the application queue when the thread for which it is destined "gets" it. The messages are prioritized if more than one is waiting. If there are none, the thread is suspended until a message arrives.

The most efficient use of the system will be achieved if you structure your application so that one thread remains responsive to user input while others continue processing work. To be considered responsive to the user, the system must complete processing input within 0.5 seconds. That is, the thread handling input should check for the next message in the queue within that time.

A window procedure can control more than one window. The procedure receives messages in the form of four input parameters. The first parameter specifies the handle of the window for which the message is intended. The second parameter indicates the type of message. The last two parameters contain message parameters. Their interpretation depends on the particular message.

The window procedure processes the message and then sends a return value to the sending code. A window procedure must respond to all messages sent to it, even if the response is to send the message back to the system's default window procedure.

There are many types of messages, each with a unique identifier, and applications can define their own message types using a range of identifier values.

Handling Application Resources

An *application resource* is held in a *resource file*, a file with information that helps to define a window. The resource file defines the names and attributes of the application resources that must be added to the application's executable file. The resources are as follows:

Accelerator table Used to define which key strokes are treated as *accelerators* and the commands into which they are translated. An accelerator is a single key stroke that invokes an application-defined function.

Bit map A representation in memory of the data displayed on an all-points-addressable (APA) device, usually the screen.

Dialog and window templates
The definitions of a dialog box or window containing details of its position, appearance, window identifier, and the identifiers of its child windows.

Dialog include A definition of a dialog box in a header file.

Fonts A typeface definition for character sets, marker sets, and pattern sets.

Icon A graphical representation of an object, consisting of an image, image background, and a label.

Menu A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are indicated by reduced contrast.

Pointer The symbol displayed on the screen that can be moved by a pointing device. The pointer is defined in a bit map.

String table A null-terminated ASCII string. A string table is loaded when it is needed by the executable file.

For guidance in building the resource file using the *Resource Compiler* (RC) utility in the *Developer's Toolkit for OS/2 Version 3*, see the online Tools Reference. The RC processes the resource text file to produce a binary file, and then attaches it to the application's executable file so that an application can access its resources.

Resource Editors

The *Dialog Box Editor* in the *Developer's Toolkit for OS/2 Version 3* enables you to design dialog boxes interactively on the screen and save the definitions in a resource file. The definition of the dialog box is included with other resource definitions in the application's resource file.

The *Font Editor* in the Toolkit enables you to edit font files interactively on the screen, save the definitions in a font file, and include the font file names in the application's resource file. The font file consists of a header file and a collection of character bit maps representing the individual letters, digits, and punctuation characters that display text on a screen.

The *Icon Editor* in the Toolkit enables you to create customized icons, pointers, and bit maps interactively on the screen and save the definitions in a resource file. You can work on a large-scale version of the icon or pointer while displaying a replica of the actual size.

Exchanging Data Among Applications

User-Generated Data Exchange

Data exchange requested by a user is held in an object called a *clipboard*. The user can transfer data from one application to another using the COPY, CUT, and PASTE commands. The first step is to copy or cut (delete) selected data from the source application; the data is now in the clipboard. Next, paste (insert) the clipboard data into the target application. The same process can be used to move data from one window to another within a single application. The CUT, COPY, and PASTE commands must be supported by an application as defined in CUA guidelines. They are implemented using a set of PM API functions.

The clipboard is the object that temporarily holds data. Generally, data is placed in the clipboard when a request to paste it is received. Once data has been sent to the clipboard, it should not be changed. Only one item of data at a time can be in the clipboard, but the data can be in a variety of formats, such as text, metafile, or bit map. The application can either define the formats or use one of the preregistered standard formats. The application also can register formats, as it can window classes, so they can be used by all applications in the system.

The application should support as many formats as possible to satisfy requests from target applications. For example, a spreadsheet application should support a spreadsheet format and as many common text formats as possible. Generating data in all formats supported by an application can consume a lot of the operating system's resources. It would not make sense, for example, for a word-processing application to support a spreadsheet format because that format is beyond the scope of the operation of a word processor.

A clipboard can be owned by a thread. If a thread opens the clipboard, it has exclusive access to the clipboard until the thread closes the clipboard.

The clipboard is owned by the last window that requested ownership. Only the owning application can change the owner of the clipboard. If an owning window is canceled, data can remain in the clipboard. Before being canceled, the owning window must generate its data to satisfy subsequent paste requests.

Topics Covered in This Guide

The following section gives a brief overview of the topics included in each chapter.

Windows

The PM user interface is based on windows. This chapter describes window types and relationships and defines programming techniques for managing the window environment. Creating, sizing, moving, and controlling windows and their contents is discussed, with programming examples provided.

Messages and Message Queues

The OS/2 operating system uses messages and message queues to communicate with applications and the windows belonging to those applications. This chapter describes types of messages, message queues, and message handling in Presentation Manager applications.

Window Classes

A window class determines which styles and which window procedure are given to a window when it is created. This chapter explains types of window classes, class registration, and how an application creates and uses window classes.

Window Procedures

Windows have an associated window procedure—a function that processes all messages sent or posted to a window. Every aspect of a window's appearance and behavior depends on the window procedure's response to the messages. This chapter discusses types of window procedures and how they are customized and used in Presentation Manager applications.

Frame Windows

The standard window is created using a frame window. A frame window is the basic window used by most Presentation Manager applications to enable the user to perform manipulation functions. This chapter describes creation and use of frame windows in Presentation Manager applications.

Painting and Drawing

An application typically maintains an internal representation of the data that it is manipulating. The information displayed in a screen, window, or printed copy is a visual representation of some portion of that data. This chapter introduces the concepts and strategies necessary to make your PM application function smoothly and cooperatively in the OS/2 display environment. This chapter describes presentation spaces, device contexts, window styles, and window regions, explaining how a Presentation Manager application uses them for painting and drawing in windows.

Drawing in Windows

This chapter describes the Presentation Manager functions for drawing in windows. These drawing functions are somewhat easier to use than the special purpose graphics functions (the Gpi* functions), but they offer a less complete graphics-drawing interface.

Mouse and Keyboard Input

PM supports user input from the keyboard and pointing devices. This chapter discusses keyboard focus and messages, mouse messages, window activation, and handling of input messages in Presentation Manager applications.

Mouse Pointers and Icons

A mouse pointer is a special bit map the operating system uses to show a user the current location of the mouse on the screen. When the user moves the mouse, the mouse pointer moves on the screen. This chapter describes how to create and use mouse pointers and icons in Presentation Manager applications.

Cursors

The cursor is a symbol displayed within a window that indicates where characters entered from the keyboard will be placed. This chapter discusses creating, destroying, positioning, and sizing cursors.

Resource Files

These files are used to specify resource information and allow modification of resources without recompiling the entire application. Examples of resources that can be specified are menus, fonts, strings, and icons. This chapter describes resource file statements and directives, and how files are created and compiled.

Menus

A menu is a window that contains a list of items— text strings, bit maps, or images drawn by the application—that enables the user, by mouse or keyboard, to choose from these predetermined choices. This chapter describes types of menus, menu items, menu access, and how to use menus in Presentation Manager applications.

Keyboard Accelerators

A keyboard accelerator (shortcut key) is a keystroke that generates a command message for an application. This chapter discusses accelerator tables, resources, styles, data structures, and usage in applications.

Dialog Windows

A dialog window is a temporary window that contains one or more control windows and, typically, is used to display messages to and gather input from the user. This chapter describes types of dialog windows and message boxes, and how they are created and used in Presentation Manager applications.

Control Windows

A control window is a window that an application uses in conjunction with another window to carry out simple input and output tasks. This chapter discusses use of control windows in various types of windows, and how to create customized control windows.

Title Bars

The title bar in a standard frame window performs the following functions:

- Displays the title of the window across the top of the frame window.
- Changes its highlighted appearance to show whether the frame window is active.
- Responds to the actions of the user.
- Flashes to get the attention of the user.

This chapter describes default title-bar behavior and use of title bars in frame windows.

Scroll Bars

Scroll bars are control windows that convert mouse and keyboard input into integers; they are used by an application to scroll the contents of a client window. This chapter describes scroll-bar styles, ranges, positions, and notification messages and how to create and use scroll bars in Presentation Manager applications.

Button Controls

A button is a type of control window used to initiate an operation or to set the attributes of an operation. This chapter discusses button-control types, styles, default behavior, notification messages, and states. Creation of button controls and their use in Presentation Manager applications is also described.

Entry Field Controls

An entry field is a control window that enables a user to view and edit a single line of text. This chapter discusses entry-field styles, notification codes, and behavior, as well as creating and using entry field controls in Presentation Manager applications.

List Boxes

A list box is a control window that displays several text items at a time, one or more of which can be selected by the user. This chapter describes how list-box controls are created and used in Presentation Manager applications.

Clipboards

The clipboard is a small amount of system-managed random-access memory (RAM) used for user-driven data exchange. This chapter describes how to use the clipboard in Presentation Manager applications.

Window Timers

A window timer enables an application to post timer messages at specified intervals. This chapter discusses types of timers and their applications.

Initialization Files

Initialization files enable an application to store and retrieve information that the application uses when it starts up. This chapter describes creating, opening, closing, and using application and system initialization files.

Chapter 2. Windows

To most users, a *window* is a rectangular area of the display screen where an application receives input from the user and displays output. This chapter describes the parts of the operating system that enable a Presentation Manager (PM) application to create and use windows; manage relationships between windows; and size, move, and display windows. An overview of the following topics is presented:

- Window types, classes, and styles
- Window-creation techniques
- Window messages and message queues
- Methods of window input and output
- Window resources and procedures
- Window identification and modification.

Subsequent chapters present more in-depth descriptions of windows, their advantages and uses, along with example code fragments.

About Windows

A PM application can interact with the user and perform tasks only by way of windows. Each window shares the screen with other windows, including those from other applications. The user employs the mouse and keyboard to interact with windows and their owner applications.

Desktop Window and Desktop-Object Window

The OS/2 operating system automatically creates the *desktop window* (known as the *workplace* in user terminology) when it starts a PM session.

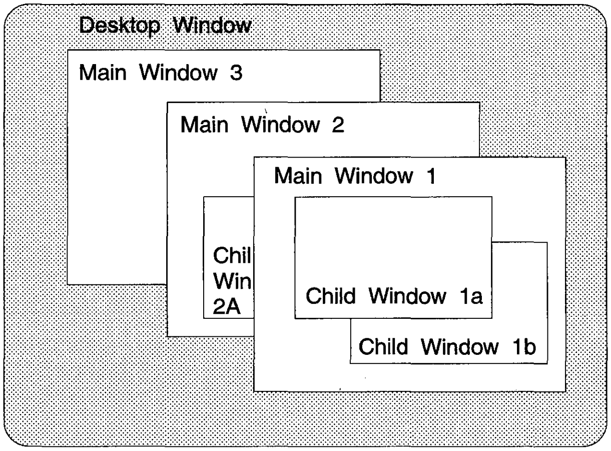


Figure 2-1. Desktop Window Containing Windows of Several Applications

The desktop window paints the background color of the screen and serves as the "progenitor" of all the windows displayed by all PM applications (but not of object windows, which do not require screen display). To make the desktop the parent in the WinCreateStdWindow function, you specify HWND_DESKTOP.

The windows immediately below the desktop are called *main* or *top-level* windows; these are called *primary windows* in user terminology. Every PM application creates at least one window to serve as the main window for that application. Most applications also create many other windows, directly or indirectly, to perform tasks related to the main window.

Each window helps display output and receive input from the user. Figure 2-1 on page 2-1 shows the desktop window containing windows of several applications. Notice that the main windows can overlap one another. (At times, it is possible for a main window to be completely hidden.) Operations in one main window normally do not affect the other main windows.

The *desktop-object window* is like a desktop window that is never displayed; it serves as the base window to coordinate the activity of an application's object windows. The desktop-object window cannot display windows nor process keyboard and mouse input. The primary purpose of the desktop-object window is to enable you to create windows that need not respond to messages at the same rate as the user interface.

Window Relationships

Window relationships define how windows interact with each other—on the screen and through messages. There are parent-child window relationships and window-owner relationships.

The *parent-child relationship* determines where and how windows appear when drawn on the screen. It also determines what happens to a window when a related window is destroyed or hidden. The parent-child rules apply to all windows at all times and cannot be modified.

Ownership determines how windows communicate using messages. Cooperating windows define and carry out their rules of ownership. Although some windows (such as windows of the preregistered public window class, WC_FRAME) have very complex rules of ownership, the application usually defines the ownership rules.

Figure 2-2 represents the logical relationship of the windows in two applications.

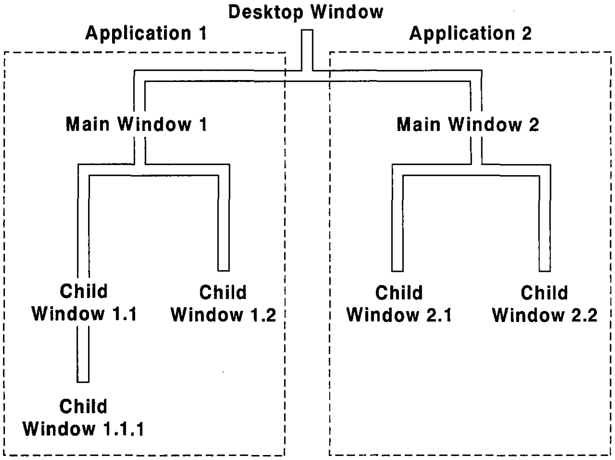


Figure 2-2. Typical Window Relationships

Parent-Child Relationship

Most windows have a *parent window.* (The exceptions are the desktop and desktop-object windows, which the system creates at system startup.) An application specifies the parent when it creates a window; then, the system uses the parent to determine where and how to draw any new windows, as well as when to *destroy* the windows (free all associated resources and remove the windows from the screen).

A *child window* is drawn relative to its parent. The coordinates given to specify the position of a window's lower-left corner are relative to the lower-left corner of its parent. For example, a main window (child of the desktop) is drawn relative to the lower-left corner of the screen (the desktop window's lower-left corner).

All main windows are *siblings* because they share a common parent, the desktop window. Because sibling windows can overlap, an application or a user arranges the windows, one behind another (like a stack of papers on a desk), in the desired viewing order (called *z-order*) as illustrated in Figure 2-1 on page 2-1. Z-order uses the desktop as a reference point for a "three-dimensional" ranking of the overlapping windows: the topmost window has the highest ranking, while the window at the bottom of the stack has the lowest ranking. The parent of the sibling windows is always at the bottom of the z-order.

Figure 2-3 illustrates the hierarchy of such an arrangement.

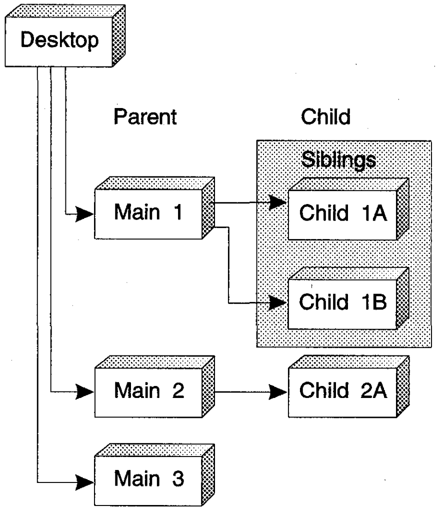


Figure 2-3. Window Hierarchy

Although PM *supports* z-order, it does not *enforce* the expected appearance unless you specify the CS_CLIPCHILDREN or CS_CLIPSIBLINGS styles. No part of a child window ever appears outside the borders of its parent. If an application creates a window that is larger than its parent, or positions a window so that some or all of it extends beyond the borders of the parent, the extended portion of the child window is not drawn.

An application can use the WS_CLIPCHILDREN or WS_CLIPSIBLINGS styles to remove from a window's *clipping area* (the area in which the window can paint) the area occupied by its child or sibling windows. For example, an application can use these styles to prevent a window from painting over a child or sibling window containing a complex graphic that would be time-consuming to redraw.

When a window is minimized, hidden, or destroyed, all of its children are hidden, minimized, or destroyed as well. The order of destruction is always such that every window is destroyed before its parent. The window-destruction sequence starts at the bottom of descendancy so that all related windows can be cleaned up; the last one to go is the window you asked to be destroyed. The final PM task in a window-destruction sequence is to send a WM_DESTROY message to that window, so it has one last chance to release any resources it has allocated and may still be holding.

Every window has only one parent, but can have any number of children. Referring back to Figure 2-3, any window in this tree is said to be a *descendant* of any window appearing above it in the branch, and an *ancestor* of any window appearing below it. There are two special cases, of course: the window immediately above is called the window's *parent*, and any window immediately below it is called its *child*. An application can change a window's parent window at any time by using the WinSetParent function. Changing the parent window also changes where and how the child window is drawn. The system displays the child within the borders of the new parent and draws the window according to the styles specified for the new parent.

Ownership

Any window can have an *owner window*. Typically, an application uses ownership to establish a connection between windows so that they can perform useful tasks together. For example, the title bar in an application's main window is owned by the frame window; but, together, the user can move the entire main window by clicking the mouse in the title bar and dragging. An application can set the owner window when it creates the window or at a later time.

Ownership establishes a relationship between windows that is independent of the parent-child relationship. While there are few predefined rules for owner- and owned-window interaction, a window *always* notifies its owner of anything considered a *significant event.*

The preregistered public window classes provided by the OS/2 operating system recognize ownership. Control windows of classes such as WC_TITLEBAR and WC_SCROLLBAR, notify their owners of events; frame windows, of class WC_FRAME, receive and process notification messages from the control windows they own. For example, a title-bar control sends a notification message to its owner when it receives a mouse click. If the owner is a frame window, it receives the notification message and prepares to move itself and its children.

Owner and owned windows must be created by the same thread; that is, they must belong to the same message queue. Because ownership is independent of the parent-child relationship, the owner and owned windows do not have to be descendants of the same parent window. However, this can affect how windows are destroyed. Destroying an owner window does not necessarily destroy an owned window. Except for frame windows, an application that needs to destroy an owned window that is not a descendant of the owner window must do so explicitly.

Frame windows sometimes own windows that are not descendants but, instead, are siblings. A frame window has the following special ownership properties:

- When the frame window is destroyed, it destroys all of the windows it owns, even if they are not descendants.
- When a frame window moves, the windows it owns move also. Owned windows that are not descendants maintain their positions, relative to the upper-left (not the usual lower-left) corner of the owner window. An owned window with the style FS_NOMOVEWITHOWNER does not move.
- When the frame window changes its position in the z-order, it changes the z-order of all the windows it owns.
- When the frame window is minimized or hidden, it hides all the windows it owns. Owned windows hidden this way are restored when the frame window is restored.

If an application needs this type of special processing for its own window classes, it must provide that support in the window procedures for those classes.

Object Windows

Any descendant of the desktop-object window is called an *object window*. Typically, an application uses an object window to provide services for another window. For example, an application can use an object window to manage a shared database. In this way, a window can obtain information from the shared database by sending a message to and receiving a reply from the object window.

Only two system-defined messages are available to an object window—WM_CREATE and WM_DESTROY—but the object window enables the user to implement a set of user-defined messages. The window procedure for an object window does not have to process paint messages or user input. The object window processes only messages that affect the data belonging to the object.

HWND_OBJECT is the only identifier needed to create an object window. It is very unwise to create descendants of HWND_OBJECT in the same thread that creates descendants of HWND_DESKTOP: this causes the system to hang up or, at the very least, behave slowly. Object windows, sometimes referred to as *orphan* windows, require no owner.

The rules for parent-child and ownership relationships also apply to object windows. In particular, changing the parent window of an object window to the desktop window, or to a descendant of the desktop window, causes the system to display the object window if the object window has the WS_VISIBLE style.

Application Windows

An application can use several types of *secondary* windows: frame windows, client windows, control windows, dialog windows, message boxes, and menus. Typically, an application's main window consists of several of these windows acting as one. Figure 2-4 on page 2-7 shows an example of a main window and its secondary windows.

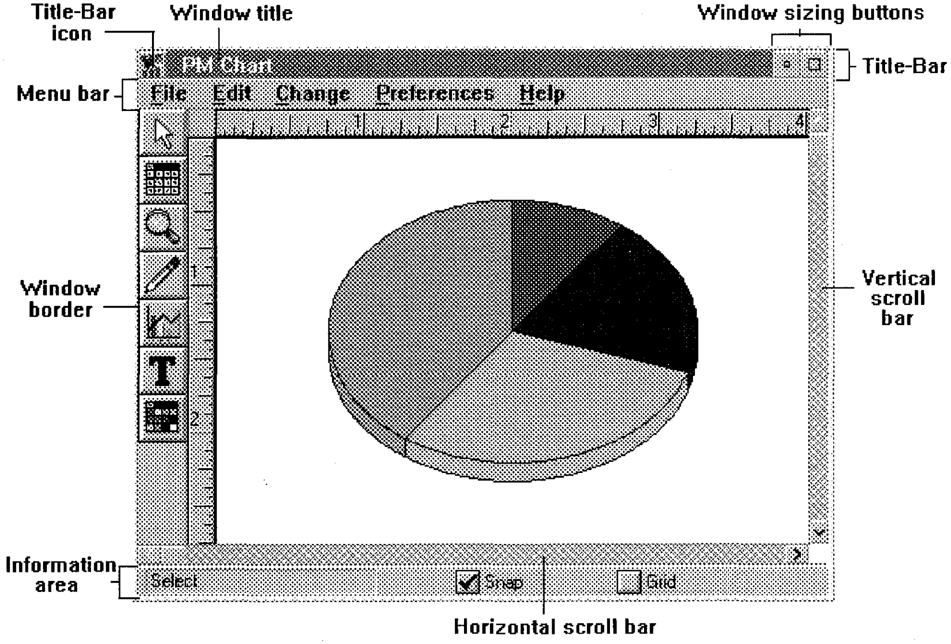


Figure 2-4. Main Window with Secondary Windows

A *frame window* is a window that an application uses as the base when constructing a main window or other composite window, such as a dialog window or message box. (A *composite window* is a collection of windows that interact with one another and are kept together as a unit.) A frame window provides basic features, such as borders and a menu bar. Frame windows have a set of resources associated with them. These include icons, menus, and accelerators (*shortcut keys* to the user), which, typically, are defined in an application's resource file.

A *dialog window* is a frame window that contains one or more control windows. Dialog windows are used almost exclusively for prompting the user for input. An application usually creates a dialog window when it needs additional information to complete a command. The application destroys the dialog window after the user has provided the requested information.

A *message box* is a frame window that an application uses to display a note, caution, or warning to the user. For instance, an application can use a message box to inform the user of a problem that the application encountered while performing a task.

A *client window* is the window in which the application displays the current document or data. For example, a desktop-publishing application displays the current page of a document in a client window. Most applications create at least one client window. The application must provide a function, called a *window procedure*, to process input to the client window and to display output.

A *control window* is a window used in conjunction with another window to perform useful tasks, such as displaying a menu or scrolling information in a client window. The operating system provides several predefined control-window classes that an application can use to create control windows. Control windows include buttons, entry fields, list boxes, combination boxes, menus, scroll bars, static text, and title bars.

A *menu* is a control window that presents a list of commands and other menus to the user. Using a mouse or the keyboard, the user can select a task; the application then performs the selected task.

Window Input and Output

The user directs input data to windows from a mouse and the keyboard. Keyboard input goes to the window with *input focus*, and, normally, mouse input goes to the window under the mouse pointer.

Windows also are places to display output data. PM uses windows to display text and graphics on the screen and to process input from the mouse and keyboard. Windows provide the same input and output capabilities as a virtual graphics terminal without having direct control of the hardware.

An application is responsible for painting the data for the window classes it registers and creates. This data can be graphics text or pictures or fixed-size alphanumeric text. Normally it is not necessary for the application to paint the system-provided window classes; the OS/2 window procedures for those window classes do the painting.

Active Window and Focus Window

All frame-window ancestors of the input focus window are said to be *active*, meaning that the user interacts with them. The active window usually is the topmost main window, which is positioned above all other top-level windows on the screen. The active window is indicated by some form of highlighting. For example, a highlighted title bar shows that a standard frame window is active; an active dialog window has a highlighted border. These types of highlighting ensure that the user can see the window that is accepting input.

A main window (or one of its child windows) is activated by using a mouse or the keyboard. When a window is activated, it receives a WM_ACTIVATE message with its first parameter set to TRUE. When it is deactivated, it receives a WM_ACTIVATE message with its first parameter set to FALSE. Figure 2-5 on page 2-9 illustrates user interaction with a window.

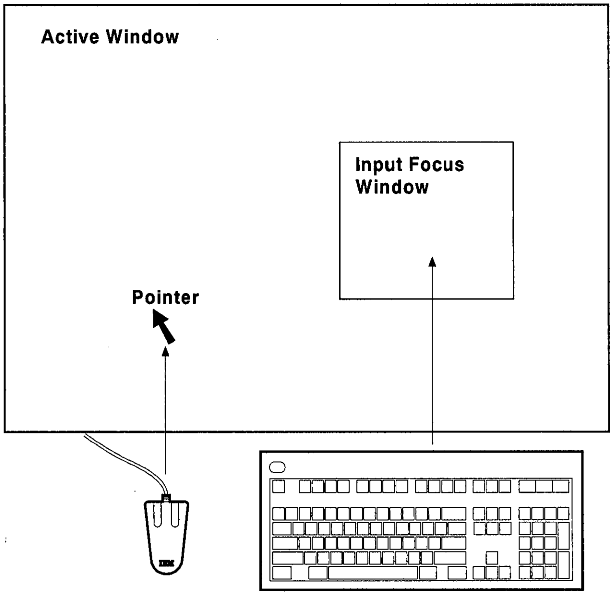


Figure 2-5. User Input to a Window

The *focus window* can be the active window or one of its descendant windows. The user can change the *input focus* the same way active windows are changed—by mouse or keyboard. However, the application has more control over the input focus. For example, in a window containing several text entry fields, the tab keys can move the input focus from one input field to another. A WM_SETFOCUS message is sent to the window procedure when a window is gaining or losing the input focus. The WinQueryFocus function tells the user which window has the input focus.

Messages

Messages are a fundamental part of the operating system. PM applications use messages to communicate with the operating system and one another. The system uses messages to communicate with applications to ensure concurrent running and sharing of devices. Typically, a message notifies the receiving application that an *event* has occurred. The operating system identifies the appropriate application window to receive a message by the window handle included in the message. Sources of events that cause messages to be issued to applications are the user, the operating system, the application, or another application.

The User: Mouse or keyboard input to an application window causes the operating system to direct messages to that window.

The Operating System: Managing the application windows on the screen, the operating system issues messages to the windows, usually as an indirect result of user interaction. These messages enable the system to work in a uniform and well-ordered manner. For example, where several application windows overlap, and the user terminates an application so that its window disappears, the operating system issues messages to the underlying application windows so that they can repaint themselves.

The Application: An event can occur in the application to which another part of that application should respond; for example, when the contents of its window no longer accurately reflect the status of the application. The application can define its own messages outside the range of system-defined messages to communicate such events.

Another Application: Communication with other applications through the operating system ensures cooperative use of the system; it even can be used to exchange data. For example, an arithmetic application can supply the results of a lengthy calculation to a business graphics application.

Enabled and Disabled Windows

An application uses the WinEnableWindow function to enable or disable window input. By default, a window is enabled when it is created. However, an application can disable a newly created window.

An application usually disables a window to prevent the user from using the window. For example, an application might disable a push button in a dialog window. Enabling a window restores normal input; an application can enable a disabled window at any time.

When an application uses the WinEnableWindow function to disable an existing window, that window also loses keyboard focus. WinEnableWindow sets the keyboard focus to NULL, which means that no window has the focus. If a child window or other descendant window has the keyboard focus, it loses the focus when the parent window is disabled.

An application can determine whether a window is enabled by calling WinIsWindowEnabled.

System-Modal Window

An application can designate a *system-modal window*: a window that receives all keyboard and mouse input, effectively disabling all other windows. The user must respond to the system-modal window before continuing work in other windows. An application sets and clears the system-modal window by using the WinSetSysModalWindow function.

Because system-modal windows have absolute control of input, you must be careful when using them in your applications. Ideally, an application uses a system-modal window only when there is danger of losing data if the user does not respond to a problem immediately.

Although an application can destroy a system-modal window, the new active window then becomes a system-modal window. An application can make another window active while the first system-modal window exists. But again, the new active window will become the system-modal window. In general, once a system-modal window is set, it continues to exist in the PM session until the application explicitly clears it.

Window Creation

Before any thread in an application can create windows, it must:

1. Call WinInitialize to create an anchor block
2. Call WinCreateMsgQueue to create a message queue for the thread.

Then, it can create one or more windows by calling one of the window-creation functions, such as WinCreateWindow.

The window-creation functions require that the following information be supplied in some form:

- Class
- Styles
- Name
- Parent window
- Position relative to the parent window
- Position relative to any sibling windows (z-order)
- Dimensions
- Owner window
- Identifier
- Class-specific data
- Resources.

Every window belongs to a *window class* that defines that window's appearance and behavior. The chief component of the window class is the *window procedure*. The window procedure is the function that receives and processes all messages sent to the window.

Every window has a *style*. The window style specifies aspects of a window's appearance and behavior that are not specified by the window's class. For example, the WC_FRAME class always creates a frame window, but the FS_BORDER, FS_DLGBORDER, and FS_SIZEBORDER styles determine the style of a frame window's border. A few window styles apply to all windows, but most apply only to windows of specific window classes. The window procedure for a given class interprets the style and allows an application to adapt a window of a given class for a special circumstance. For example, an application can give a window the style WS_SYNCPAINT to cause it to be painted immediately whenever any portion of the window becomes invalid. Normally, a window is painted only if there are no messages waiting in the message queue.

A window can have a text string associated with it. Typically, the window text is displayed in the window or in a title bar. The class of window determines whether the window displays the text and, if so, where the text appears within the window.

Every window except the desktop window and desktop-object window has a *parent window*. The parent provides the coordinate system used to position the window and also affects aspects of a window's appearance. For example, when the parent window is minimized, hidden, or destroyed, the parent's child windows are minimized, hidden, or destroyed also.

Every window has a screen position, size, and z-order position. The *screen position* is the location of the window's lower-left corner, relative to the lower-left corner of its parent window. A window's size is its width and height, measured in pels. A window's *z-order position* is the position of the window in the order of overlapping windows. This viewing order is oriented along an imaginary axis, the *z* axis, extending outward from the screen. The window at the top of the z-order overlaps all *sibling* windows (that is, windows having the same parent window). A window at the bottom of the z-order is overlapped by all sibling windows. An application sets a window's z-order position by placing it behind a given sibling window or at the top or bottom of the z-order of the windows.

A window can own, or be owned by, another window. The owner-owned relationship affects how messages are sent between windows, allowing an application to create combinations of windows that work together. A window issues messages about its state to its owner window; the owner window issues messages back about what action to perform next.

The *window handle* is a unique number across the system that is totally unambiguous—it identifies one particular window in the system and is assigned by the system. A *window identifier* is analogous to a "given" name in family relationships; the only requirement is that the name be unique among siblings.

A window can have class-specific data that further defines how the window appears and behaves when it is created. The system passes the class-specific data to the window procedure, which then applies the data to the new window.

Window-Creation Functions

The basic window-creation function is WinCreateWindow. This function uses information about a window's class, style, size, and position to create a new window. All other window-creation functions, such as WinCreateStdWindow and WinCreateDlg, supply some of this information by default and create windows of a specific class or style.

Although the WinCreateWindow function provides the most direct means of creating a window, most applications do not use it. Instead, they often use the WinCreateStdWindow function to create a main window and the WinDlgBox or WinCreateDlg functions to create dialog windows.

The WinCreateMenu, WinLoadMenu, WinLoadDlg, WinMessageBox, and WinCreateFrameControls functions also create windows. Each of these functions substitutes for one or more required calls to WinCreateWindow to create a given window. For example, an application can create a frame window, one or more control windows, and a client window in a single call to WinCreateStdWindow.

Window-Creation Messages

While creating a window, the system sends messages to that window's window procedure. The window procedure receives a WM_CREATE message, saying that the window is being created. The window also receives a WM_ADJUSTWINDOWPOS message, specifying the initial size and position of the window being created. This message lets the window procedure adjust the size and position of the window before the window is displayed.

The system also sends other messages while creating a window; the number and order of these messages depend on the class and style of the window and the function used to create it.

Window Classes

Each window of a specific window class uses the window procedure associated with that class. An application can create one or more windows that belong to the same window class. Because each window of the same class is processed by the same window procedure, they all behave the same way. Since many windows can result from one window procedure, coding overhead is greatly reduced. There are two types of window classes: public and private.

Public Window Classes

A *public window class* is one that has a reentrant window procedure that is registered and resides in a dynamic link library (DLL); it can be used by any process in the system to create windows. The operating system provides several preregistered public window classes. You can specify the system-provided window classes by using the symbolic identifiers that have the prefix WC_, as shown in the following table:

Table 2-1 (Page 1 of 2). Window Classes

Class Name	Description
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_CONTAINER	Creates a control for the user to group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as menu bars, or vertically as pull-down menus. Menus usually are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.
WC_SCROLLBAR	Lets the user scroll the contents of an associated window.
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.

Table 2-1 (Page 2 of 2). Window Classes

Class Name	Description
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to the radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

With the exception of WC_FRAME, the system-provided window classes are known as *control window classes* because they give the user an easy means of controlling specific types of interaction. For example, the WC_BUTTON class allows single or multiple selections. These windows conform to the IBM* Systems Application Architecture (SAA) Common User Access (CUA) definition. They are designed specifically to provide function that meets the needs for a graphics-based standard user interface. The code fragments provided in this guide make extensive use of the system window classes.

Private Window Classes

A *private window class* is one that an application registers for its own use; it is available only to the process that registers it. The application-provided window procedure for a private window class resides either in the application's executable files or in a DLL file. A private window class is deleted when its registering process is terminated.

Window Styles

A window can have a combination of styles; an application can combine styles by using the bitwise inclusive OR operator. An application usually sets the window styles when it creates the window. The OS/2 operating system provides several standard window styles that apply to all windows. It also provides many styles for the predefined frame and control windows. The frame and control styles are unique to each predefined window class and can be used only for windows of the corresponding class.

Initially, the styles of the window class used to create the window determine the styles of the new window. For example, if the window class has the style CS_SYNCPAINT, all windows created using that class, by default, will have the window style WS_SYNCPAINT.

The OS/2 operating system has the following standard window styles:

Table 2-2 (Page 1 of 2). Standard Window Styles

Style Name	Description
WS_CLIPCHILDREN	Prevents a window from painting over its child windows. This style increases the time necessary to calculate the visible region. This style is usually not necessary because if the parent and child windows overlap and both are invalidated, the system draws the parent window before drawing the child window. If the child window is invalidated independently of the parent window, the system redraws only the child window. If the update region of the parent window does not intersect the child window, drawing the parent window causes the child window to be redrawn. This style is useful to prevent a child window that contains a complex graphic from being redrawn unnecessarily. WS_CLIPCHILDREN is an absolute requirement if a window with children ever performs output in response to any message other than WM_PAINT. Only WM_PAINT processing is synchronized such that the children will get their messages after the parent.
WS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows. This style protects sibling windows but increases the time necessary to calculate the visible region. This style is appropriate for windows that overlap and that have the same parent window.
WS_DISABLED	Used by an application to disable a window. It is up to the window to recognize this style and reject input.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the ARROW keys. All controls defined after the control with the WS_GROUP style belong to the same group. The next control with the WS_GROUP style ends the first group and starts a new group.
WS_MAXIMIZED	Enlarges a window to the maximum size.
WS_MINIMIZED	Reduces a window to the size of an icon.
WS_PARENTCLIP	Extends a window's visible region to include that of its parent window. This style simplifies the calculation of the child window's visible region but is potentially dangerous because the parent window's visible region is usually larger than the child window.
WS_SAVEBITS	Saves the screen area under a window as a bit map. When the user hides or moves the window, the system restores the image by copying the bits; there is no need to add the area to the uncovered window's update region. The style can improve system performance but also can consume a great deal of memory. It is recommended only for transient windows, such as menus and dialog windows, not for main application windows.
WS_SYNCPAINT	Causes a window to receive WM_PAINT messages immediately after a part of the window becomes invalid. Without this style, the window receives WM_PAINT messages only if no other message is waiting to be processed.

Table 2-2 (Page 2 of 2). Standard Window Styles

Style Name	Description
WS_TABSTOP	Specifies one of any number of controls through which the user can move by tabbing. Pressing the TAB key moves the keyboard focus to the next control that has the WS_TABSTOP style.
WS_VISIBLE	Makes a window visible. The operating system draws the window on the screen unless overlapping windows completely obscure it. Windows without this style are hidden. If overlapping windows completely obscure the window, the window is still considered visible. (*Visibility* means that the operating system draws the window if it can.)

Window Handles

After creating a window, the creation function returns a window handle that uniquely identifies the window. An application can use this handle to direct the action of functions to the window. Window handles have the data type HWND; applications must use this data type when declaring variables that hold window handles.

There are special constants that an application can use instead of a window handle in certain functions. For example, an application can use HWND_DESKTOP in the WinCreateWindow function to specify the desktop window as the new window's parent. Similarly, HWND_OBJECT represents the desktop-object window. HWND_TOP and HWND_BOTTOM represent the top and bottom positions relative to the z-order position of a window.

Although the NULL constant is not a window handle, an application can use it in some functions to specify that no window is affected. For example, an application can use NULL in the WinCreateWindow function to create a window that has no owner window. Some functions might return NULL, indicating that the given action applies to no window.

Window Size and Position

A window's size and position can be expressed as a bounding rectangle, given in coordinates relative to its parent. An application specifies the window's initial size and position when creating the window.

To use the system-default values for the initial size and position of a frame window, an application can specify the FCF_SHELLPOSITION frame-creation flag. The application can change a window's size and position at any time. Figure 2-6 on page 2-17 indicates the size and position coordinates of a parent window and a child window.

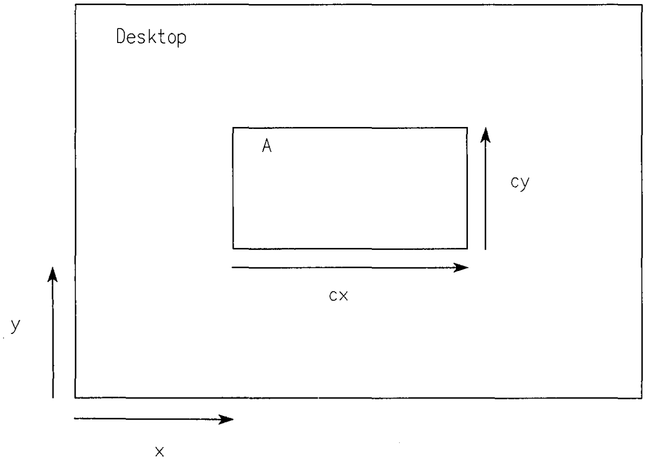


Figure 2-6. Window Sizing and Positioning

Notes:

1. The default coordinate system for a window specifies that the point (0,0) is at the lower-left corner of the window, with coordinates increasing as they go upward and to the right.
2. A window can be positioned anywhere in relation to its parent.

Size

A window's *size* (width and height) is given in pels, in the range 0 through 65535. A window can have 0 width and height; however, a window with 0 width or height is not drawn on the screen, even though it has the WS_VISIBLE style.

An application can create very large windows; however, it should check the size of the screen before enlarging a window size. One way to choose an appropriate size is to use the WinGetMaxPosition function to retrieve the size of the maximized window. A window that is larger than its maximized size will be larger than the screen also.

An application can retrieve the current size of the window by using the WinQueryWindowRect function.

Position

A window's *position* is defined as the *x,y* coordinates of its lower-left corner. These coordinates, sometimes called *window coordinates*, always are relative to the lower-left corner of the parent window. For example, a window having the coordinates (10,10) is placed 10 pels to the right of, and 10 pels up from, the lower-left corner of its parent window. Notice, however, that a window can be positioned anywhere in relation to its parent, but always relative to the parent's lower-left corner.

Adjusting a window's position can improve drawing performance. For example, an application could position a window so that its horizontal position is a multiple of 8, relative to the screen *origin* (the lower-left corner of the screen). Coordinates that are multiples of 8 correspond to byte boundaries in the screen-memory bit map. It is usually faster to start drawing at a byte boundary.

By default, the system positions a frame window on a byte boundary; but an application can override this action by using the FCF_NOBYTEALIGN style when creating the window.

Size and Position Messages

A window receives messages when it changes size or position. Before a change is made, the system might send a WM_ADJUSTWINDOWPOS message to allow the window procedure to make final adjustments to the window's size and position. This message includes a pointer to an SWP structure that contains the requested width, height, and position. If the window procedure adjusts these values in the structure, the system uses the adjusted values to redraw the window. The WM_ADJUSTWINDOWPOS message is not sent if the change is a result of a call to the WinSetWindowPos function with the SWP_NOADJUST constant specified.

After a change has been made to a window, the system sends a WM_SIZE message to specify the new size of the window. If the window has the class style CS_MOVENOTIFY, the system also sends a WM_MOVE message, which includes the new position for the window. The system sends a WM_SHOW message if the visibility of the window has changed.

System Commands

An application that has a window with a system menu can change the size and position of that window by sending system commands. The system commands are generated when the user chooses commands from the system menu. An application can emulate the user action by sending a WM_SYSCOMMAND message to the window.

Following are some of the system commands:

Table 2-3. System Commands

Command	Description
SC_SIZE	Starts a Size command. The user can change the size of the window with a mouse and the keyboard.
SC_MOVE	Starts a Move command. The user can move the window with a mouse and the keyboard.
SC_MINIMIZE	Minimizes the window.
SC_MAXIMIZE	Maximizes the window.
SC_RESTORE	Restores a minimized or maximized window to its previous size and position.
SC_CLOSE	Closes the window. This command sends a WM_CLOSE message to the window. The window performs all tasks needed to clean up and destroy itself.

Window Data

Every window has an associated data structure. The window data structure contains all the information specified for the window at the time it was created and any additional information supplied for the window since that time. Although the exact size and meaning of the information in the window data structure are private to the system, an application can access any of the following data items via system-provided functions:

- Pointer to window-instance data structure
- Pointer to window procedure
- Parent-window handle
- Owner-window handle
- Handle of first child window
- Handle of next sibling window
- Window size and position (expressed as a rectangle)
- Window style
- Window identifier
- Update-region handle
- Message-queue handle.

An application can examine and modify this data by using functions such as WinQueryWindowUShort and WinSetWindowUShort. These functions let an application access data that is stored as 16-bit integers. Other functions let an application access data containing 32-bit integers and pointers. Several functions indirectly affect the data items in the window data structure. For example, the WinSubclassWindow function replaces the window-procedure pointer, and the WinSetWindowPos function changes the size and position of the window.

An application can extend the number of available data items in the window data structure by specifying a count of extra bytes when it registers the corresponding window class. Then, the window procedure can use these bytes to store information about the window. The WinQueryWindowUShort and WinSetWindowUShort functions give direct access to the extra bytes.

It generally is not a good idea to use direct storage in the window data. It is better to allocate a data structure dynamically and set a pointer to that data structure in the window words. This provides two advantages:

1. Most importantly, it is a symbolic way of referencing the data structure. It is very easy to make mistakes and provide the wrong offsets to WinQueryWindowUShort and so forth.
2. You now can add and remove fields without cross dependencies because you now use *symbolic* references; whereas, when you use the technique of putting window words directly in the window data structure, you have to account for changed offsets.

Window Resources

Window resources are read-only data segments stored in an application's EXE file or in a dynamic link library's DLL file. Predefined PM window resources include keyboard accelerator tables, icons, menus, bit maps, dialog boxes, and so forth; these are not a regular part of the application window's code and data. Because, in most cases, window resources are not loaded into memory when the operating system runs a program, the resources can be shared by multiple instances of the same application.

Most window resources are stored in a format that is unique to each resource type. The application does not need to know these formats because the system translates them, as necessary, for use in PM functions. The following table lists the ten most commonly used PM window resource types.

Table 2-4. Presentation Manager-Defined Resource Types

Resource Identifier	Description
RT_ACCELTABLE	Keyboard accelerator table
RT_BITMAP	Bit map
RT_DIALOG	Dialog box template
RT_FONT	Font
RT_FONTDIR	Font directory
RT_MENU	Menu template
RT_MESSAGE	Message string
RT_POINTER	Icon or mouse
RT_RCDATA	Programmer-defined data
RT_STRING	Text string

To access these resources, you must prepare a *resource file* (ASCII file with the extension .RC). Then the ASCII resource file must be compiled into binary images using the resource compiler. The compiled resource file extension is RES; it can be linked into your program's EXE file or to a dynamic link library's DLL file.

Maximized and Minimized Windows

A *maximized window* is a window that has been enlarged to fill the screen. Although a window's size can be set so that it fills the screen exactly, a maximized window is slightly different: the system automatically moves the window's title bar to the top of the screen and sets the WS_MAXIMIZED style for the window.

A *minimized window* is a window whose size has been reduced to exactly the size of an icon *or*, in the Workplace Shell*, it disappears altogether (by default). Like a maximized window, a minimized window is more than just a window of a given size; typically, the system moves the (icon) minimized window to the lower part of the screen and sets the WS_MINIMIZED style for that window. The lower part of the screen is sometimes called the *icon area.* Unless the application specifies another position, the system moves a minimized window into the first available icon position in the icon area.

If a window is created with the WS_MAXIMIZED or WS_MINIMIZED styles, the system draws the window as a maximized or minimized window.

An application can restore maximized or minimized windows to their previous size and position by specifying the SWP_RESTORE flag in a call to the WinSetWindowPos function.

Window Visibility

A window that is a descendant of the desktop window can be either visible or invisible. The system displays a visible window on the screen. It hides an invisible window by not drawing it. If a window is visible, the user can supply input to the window and view the window's output. If a window is invisible, the window, in effect, is disabled. An invisible window can process messages from the system or from other windows, but it cannot process user input or display output. An application sets a window's visibility state when it creates the window. Later, a user or the application can change the visibility state.

The visible region of a window is the position clipped by any overlapping windows. These overlapping windows can be child windows or other main windows in the system. The visible region is defined by a set of one or more rectangles, as shown in Figure 2-7.

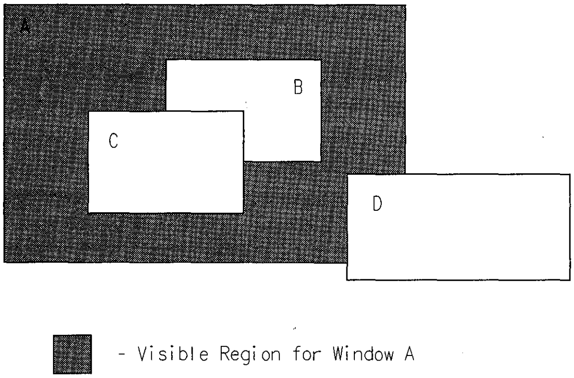


Figure 2-7. Visible Region for Window A

A window is visible if the WS_VISIBLE style is set for the window. By default, the WinCreateWindow function creates invisible windows unless the application specifies WS_VISIBLE. The application often hides a window to keep its operational details from the user. For example, an application can keep a new window invisible while it customizes the window's appearance. An application can determine whether a window has the WS_VISIBLE style by using the WinIsWindowVisible function.

Even if a window has the WS_VISIBLE style, the user might not be able to see the window on the screen because other windows completely overlap it, or it might have been moved beyond the edge of its parent. A visible window is subject to the clipping rules established by its parent-child relationship. If the window's parent window is not visible, the window will not be visible. Because a child window is drawn relative to its parent's lower-left corner, if the parent window is moved beyond the edge of the screen, the child window also will be moved. In other words, if a user moves the parent window containing the child window far enough off the edge of the screen, the user will not be able to see the child window, even though the child window and its parent window have the WS_VISIBLE style. To determine whether the user actually can see a window, an application can use the WinIsWindowShowing function.

Window Destruction

In general, an application must destroy all the windows it creates. It does this by using the WinDestroyWindow function. When a window is destroyed, the system hides the window, if it is visible, and then removes any internal data associated with the window. This invalidates the window handle so that it can no longer be used by the application.

An application destroys many of the windows it creates soon after creating them. For example, an application usually destroys a dialog window as soon as the application has sufficient input from the user to continue its task. An application eventually destroys the main window of the application (before terminating).

Destroying a window does not affect the window class from which the window was created. New windows still can be created using that class, and any existing windows of that class continue to operate.

When the application calls WinDestroyWindow, the system searches the descendancy tree for all windows below the specified window and destroys them from the bottom up, so each child receives WM_DESTROY before its parent. Each destroyed window is responsible for cleaning up its own resources in response to the WM_DESTROY message.

If a presentation space was created by the WinGetPS function for any of the windows to be destroyed, it must be released by calling the WinReleasePS function. The application must do this before calling the WinDestroyWindow function. If a presentation space is associated with the device context for the window, the application must disassociate or destroy the presentation space by using the GpiAssociate or GpiDestroyPS function before calling WinDestroyWindow. Failing to release a resource can cause an error.

For more information about presentation spaces and device contexts, see Chapter 7, "Painting and Drawing."

If the window being destroyed is the active window, both the active and focus states are transferred to another window. The window that becomes the active window is the next window, as determined by the Alt+Esc key combination. The new active window then determines which window receives the keyboard focus.

Using Windows

The following sections explain how to create and use windows in an application, how to manage ownership and parent-child window relationships, and how to move and size windows.

Creating a Top-Level Frame Window

The main window in most applications is a top-level frame window. An application creates a top-level frame window by specifying the handle of the desktop window, or HWND_DESKTOP, as the hwndParent parameter in a call to the WinCreateStdWindow function.

Figure 2-8 on page 2-24 shows the main() function for a simple PM application. This function initializes the application, creates a message queue, and registers the window class for the client window before creating a top-level frame window.

```
#define IDR_RESOURCES 1

MRESULT EXPENTRY ClientWndProc(HWND, ULONG, MPARAM, MPARAM);

int main(VOID)
{
    HWND hwndFrame;
    HWND hwndClient;
    HMQ  hmq;
    QMSG qmsg;
    HAB  hab;

    /* Set the frame-window creation flags.                   */
    ULONG flFrameFlags =
        FCF_TITLEBAR      |   /* Title bar                    */
        FCF_SIZEBORDER    |   /* Size border                  */
        FCF_MINMAX        |   /* Minimize and maximize buttons.  */
        FCF_SYSMENU       |   /* System menu                  */
        FCF_SHELLPOSITION |   /* System-default size and position */
        FCF_TASKLIST ;        /* Add name to Task List.       */

    /* Initialize the application for PM                      */
    hab = WinInitialize(0);

    /* Create the application message queue.                  */
    hmq = WinCreateMsgQueue(hab, 0);

    /* Register the class for the client window.              */
    WinRegisterClass(
        hab,                    /* Anchor block handle          */
        "MyPrivateClass",       /* Name of class being registered */
        (PFNWP)ClientWndProc,   /* Window procedure for class   */
        CS_SIZEREDRAW |         /* Class style                  */
        CS_HITTEST,             /* Class style                  */
        0);                     /* Extra bytes to reserve       */

    /* Create a top-level frame window with a client window   */
    /* that belongs to the window class "MyPrivateClass".     */
    hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,      /* Parent is desktop window.         */
        WS_VISIBLE,        /* Make frame window visible.        */
        &flFrameFlags,     /* Frame controls                    */
        "MyPrivateClass",  /* Window class for client           */
        NULL,              /* No window title                   */
        WS_VISIBLE,        /* Make client window visible .      */
        (HMODULE) 0,       /* Resources in application module   */
        IDR_RESOURCES,     /* Resource identifier               */
        NULL);             /* Pointer to client window handle   */

    /* Start the main message loop. Get messages from the     */
    /* queue and dispatch them to the appropriate windows.    */
    while (WinGetMsg(hab, &qmsg, 0, 0, 0))
           WinDispatchMsg(hab, &qmsg);

    /* Main loop has terminated. Destroy all windows and the  */
    /* message queue; then terminate the application.         */
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);

    return 0;
}
```

Figure 2-8. Structure of a Simple Presentation Manager Application

Creating an Object Window

An application can create an object window by using the WinCreateWindow function and setting the desktop-object window as the parent window. The code fragment in Figure 2-9 on page 2-25 shows how to create an object window.

```
#define ID_OBJWINDOW 2

HWND hwndObject;

hwndObject = WinCreateWindow(
    HWND_OBJECT,        /* Parent is object window.          */
    "MyObjClass",       /* Window class for client           */
    NULL,               /* Window text                       */
    0,                  /* No styles for object window       */
    0, 0,               /* Lower-left corner                 */
    0, 0,               /* Width and height                  */
    NULL,               /* No owner                          */
    HWND_BOTTOM,        /* Inserts window at bottom of z-order */
    ID_OBJWINDOW,       /* Window identifier                 */
    NULL,               /* No class-specific data            */
    NULL);              /* No presentation data              */
```

Figure 2-9. Creating an Object Window

Querying Window Data

An application can examine the values in the data structure associated with a window by using the WinQueryWindowUShort and WinQueryWindowULong functions. Each of these functions specifies a structure data item to examine. The index value can be an integer representing a zero-based byte index or a constant (QWS_) that identifies a specific item of data. The code fragment in Figure 2-10 obtains the programmer-defined identifier of the object window defined in the previous example:

```
HWND hwndObject;
USHORT usObjID;

usObjID = WinQueryWindowUShort(hwndObject, QWS_ID);
```

Figure 2-10. Getting the Window Identifier

Changing the Parent Window

An application can change a window's parent window by using the WinSetParent function. For example, in an application that uses child windows to display documents, you might want only the active document window to show a system menu. You can do this by changing that menu's parent window back and forth between the document window and the object window when WM_ACTIVATE messages are received. This technique is shown in the code fragment in Figure 2-11 on page 2-26.

```
switch (msg) {

case WM_ACTIVATE:  {

HWND hwndFrame, hwndSysMenu, hwnd;

    /* Get the handles of the frame window and system menu.        */
    hwndFrame = WinQueryWindow(hwnd, QW_PARENT);
    hwndSysMenu = WinWindowFromID(hwndFrame, FID_SYSMENU);

    /* If the window is being activated, make the frame window the */
    /* parent of the system menu. Otherwise, hide the system menu  */
    /* by making the object window the parent.                     */

    if ( SHORT1FROMMP(mp1))
        WinSetParent(hwndSysMenu, hwndFrame, TRUE);

    else
        WinSetParent(hwndSysMenu, HWND_OBJECT, TRUE);

    }

    return 0;
}
```

Figure 2-11. Changing the Parent Window

Finding a Parent, Child, or Owner Window

An application can determine the parent, child, and owner windows for any window by using the WinQueryWindow function. This function returns the window handle of the requested window.

The code fragment in Figure 2-12 determines the parent window of the given window:

```
HWND hwndParent;
HWND hwndMyWindow;

hwndParent = WinQueryWindow(hwndMyWindow, QW_PARENT);
```

Figure 2-12. Finding the Parent Window

The code fragment in Figure 2-13 determines the *topmost* child window (the child window in the top z-order position):

```
HWND hwndTopChild;
HWND hwndParent;

hwndTopChild = WinQueryWindow(hwndParent, QW_TOP);
```

Figure 2-13. Finding the Topmost Child Window

If a given window does not have an owner or child window, WinQueryWindow returns NULL.

Setting an Owner Window

An application can set the owner for a window by using the WinSetOwner function. Typically, after setting the owner, a window notifies the owner window of the new relationship by sending it a message.

The code fragment in Figure 2-14 shows how to set the owner window and send it a message:

```
#define NEW_OWNER 1

HWND hwndMyWindow;
HWND hwndNewOwner;

if (WinSetOwner(hwndMyWindow, hwndNewOwner))

    /* Send a notification message.                               */
    WinSendMsg(hwndNewOwner,   /* Sends to owner                   */
        WM_CONTROL,            /* Control message for notification */
        (MPARAM) NEW_OWNER,    /* Notification code                */
        NULL);                 /* No extra data                    */
```

Figure 2-14. Setting the Owner Window

A window can have only one owner, so WinSetOwner removes any previous owner.

Retrieving the Handle of a Child or Owned Window

A parent or owner window can retrieve the handle of a child or owned window by using the WinWindowFromID function and supplying the identifier of the child or owned window. WinWindowFromID searches all child and owned windows to locate the window with the given identifier. The window identifier is set when the application creates the child or owned window.

Typically, an owned window uses WinQueryWindow to get the handle of the owner window; then uses WinSendMsg to issue a notification message to its owner window.

The code fragment in Figure 2-15 on page 2-28 retrieves the window handle of an owner window and sends the window a WM_ENABLE message.

```
HWND hwndOwned;
HWND hwndOwner;

case WM_CONTROL:
    switch (SHORT2FROMMP (mp2)) {
        case BN_CLICKED:
            hwndOwned = WinWindowFromID(hwndOwner,
            (ULONG)SHORT1FROMMP(mp1));
            WinSendMsg(hwndOwned, WM_ENABLE,
            (MPARAM)TRUE, (MPARAM) NULL);
            return 0;
        .
        . /* Check for other notification codes. */
        .

        }
```

Figure 2-15. Getting a Handle to an Owner or Child Window

An application also can retrieve the handle of a child window by using the WinWindowFromPoint function and supplying a point in the corresponding parent window.

Enumerating Top-Level Windows

An application can enumerate all top-level windows in the system by using the WinBeginEnumWindows and WinGetNextWindow functions. An application also can create a list of all child windows for a given parent window using WinBeginEnumWindows. This list contains the window handles of immediate child windows. By using WinGetNextWindow, the application then can retrieve the window handles, one at a time, from the list. When the application has finished using the list, it must release the list with the WinEndEnumWindows function.

The code fragment in Figure 2-16 shows how to enumerate all top-level windows (all immediate child windows of the desktop window):

```
HWND hwndTop;
HENUM henum;

/* Enumerate all top-level windows.           */

henum = WinBeginEnumWindows(HWND_DESKTOP);

/* Loop through all enumerated windows.       */
while (hwndTop = WinGetNextWindow(henum)) {
    .
    . /* Perform desired task on each window. */
    .
}

WinEndEnumWindows(henum);
```

Figure 2-16. Enumerating Top-Level Windows

Moving and Sizing a Window

An application can move a window by using the WinSetWindowPos function and specifying the SWP_MOVE constant. The function changes the position of the window to the specified position. The position is always given in coordinates relative to the parent window.

The code fragment in Figure 2-17 moves the window to the position (10,10):

```
HWND hwnd;

WinSetWindowPos(
    hwnd,                 /* Window handle                 */
    NULL,                 /* Not used for moving and sizing */
    10, 10,               /* New position                  */
    0, 0,                 /* Not used for moving           */
    SWP_MOVE);            /* Move window                   */
```

Figure 2-17. Moving a Window

An application can set the size of a window by using the WinSetWindowPos function and specifying the SWP_SIZE constant. WinSetWindowPos changes the width and height of the window to the specified width and height.

An application can combine moving and sizing in a single function call, as shown in Figure 2-18.

```
HWND hwnd;

WinSetWindowPos(
    hwnd,                 /* Window handle                 */
    NULL,                 /* Not used for moving and sizing */
    10, 10,               /* New position                  */
    200, 200,             /* Width and height              */
    SWP_MOVE | SWP_SIZE); /* Move and size window.         */
```

Figure 2-18. Moving and Sizing a Window

An application can retrieve the current size and position of a window by using the WinQueryWindowPos function. This function copies the current information to an SWP structure.

The code fragment in Figure 2-19 on page 2-30 uses the current size and position to change the height of the window, leaving the width and position unchanged.

```
HWND hwnd;
SWP swp;

WinQueryWindowPos(hwnd, &swp);
WinSetWindowPos(
    hwnd,                 /* Window handle                */
    NULL,                 /* Not used for moving and sizing */
    0, 0,                 /* Not used for sizing          */
    swp.cx,               /* Current width                */
    swp.cy + 200,         /* New height                   */
    SWP_SIZE);            /* Change the size.             */
```

Figure 2-19. Changing the Size of a Window

An application also can move and change the size of several windows at once by using the WinSetMultWindowPos function. This function takes an array of SWP structures. Each structure specifies the window to be moved or changed.

An application can move and size a window even if it is not visible, although the user is not able to see the effects of the moving and sizing until the window is visible.

Redrawing Windows

When the system moves a window or changes its size, it can invalidate all or part of that window. The system attempts to preserve the contents of the window and copy them to the new position; however, if the window's size is increased, the window must fill the area exposed by the size change. If a window is moved from behind an overlapping window, any area formerly obscured by the other window must be drawn. In these cases, the system invalidates the exposed areas and sends a WM_PAINT message to the window.

An application can require that the system invalidate an entire window every time the window moves or changes size. To do this, the application sets the CS_SIZEREDRAW class style in the corresponding window class. Typically, this class style is selected for use in an application that uses a window's current size and position to determine how to draw the window. For example, a clock application always would draw the face of the clock so that it filled the window exactly.

An application also can explicitly specify which parts of the window to preserve during a move or size change. Before any change is made, the system sends a WM_CALCVALIDRECTS message to windows that do not have the style CS_SIZEREDRAW. This enables the window procedure to specify what part of the window to save and where to align it after the move or size change.

Changing the Z-Order of Windows

An application can move a window to the top or bottom of the z-order by passing the SWP_ZORDER constant to the WinSetWindowPos function. An application specifies where to move the window by specifying the HWND_TOP or HWND_BOTTOM constants.

The code fragment in Figure 2-20 uses WinSetWindowPos to change the z-order of a window.

```
HWND hwndParent;
HWND hwndNext;
HENUM henum;

    WinSetWindowPos(
        hwndNext,       /* Next window to move  */
        HWND_TOP,       /* Put window on top    */
        0, 0, 0, 0,     /* Not used for z-order */
        SWP_ZORDER);    /* Change z-order       */
```

Figure 2-20. Changing the Z-order of a Window

An application also can specify the window that the given window is to move behind. In this case, the application specifies the window handle instead of the HWND_TOP or HWND_BOTTOM constant.

```
HWND hwndParent;
HWND hwndNext;
HWND hwndExchange;
HENUM henum;

henum = WinBeginEnumWindows(hwndParent);

hwndExchange = WinGetNextWindow(henum);

                    /* hwndNext has top window;
        hwndExchange has window under the top. */

WinSetWindowPos(
    hwndNext,       /* Next window to move     */
    hwndExchange,   /* Put lower window on top */
    0, 0, 0, 0,     /* Not used for z-order    */
    SWP_ZORDER);    /* Change z-order          */

WinEndEnumWindows(henum);
```

Figure 2-21. Exchanging the Z-order of Windows

Showing or Hiding a Window

An application can show or hide a window by using the WinShowWindow function. This function changes the WS_VISIBLE style of a window to the specified setting. An application can also use the WinIsWindowVisible function to check the visibility of a window. This function returns TRUE if the window is visible.

Maximizing, Minimizing, and Restoring a Frame Window

An application can maximize, minimize, or restore a frame window by using the WinSetWindowPos function and specifying the constant SWP_MAXIMIZE, SWP_MINIMIZE, or SWP_RESTORE. Only a frame window can maximize and minimize by default. For any other window, an application must provide support for these actions in the corresponding window procedure.

Figure 2-22 on page 2-32 shows how to maximize a frame window.

```
SWP swpCurrent;
HWND hwndFrame;

WinQueryWindowPos(hwndFrame, &swpCurrent);
WinSetWindowPos(
    hwndFrame,           /* Window handle               */
    NULL,                /* Not used to maximize        */
    swpCurrent.x,
    swpCurrent.y,        /* Stored for restoring window */
    swpCurrent.cx,
    swpCurrent.cy,       /* Stored for restoring window */
    SWP_MAXIMIZE | SWP_SIZE | SWP_MOVE);    /* Maximize */
```

Figure 2-22. Maximizing a Frame Window

Destroying a Window

An application can destroy a window by using the WinDestroyWindow function. Figure 2-23 shows how to create and then destroy a control window:

```
HWND hwndCtrl;
HWND hwndParent;

hwndCtrl = WinCreateWindow(hwndParent, WC_BUTTON, ...);

WinDestroyWindow(hwndCtrl);
```

Figure 2-23. Destroying a Window

Related Functions

This section covers the functions that are related to Windows.

WinBeginEnumWindows

This function begins the enumeration process for all of the immediate child windows of a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HENUM WinBeginEnumWindows  (HWND hwnd)
```

Parameters

hwnd (HWND) – input

Handle of the window whose child windows are to be enumerated.

HWND_DESKTOP	Enumerate all main windows
HWND_OBJECT	Enumerate all object windows
Other	Enumerate all immediate children of the specified window.

Returns

henumHenum (HENUM) – returns

Enumeration handle.

WinCreateStdWindow

This function creates a standard window.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>

HWND WinCreateStdWindow (HWND hwndParent, ULONG flStyle,
                         PULONG pflCreateFlags, PSZ pszClassClient,
                         PSZ pszTitle, ULONG flStyleClient,
                         HMODULE Resource, ULONG ulId,
                         PHWND phwndClient)
```

Parameters

hwndParent (HWND) – input
Parent-window handle.

flStyle (ULONG) – input
Frame-window style.

pflCreateFlags (PULONG) – input
Frame-creation flags.

pszClassClient (PSZ) – input
Client-window class name.

pszTitle (PSZ) – input
Title-bar text.

flStyleClient (ULONG) – input
Client-window style.

Resource (HMODULE) – input
Resource identifier.

NULLHANDLE Resource definitions are contained in the application .EXE file.

Other The module handle returned by the DosLoadModule or DosQueryModuleHandle call of the Dynamic Link Library (DLL) containing the resource definitions.

ulId (ULONG) – input
Frame-window identifier.

phwndClient (PHWND) – output
Client-window handle.

Returns

hwndFrame (HWND) – returns
Frame-window handle.

WinCreateWindow

This function creates a new window of class *pszClass* and returns *hwnd*.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinCreateWindow  (HWND hwndParent, PSZ pszClass, PSZ pszName,
                       ULONG flStyle, LONG x, LONG y, LONG cx, LONG cy,
                       HWND hwndOwner, HWND hwndInsertBehind,
                       ULONG id, PVOID pCtlData, PVOID pPresParams)
```

Parameters

hwndParent (HWND) – input
Parent-window handle.

pszClass (PSZ) – input
Registered-class name.

pszName (PSZ) – input
Window text.

flStyle (ULONG) – input
Window style.

x (LONG) – input
x-coordinate of window position.

y (LONG) – input
y-coordinate of window position.

cx (LONG) – input
Width of window, in window coordinates.

cy (LONG) – input
Height of window, in window coordinates.

hwndOwner (HWND) – input
Owner-window handle.

hwndInsertBehind (HWND) – input
Sibling-window handle.

id (ULONG) – input
Window identifier.

pCtlData (PVOID) – input
Pointer to control data.

pPresParams (PVOID) – input
Presentation parameters.

Returns

hwnd (HWND) – returns
Window handle.

NULLHANDLE	Error occurred
Other	Window handle.

WinDestroyWindow

This call destroys a window and its child windows.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinDestroyWindow  (HWND hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Window-destroyed indicator.

TRUE	Window destroyed
FALSE	Window not destroyed.

WinEnableWindow

This function sets the window enabled state.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinEnableWindow  (HWND hwnd, BOOL fNewEnabled)
```

Parameters

hwnd (HWND) – input
Window handle.

fNewEnabled (BOOL) – input
New enabled state.

TRUE	Set window state to enabled
FALSE	Set window state to disabled.

Returns

rc (BOOL) – returns
Window enabled indicator.

TRUE	Window enabled state successfully updated
FALSE	Window enabled state not successfully updated.

WinEndEnumWindows

This function ends the enumeration process for a specified enumeration.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinEndEnumWindows  (HENUM  henum)
```

Parameters

henum (HENUM) – input
Enumeration handle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinGetMaxPosition

The WinGetMaxPosition function fills an SWP structure with the maximized-window size and position.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinGetMaxPosition  (HWND hwnd, PSWP pswp)
```

Parameters

hwnd (HWND) – input
Frame-window handle.

pswp (PSWP) – output
Set window position structure.

Returns

fSuccess (BOOL) – returns
Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

WinGetMinPosition

This function returns the position to which a window is minimized.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinGetMinPosition  (HWND  hwnd, PSWP  pswp, PPOINTL  pptl)
```

Parameters

hwnd (HWND) – input
Frame-window handle.

pswp (PSWP) – output
Set window position structure.

pptl (PPOINTL) – input
Preferred position.

NULL   System is to choose the position
Other   System is to choose the position nearest to the specified point.

Returns

rc (BOOL) – returns
Success indicator.

TRUE   Successful completion.

The WS_MINIMIZE style is set for *hwnd*. This enables the system to determine which other frame windows are minimized, during the enumeration process performed by this function.

Also, the window words QWS_XMINIMIZE and QWS_YMINIMIZE for *hwnd* are initialized. This enables the system to ensure that no windows that have been, or are being, minimized use the same position.

FALSE   Error occurred.

WinGetNextWindow

This function gets the window handle of the next window in a specified enumeration list.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinGetNextWindow  (HENUM  henum)
```

Parameters

henum (HENUM) – input
Enumeration handle.

Returns

hwndNext (HWND) – returns
Next window handle in enumeration list.

NULLHANDLE	Error occurred, *henum* was invalid, or all the windows have been enumerated.
Other	Next window handle.

WinInitialize

This function initializes the PM programming interface facilities for use by an application.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
HAB WinInitialize  (ULONG  flOptions)
```

Parameters

flOptions (ULONG) – input
Initialization options.

0 The initial state for newly created windows is that all messages for the window are available for processing by the application.

This is the only option available in PM programming interface.

Returns

hab (HAB) – returns
Anchor-block handle.

NULLHANDLE	An error occurred.
Other	Anchor-block handle.

WinIsChild

This function tests if one window is a descendant of another window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsChild  (HWND hwnd, HWND hwndParent)
```

Parameters

hwnd (HWND) – input
Child-window handle.

hwndParent (HWND) – input
Parent-window handle.

Returns

fRelated (BOOL) – returns
Related indicator.

TRUE	Child window is a descendant of the parent window, or is equal to it
FALSE	Child window is not a descendant of the parent, or is an Object Window (even if *hwndParent* is specified as the desktop or HWND_DESKTOP), or an error occurred.

WinIsThreadActive

This function determines whether the active window belongs to the calling execution thread.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinIsThreadActive  (HAB  hab)
```

Parameters

hab (HAB) – input

Anchor-block handle of calling thread.

Returns

rc (BOOL) – returns

Active-window indicator.

TRUE   Active window belongs to calling thread
FALSE   Active window does not belong to calling thread.

WinIsWindow

This function determines if a window handle is valid.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsWindow  (HAB hab, HWND hwnd)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Validity indicator.

TRUE	Window handle is valid
FALSE	Window handle is not valid.

WinIsWindowEnabled

This function returns the enabled/disabled state of a window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinIsWindowEnabled  (HWND hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Enabled-state indicator.

TRUE	Window is enabled
FALSE	Window is not enabled.

WinIsWindowShowing

This function determines whether any part of the window *hwnd* is physically visible.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>

BOOL WinIsWindowShowing  (HWND  hwnd)
```

Parameters

hwnd (HWND) – input

Window handle.

Returns

rc (BOOL) – returns

Showing state indicator.

TRUE	Some part of the window is displayed on the screen
FALSE	No part of the window is displayed on the screen.

WinIsWindowVisible

This function returns the visibility state of a window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsWindowVisible  (HWND  hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Visibility-state indicator.

TRUE Window and all its parents have the WS_VISIBLE style bit set on
FALSE Window or one of its parents have the WS_VISIBLE style bit set off.

WinMultWindowFromIDs

This function finds the handles of child windows that belong to a specified window and have window identities within a specified range.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinMultWindowFromIDs  (HWND hwndParent, PHWND prghwnd,
                            ULONG idFirst, ULONG idLast)
```

Parameters

hwndParent (HWND) – input
Parent-window handle.

prghwnd (PHWND) – output
Window handles.

idFirst (ULONG) – input
First window identity value in the range (inclusive).

idLast (ULONG) – input
Last window identity value in the range (inclusive).

Returns

lWindows (LONG) – returns
Number of window handles returned.

0	No window handles returned
Other	Number of window handles returned.

WinQueryActiveWindow

This function returns the active window for HWND_DESKTOP, or other parent window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinQueryActiveWindow  (HWND  hwndParent)
```

Parameters

hwndParent (HWND) – input

Parent-window handle for which the active window is required.

HWND_DESKTOP   The desktop-window handle that causes this function to return the top-level frame window.

Other   Specified parent-window handle.

Returns

hwndActive (HWND) – returns

Active-window handle.

NULLHANDLE   No window is active

Other   Active-window handle.

WinQueryDesktopWindow

This function returns the desktop-window handle.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

HWND WinQueryDesktopWindow (HAB hab, HDC hdc)

Parameters

hab (HAB) – input
Anchor-block handle.

hdc (HDC) – input
Device-context handle.

NULLHANDLE	Default device (the screen).

Returns

hwndDeskTop (HWND) – returns
Desktop-window handle.

NULLHANDLE	Error occurred
Other	Desktop-window handle.

WinQueryFocus

This function returns the focus window. It is NULLHANDLE if there is no focus window.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinQueryFocus  (HWND  hwndDeskTop)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hwndFocus (HWND) – returns
Focus-handle.

NULL	Error occurred or no focus window.

WinQueryObjectWindow

This function returns the desktop object window handle.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinQueryObjectWindow  (HWND  hwndDesktop)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hwndObject (HWND) – returns
Object-window handle.

NULLHANDLE	Error occurred.

WinQuerySysModalWindow

This function returns the current system modal window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinQuerySysModalWindow  (HWND  hwndDesktop)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hwndSysModal (HWND) – returns
Handle of system modal window.

NULLHANDLE	No system modal window
Other	Handle of system modal window.

WinQueryWindow

This function returns the handle of a window that has a specified relationship to a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinQueryWindow  (HWND hwnd, LONG lCode)
```

Parameters

hwnd (HWND) – input
Handle of window to query.

lCode (LONG) – input
Type of window information.

QW_NEXT — Next window in z-order (window below).

QW_PREV — Previous window in z-order (window above).

QW_TOP — Topmost child window.

QW_BOTTOM — Bottommost child window.

QW_OWNER — Owner of window.

QW_PARENT — Parent of window.

QW_NEXTTOP — Returns the next window of the owner window hierarchy subject to their z-ordering.

The enumeration is evaluated in this order:

1. The hierarchy of windows owned by this window in their z-order.
2. The hierarchy of windows of the next z-ordered window having the same owner as this window.
3. The hierarchy of windows in their z-order having the same owner as the owner of this window. This step is repeated until the top of the owner tree for this window is reached.
4. The hierarchy of windows in their z-order of unowned windows.

QW_PREVTOP — Returns the previous main window, in the enumeration order defined by QW_NEXTTOP.

QW_FRAMEOWNER — Returns the owner of *hwnd* normalized so that if shares the same parent as *hwnd*.

Returns

hwndRelated (HWND) – returns

Window handle.

WinQueryWindowPos

This function queries the window size and position of a visible window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinQueryWindowPos  (HWND hwnd, PSWP pswp)
```

Parameters

hwnd (HWND) – input
Window handle.

pswp (PSWP) – output
SWP structure.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQueryWindowPtr

This function retrieves a pointer value from the memory of the reserved window word.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
PVOID WinQueryWindowPtr  (HWND hwnd, LONG index)
```

Parameters

hwnd (HWND) – input
Window handle which has the pointer to retrieve.

index (LONG) – input
Zero-based index of the pointer value to retrieve.

Returns

pRet (PVOID) – returns
Pointer value.

NULL	Error occurred.
Other	Pointer value.

WinQueryWindowRect

This function returns a window rectangle.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
```

BOOL WinQueryWindowRect (HWND hwnd, PRECTL prclDest)

Parameters

hwnd (HWND) – input
Window handle whose rectangle is retrieved.

prclDest (PRECTL) – output
Window rectangle.

Returns

rc (BOOL) – returns
Rectangle-returned indicator.

TRUE	Rectangle successfully returned
FALSE	Rectangle not successfully returned.

WinQueryWindowULong

This function obtains the unsigned long integer value, at a specified offset, from the memory of a reserved window word, of a given window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
ULONG WinQueryWindowULong  (HWND hwnd, LONG index)
```

Parameters

hwnd (HWND) – input

Handle of window to be queried.

index (LONG) – input

Zero-based index into the window words of the value to be queried.

QWL_HMQ — Handle of message queue of window. Note that the leading 16 bits of this value are zero.

QWL_STYLE — Window style.

QWL_HWNDFOCUSSAVE — Window handle of the child windows of this window that last possessed the focus when this frame window was last deactivated.

QWL_USER — A ULONG value for applications to use is present at offset QWL_USER in windows of the following preregistered window classes:

WC_FRAME (includes dialog windows)
WC_COMBOBOX
WC_BUTTON
WC_MENU
WC_STATIC
WC_ENTRYFIELD
WC_LISTBOX
WC_SCROLLBAR
WC_TITTLEBAR
WC_MLE
WC_SPINBUTTON
WC_CONTAINER
WC_SLIDER
WC_VALUESET
WC_NOTEBOOK

This value can be used to place application-specific data in controls.

QWL_DEFBUTTON
The default push button for a dialog.

The default push button is the one that sends its WM_COMMAND message when the enter key is pressed.

QWL_PENDATA
Reserved for use by operating system extensions. It allows an operating system extension to store data on a per window basis.

Other
Zero-based index.

Returns

ulValue (ULONG) – returns
Value contained in the window word.

WinQueryWindowUShort

This function obtains the unsigned short integer value at a specified offset from the reserved window word's memory of a given window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

USHORT WinQueryWindowUShort  (HWND hwnd, LONG index)
```

Parameters

hwnd (HWND) – input

Handle of window to be queried.

index (LONG) – input

Zero-based index into the window words of the value to be queried.

QWS_ID
Window identity. The value of the *id* parameter of the WinCreateWindow function.

QWS_FLAGS
These indicators apply only to frame or dialog windows, and contain combinations of the following indicators:

FF_ACTIVE
Frame window is displayed in the active state.

FF_DIALOGBOX
Frame window is being used as a dialog box.

FF_DLGDISMISSED
Dialog has been dismissed by the WinDismissDlg function.

FF_FLASHHILITE
Window is currently flashed. This indicator toggles with each flash.

FF_FLASHWINDOW
Frame window is flashing.

FF_OWNERDISABLED
Window's owner is disabled. This indicator is only set if the window and its owner are siblings.

FF_OWNERHIDDEN
Frame window is hidden as a result of its owner being hidden or minimized. This indicator is set only if the window and its owner are siblings.

FF_SELECTED
Frame window is selected.

QWS_RESULT
Dialog-result parameter, as established by the WinDismissDlg function.

QWS_XRESTORE
The x-coordinate of the position to which the window is restored.

See also the QWS_CYRESTORE value.

QWS_YRESTORE
The y-coordinate of the position to which the window is restored.

See also the QWS_CYRESTORE value.

QWS_CXRESTORE
The width to which the window is restored.

See also the QWS_CYRESTORE value.

QWS_CYRESTORE
The height to which the window is restored.

These values are only valid while the window is maximized or minimized (that is, while either the WS_MINIMIZED or WS_MAXIMIZED window style indicators are set). Changing these values with the WinSetWindowUShort call alters the restore size and position.

QWS_XMINIMIZE
The x-coordinate of the position to which the window is minimized. If this value is -1, the window has not been minimized.

See also the QWS_YMINIMIZE value.

QWS_YMINIMIZE
The y-coordinate of the position to which the window is minimized.

When the window is minimized for the first time an arbitrary position is chosen. Changing these values with the WinSetWindowUShort call alters the position of the minimized window, but only when the window is not in a minimized state.

Other
Zero-based index.

Returns

usValue (USHORT) – returns
Value contained in the indicated window word.

WinRequestMutexSem

WinRequestMutexSem requests ownership of a mutex semaphore or waits for a Presentation Manager message.

Syntax

```
#define INCL_WINMESSAGEMGR

#include <os2.h>

APIRET WinRequestMutexSem  (HMTX hmtx, ULONG ulTimeout)
```

Parameters

hmtx (HMTX) – input
The handle of the mutex semaphore to request.

ulTimeout (ULONG) – input
Time-out in milliseconds.

Returns

ulrc (APIRET) – returns
Return Code.

WinSetActiveWindow

This function makes the frame window the active window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinSetActiveWindow  (HWND  hwndDeskTop, HWND  hwnd)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Active-window-set indicator.

TRUE	Active window is set
FALSE	Active window is not set.

WinSetFocus

This function sets the focus window.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinSetFocus  (HWND  hwndDeskTop, HWND  hwndNewFocus)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

hwndNewFocus (HWND) – input
Window handle to receive the focus.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetMultWindowPos

This function performs the WinSetWindowPos function for *cswp* windows, using *pswp*, an array of structures whose elements correspond to the input parameters of WinSetWindowPos.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetMultWindowPos  (HAB hab, PSWP pswp, ULONG cswp)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pswp (PSWP) – input
An array of set window position (SWP) structures.

cswp (ULONG) – input
Window count.

Returns

rc (BOOL) – returns
Positioning success indicator.

TRUE	Positioning succeeded
FALSE	Positioning failed.

WinSetOwner

This function changes the owner window of a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetOwner  (HWND hwnd, HWND hwndNewOwner)
```

Parameters

hwnd (HWND) – input
Window handle whose owner window is to be changed.

hwndNewOwner (HWND) – input
Handle of the new owner.

NULLHANDLE	The window becomes "disowned"
Other	Handle of the new owner window.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetParent

This function sets the parent for *hwnd* to *hwndNewParent*.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetParent  (HWND  hwnd, HWND  hwndNewParent, BOOL  fRedraw)
```

Parameters

hwnd (HWND) – input
Window handle.

hwndNewParent (HWND) – input
New parent window handle.

fRedraw (BOOL) – input
Redraw indicator.

TRUE If *hwnd* is visible, any necessary redrawing of both the old parent and the new parent windows is performed.

FALSE No redrawing of the old and new parent windows is performed. This avoids an extra device update when subsequent calls cause the windows to be redrawn.

Returns

rc (BOOL) – returns
Parent-changed indicator.

TRUE	Parent successfully changed
FALSE	Parent not successfully changed.

WinSetSysModalWindow

This function makes a window become the system-modal window, or ends the system-modal state.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetSysModalWindow  (HWND hwndDesktop, HWND hwnd)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle, or HWND_DESKTOP.

hwnd (HWND) – input
Handle of window to become system-modal window.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetWindowBits

This function sets a number of bits into the memory of the reserved window words.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetWindowBits  (HWND hwnd, LONG index, ULONG flData,
                        ULONG flMask)
```

Parameters

hwnd (HWND) – input
Window handle.

index (LONG) – input
Zero-based index of the value to be set.

QWL_HMQ — Handle of message queue of window. Note that the leading 16 bits of this value are zero.

QWL_STYLE — Window style.

QWL_HWNDFOCUSSAVE — Window handle of the child windows of this window that last possessed the focus when this frame window was last deactivated.

QWL_USER — A ULONG value for applications to use is present at offset QWL_USER in windows of the following preregistered window classes:

WC_FRAME (includes dialog windows)
WC_COMBOBOX
WC_BUTTON
WC_MENU
WC_STATIC
WC_ENTRYFIELD
WC_LISTBOX
WC_SCROLLBAR
WC_TITTLEBAR
WC_MLE
WC_SPINBUTTON
WC_CONTAINER
WC_SLIDER
WC_VALUESET
WC_NOTEBOOK

This value can be used to place application-specific data in controls.

QWL_DEFBUTTON: The default push button for a dialog.

The default push button is the one that sends its WM_COMMAND message when the enter key is pressed.

QWL_PENDATA: Reserved for use by operating system extensions. It allows an operating system extension to store data on a per window basis.

Other: Zero-based index.

flData (ULONG) – input
Bit data to store in the window words.

flMask (ULONG) – input
Bits to be written indicator.

Returns

rc (BOOL) – returns
Success indicator.

TRUE Successful completion
FALSE Error occurred.

WinSetWindowPos

This function allows the general positioning of a window.

Note: Messages may be received from other processes or threads during the processing of this function.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetWindowPos  (HWND hwnd, HWND hwndInsertBehind, LONG x,
                       LONG y, LONG cx, LONG cy, ULONG fl)
```

Parameters

hwnd (HWND) – input
Window handle.

hwndInsertBehind (HWND) – input
Relative window-placement order.

HWND_TOP	Place *hwnd* on top of all siblings
HWND_BOTTOM	Place *hwnd* behind all siblings
Other	Identifies the sibling window behind which *hwnd* is to be placed.

x (LONG) – input
Window position, x-coordinate.

y (LONG) – input
Window position, y-coordinate.

cx (LONG) – input
Window size.

cy (LONG) – input
Window size.

fl (ULONG) – input
Window-positioning options.

SWP_SIZE	Change the window size.
SWP_MOVE	Change the window x,y position.
SWP_ZORDER	Change the relative window placement.
SWP_SHOW	Show the window.
SWP_HIDE	Hide the window.
SWP_NOREDRAW	Changes are not redrawn.

SWP_NOADJUST Do not send a WM_ADJUSTWINDOWPOS message before moving or sizing.

SWP_ACTIVATE Activate the *hwnd* window if it is a frame window. This indicator has no effect on other windows.

The frame window is made the topmost window, unless SWP_ZORDER is specified also in which instance the *hwndInsertBehind* window is used.

SWP_DEACTIVATE Deactivate the *hwnd* window if it is a frame window. This indicator has no effect on other windows.

The frame window is made the bottommost window, unless SWP_ZORDER is specified, in which instance the *hwndInsertBehind* window is used.

SWP_MINIMIZE Minimize the window. This indicator has no effect if the window is in a minimized state, and is also mutually exclusive with SWP_MAXIMIZE and SWP_RESTORE.

SWP_MAXIMIZE Maximize the window. This indicator has no effect if the window is in a maximized state, and is also mutually exclusive with SWP_MINIMIZE and SWP_RESTORE.

SWP_RESTORE Restore the window. This indicator has no effect if the window is in its normal state, and is also mutually exclusive with SWP_MINIMIZE and SWP_MAXIMIZE.

The position and size of the window in its normal state is remembered in its window words when it is first maximized or minimized, although these values can be altered by use of the WinSetWindowUShort function.

The window is restored to the position and size remembered in its window words, unless the SWP_MOVE or SWP_SIZE indicators are set. These indicators cause the position and size values specified in this function to be used.

Returns

rc (BOOL) – returns

Repositioning indicator.

TRUE	Window successfully repositioned
FALSE	Window not successfully repositioned.

WinSetWindowPtr

This function sets a pointer value into the memory of the reserved window words.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetWindowPtr  (HWND hwnd, LONG lb, PVOID pp)
```

Parameters

hwnd (HWND) – input
Window handle.

lb (LONG) – input
Zero-based index into the window words.

pp (PVOID) – input
Pointer value to store in the window words.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetWindowULong

This function sets an unsigned, long integer value into the memory of the reserved window words.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetWindowULong  (HWND hwnd, LONG index, ULONG ul)
```

Parameters

hwnd (HWND) – input
Window handle.

index (LONG) – input
Zero-based index of the value to be set.

QWL_HMQ
Handle of message queue of window. Note that the leading 16 bits of this value are zero.

QWL_STYLE
Window style.

QWL_HHEAP
Heap handle used by child windows of this window.

QWL_HWNDFOCUSSAVE
Window handle of the child windows of this window that last possessed the focus when this frame window was last deactivated.

QWL_USER
A ULONG value for applications to use is present at offset QWL_USER in windows of the following preregistered window classes:

WC_FRAME (includes dialog windows)
WC_LISTBOX
WC_BUTTON
WC_STATIC
WC_ENTRYFIELD
WC_SCROLLBAR
WC_MENU

This value can be used to place application-specific data in controls.

QWL_DEFBUTTON
The default push button for a dialog.

The default push button is the one that sends its WM_COMMAND message when the enter key is pressed.

QWL_PENDATA	Reserved for use by operating system extensions. It allows an operating system extension to store data on a per window basis.
Other	Zero-based index.

ul (ULONG) – input
Unsigned, long integer value to store in the window words.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetWindowUShort

This function sets an unsigned, short integer value into the memory of the reserved window words.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetWindowUShort  (HWND hwnd, LONG index, USHORT us)
```

Parameters

hwnd (HWND) – input
Window handle.

index (LONG) – input
Zero-based index of the value to be set.

QWL_HMQ
Handle of message queue of window. Note that the leading 16 bits of this value are zero.

QWL_STYLE
Window style.

QWL_HWNDFOCUSSAVE
Window handle of the child windows of this window that last possessed the focus when this frame window was last deactivated.

QWL_USER
A ULONG value for applications to use is present at offset QWL_USER in windows of the following preregistered window classes:

WC_FRAME (includes dialog windows)
WC_COMBOBOX
WC_BUTTON
WC_MENU
WC_STATIC
WC_ENTRYFIELD
WC_LISTBOX
WC_SCROLLBAR
WC_TITTLEBAR
WC_MLE
WC_SPINBUTTON
WC_CONTAINER
WC_SLIDER
WC_VALUESET
WC_NOTEBOOK

This value can be used to place application-specific data in controls.

QWL_DEFBUTTON
The default push button for a dialog.

The default push button is the one that sends its WM_COMMAND message when the enter key is pressed.

QWL_PENDATA
Reserved for use by operating system extensions. It allows an operating system extension to store data on a per window basis.

Other
Zero-based index.

us (USHORT) – input
Unsigned, short integer value to store in the window words.

Returns

rc (BOOL) – returns
Success indicator.

TRUE Successful completion
FALSE Error occurred.

WinShowWindow

This function sets the visibility state of a window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinShowWindow  (HWND hwnd, BOOL fNewVisibility)
```

Parameters

hwnd (HWND) – input
Window handle.

fNewVisibility (BOOL) – input
New visibility state.

TRUE	Set window state visible
FALSE	Set window state invisible.

Returns

rc (BOOL) – returns
Visibility changed indicator.

TRUE	Window visibility successfully changed
FALSE	Window visibility not successfully changed.

WinStartApp

This function starts an application.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HAPP WinStartApp  (HWND hwndNotify, PPROGDETAILS pDetails, PSZ pParams,
                   PVOID pReserved, ULONG ulOptions)
```

Parameters

hwndNotify (HWND) – input
Notification-window handle.

NULLHANDLE	Do not post the notification message
Other	Post the notification message to this window.

pDetails (PPROGDETAILS) – input
Program list structure.

pParams (PSZ) – input
Input parameters for the application to be started.

NULL	There are no parameters to be passed to the application
Other	The parameters to be passed to the application.

pReserved (PVOID) – input
Start data.

ulOptions (ULONG) – input
Option indicators.

0	No options selected.
SAF_INSTALLEDCMDLINE	The command line parameters installed in the program starter list are used; the *pParams* parameter is ignored.
SAF_STARTCHILDAPP	The specified application is started as a child session of the session from which WinStartApp is issued. The calling application may terminate the called application with a WinTerminateApp function.

Returns

happ (HAPP) – returns
Application handle.

NULL	Application not started
Other	Application handle.

WinTerminate

This function terminates an application thread's use of the Presentation Manager and releases all of its associated resources.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinTerminate  (HAB hab)
```

Parameters

hab (HAB) – input
Anchor-block handle.

Returns

rc (BOOL) – returns
Termination indicator.

TRUE	Application usage of Presentation Manager successfully terminated
FALSE	Application usage of Presentation Manager not successfully terminated, or WinInitialize has not been issued on this thread.

WinTerminateApp

This function terminates an application previously started with the WinStartApp function.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinTerminateApp  (HAPP  happ)
```

Parameters

happ (HAPP) – input
Anchor-block handle.

Returns

rc (BOOL) – returns
Termination indicator.

TRUE	Application successfully terminated
NULL	Error occurred.

WinWaitEventSem

WinWaitEventSem waits for an event semaphore to be posted or for a Presentation Manager message.

Syntax

```
#define INCL_WINMESSAGEMGR

#include <os2.h>

APIRET WinWaitEventSem  (HEV hev, ULONG ulTimeout)
```

Parameters

hev (HEV) – input
The handle of the event semaphore to wait for.

ulTimeout (ULONG) – input
Time-out in milliseconds.

Returns

rc (APIRET) – returns
Return Code.

WinWaitMuxWaitSem

WinWaitMuxWaitSem waits for a muxwait semaphore to clear or for a Presentation Manager message.

Syntax

```
#define INCL_WINMESSAGEMGR

#include <os2.h>

APIRET WinWaitMuxWaitSem  (HMUX hmux, ULONG ulTimeout,
                           PULONG pulUser)
```

Parameters

hmux (HMUX) – input

The handle of the muxwait semaphore to wait for.

ulTimeout (ULONG) – input

Time-out in milliseconds.

SEM_IMMEDIATE_RETURN (0) WinWaitMuxWaitSem returns without blocking the calling thread.

SEM_INDEFINITE_WAIT (minus.1) WinWaitMuxWaitSem blocks the calling thread indefinitely.

pulUser (PULONG) – output

Pointer to receive the user field.

Returns

ulrc (APIRET) – returns

Return Code.

WinWindowFromID

This function returns the handle of the child window with the specified identity.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinWindowFromID  (HWND  hwndParent, ULONG  id)
```

Parameters

hwndParent (HWND) – input
Parent-window handle.

id (ULONG) – input
Identity of the child window.

Returns

hwnd (HWND) – returns
Window handle.

NULLHANDLE	No child window of the specified identity exists
Other	Child-window handle.

WinWindowFromPoint

This function finds the window below a specified point, that is a descendant of a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinWindowFromPoint  (HWND hwndParent, PPOINTL pptlPoint,
                          BOOL fEnumChildren)
```

Parameters

hwndParent (HWND) – input

Window handle whose child windows are to be tested.

HWND_DESKTOP	The desktop-window handle, implying that all main windows are tested. In this instance, *pptlPoint* must be relative to the bottom left corner of the screen.
Other	Parent-window handle.

pptlPoint (PPOINTL) – input

The point to be tested.

fEnumChildren (BOOL) – input

Test control.

TRUE	Test all the descendant windows, including child windows of child windows
FALSE	Test only the immediate child windows.

Returns

hwndFound (HWND) – returns

Window handle beneath *pptlPoint*.

NULLHANDLE	*pptlPoint* is outside *hwndParent*
Parent	*pptlPoint* is not inside any of the children of *hwndParent*
Other	Window handle is beneath *pptlPoint*.

Related Messages

This section covers the messages that are related to Windows.

WM_ACTIVATE

This message occurs when an application causes the activation or deactivation of a window.

Parameters

param1

usactive (USHORT)
Active indicator.

TRUE	The window is being activated
FALSE	The window is being deactivated.

param2

hwnd (HWND)
Window handle.

In the case of activation, *hwnd* identifies the window being activated. In the case of deactivation, *hwnd* identifies the window being deactivated.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_ADJUSTWINDOWPOS

This message is sent by the WinSetWindowPos call to enable the window to adjust its new position or size whenever it is about to be moved.

Parameters

param1

pswp (PSWP)
SWP structure pointer.

The structure has been filled in by the WinSetWindowPos function with the proposed move or size data. The control can adjust this new position by changing the contents of the SWP structure. It can change the *x* or *y* fields to adjust its new position; or the *cx* or *cy* fields to adjust its new size, or the *hwndInsertBehind* field to adjust its new z-order.

param2

flzero (ULONG)
Zero.

Returns

flResult (ULONG)
Window-adjustment status indicators.

0	No changes have been made
AWP_MINIMIZED	The frame window has been minimized.
AWP_MAXIMIZED	The frame window has been maximized.
AWP_RESTORED	The frame window has been restored.
AWP_ACTIVATE	The frame window has been activated.
AWP_DEACTIVATE	The frame window has been deactivated.

WM_CALCFRAMERECT

This message occurs when an application uses the WinCalcFrameRect function.

Parameters

param1

pRect (PRECTL)
Rectangle structure.

This points to a RECTL structure.

param2

usFrame (USHORT)
Frame indicator.

TRUE	Frame rectangle provided
FALSE	Client area rectangle provided.

Returns

rc (BOOL)
Rectangle-calculated indicator.

TRUE	Successful completion
FALSE	Error occurred or the calculated rectangle is empty.

WM_CALCVALIDRECTS

This message is sent from WinSetWindowPos and WinSetMultWindowPos to determine which areas of a window can be preserved if a window is sized, and which should be redisplayed.

Parameters

param1

pOldNew (PRECTL)
Window-rectangle structures.

This points to two RECTL structures. The first structure contains the rectangle of the window before the move, the second contains the rectangle of the window after the move. The coordinates of the rectangles are relative to the parent window.

param2

pNew (PSWP)
New window position.

This points to a SWP structure that contains information about the window after it is resized (see the WinSetWindowPos function).

Returns

usAlign (USHORT)
Alignment control.

CVR_ALIGNLEFT	Align with the left edge of the window.
CVR_ALIGNBOTTOM	Align with the bottom edge of the window.
CVR_ALIGNTOP	Align with the top edge of the window.
CVR_ALIGNRIGHT	Align with the right edge of the window.
CVR_REDRAW	The whole window is invalid. If CVR_REDRAW, is set, the whole window is assumed invalid, otherwise, the remaining flags can be ORed together to get different kinds of alignment. For example: `(CVR_ALIGNLEFT | CVR_ALIGNTOP)` aligns the valid window area with the top-left of the window.
0	It is assumed the application has changed the rectangles pointed to by *pOldNew* and *pNew* itself.

WM_CLOSE

This message is sent to a frame window to indicate that the window is being closed by the user.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_CREATE

This message occurs when an application requests the creation of a window.

Parameters

param1

ctldata (PVOID)

Pointer to control data.

This points to a Control-Data data structure initialized with the data provided in the *pCtlData* parameter of the WinCreateWindow function. This pointer is also contained in the *pCREATE* parameter.

This parameter MUST be a pointer rather than a long.

The first 2 bytes in the data referenced by this pointer should be the total size of the data referenced by the pointer, (for example, see the ENTRYFDATA or the FRAMECDATA structure). PM requires this information to enable it to ensure that the referenced data is accessible to both 16-bit and 32-bit code.

param2

pCREATE (PCREATESTRUCT)

Create structure.

This points to a CREATESTRUCT data structure. See the description of *ctldata* for a complete description.

Returns

rc (BOOL)

Error indicator.

TRUE	Discontinue window creation
FALSE	Continue window creation.

WM_DESTROY

This message occurs when an application requests the destruction of a window.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_ENABLE

This message notifies a windows of a change to its enable state.

Parameters

param1

usnewenabledstate (USHORT)

New enabled state indicator.

TRUE	The window was set to the enabled state.
FALSE	The window was set to the disabled state.

param2

ulReserved (ULONG)

Reserved value, should be 0.

Returns

ulReserved (ULONG)

Reserved value, should be 0.

WM_MOVE

This message occurs when a window with style CS_MOVENOTIFY changes its absolute position.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_QUERYWINDOWPARAMS

This message occurs when an application queries the window parameters.

Parameters

param1

pwndparams (PWNDPARAMS)
Window parameter structure.

This points to a window parameter structure; see "WNDPARAMS" on page 2-108.

The valid values of *fsStatus* are WPM_CCHTEXT, WPM_TEXT, WPM_CBCTLDATA, and WPM_CTLDATA.

The flags in *fsStatus* are cleared as each item is processed. If the call is successful, *fsStatus* is 0. If any item has not been processed, the flag for that item is still set.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_SETWINDOWPARAMS

This message occurs when an application sets or changes the window parameters.

Parameters

param1

pwndparams (PWNDPARAMS)
Window parameter structure.

This points to a window parameter structure; see "WNDPARAMS" on page 2-108.

The valid values of *fsStatus* are WPM_TEXT and WPM_CTLDATA.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful operation
FALSE	Error occurred.

WM_SHOW

This message occurs when the WS_VISIBLE state of a window is being changed.

Parameters

param1

usshow (USHORT)
Show indicator.

TRUE	Show the window
FALSE	Hide the window.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SIZE

This message occurs when a window changes its size.

Parameters

param1

scxold (SHORT)
Old horizontal size.

scyold (SHORT)
Old vertical size.

param2

scxnew (SHORT)
New horizontal size.

scynew (SHORT)
New vertical size.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SYSCOMMAND

This message occurs when a control has a significant event to report to its owner or when a key stroke has been translated by an accelerator table.

Parameters

param1

uscmd (USHORT)
Command value.

The command value can be one of the SC_* values. It is the responsibility of the application to be able to relate *uscmd* to an application function.

param2

ussource (USHORT)
Source type.

Identifies the type of control:

CMDSRC_PUSHBUTTON	Posted by a push-button control. *uscmd* is the window identifier of the push button.
CMDSRC_MENU	Posted by a menu control. *uscmd* is the identifier of the menu item.
CMDSRC_ACCELERATOR	Posted as the result of an accelerator. *uscmd* is the accelerator command value.
CMDSRC_OTHER	Other source. *uscmd* gives further control-specific information defined for each control type.

uspointer (USHORT)
Pointing-device indicator.

TRUE	The message is posted as a result of a pointing-device operation.
FALSE	The message is posted as a result of a keyboard operation.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_WINDOWPOSCHANGED

If this message has any of the values of the *fl* parameter of the SWP structure set, with the exception of the SWP_NOADJUST and SWP_NOREDRAW values, it is sent to the window procedure of the window whose position is changed.

This message is also sent if the return value from the WM_ADJUSTWINDOWPOS is not NULL.

Parameters

param1

pswp (PSWP)
SWP structures.

This points to two SWP structures. The first SWP structure describes the entire new window state, whereas the second structure describes the entire old window state. The *fl* parameter of the first structure contains only those indicators corresponding to the state changes that occurred.

param2

flAwp (ULONG)
Adjust window position status indicators.

The AWF_* flags specify the state change of the frame window.

The return value from the WM_ADJUSTWINDOWPOS message:

0	The SWP_NOADJUST option has been specified.
Other	Adjust window position status indicators.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Windows.

CREATESTRUCT

Create-window data structure.

Syntax

```
typedef struct _CREATESTRUCT {
PVOID       pPresParams;
PVOID       pCtlData;
ULONG       id;
HWND        hwndInsertBehind;
HWND        hwndOwner;
LONG        cy;
LONG        cx;
LONG        y;
LONG        x;
ULONG       flStyle;
PSZ         pszText;
PSZ         pszClassName;
HWND        hwndParent;
 } CREATESTRUCT;

typedef CREATESTRUCT *PCREATESTRUCT;
```

Fields

pPresParams (PVOID)
Presentation parameters.

pCtlData (PVOID)
Control data.

id (ULONG)
Window identifier.

hwndInsertBehind (HWND)
Window behind which the window is to be placed.

hwndOwner (HWND)
Window owner.

cy (LONG)
Window height.

cx (LONG)
Window width.

y (LONG)
Y-coordinate of origin.

x (LONG)
X-coordinate of origin.

flStyle (ULONG)
Window style.

pszText (PSZ)
Window text.

pszClassName (PSZ)
Registered window class name.

hwndParent (HWND)
Parent window handle.

WNDPARAMS

Window parameters.

Syntax

```
typedef struct _WNDPARAMS {
ULONG       fsStatus;
ULONG       cchText;
PSZ         pszText;
ULONG       cbPresParams;
PVOID       pPresParams;
ULONG       cbCtlData;
PVOID       pCtlData;
 } WNDPARAMS;

typedef WNDPARAMS *PWNDPARAMS;
```

Fields

fsStatus (ULONG)

Window parameter selection.

Identifies the window parameters that are to be set or queried:

WPM_CBCTLDATA	Window control data length
WPM_CCHTEXT	Window text length
WPM_CTLDATA	Window control data
WPM_PRESPARAMS	Presentation parameters
WPM_TEXT	Window text.

cchText (ULONG)

Length of window text.

pszText (PSZ)

Window text.

cbPresParams (ULONG)

Length of presentation parameters.

pPresParams (PVOID)

Presentation parameters.

cbCtlData (ULONG)

Length of window class specific data.

pCtlData (PVOID)

Window class specific data.

Summary

Following are the OS/2 functions, messages, and data structures used with windows.

Table 2-5 (Page 1 of 3). Window Functions

Window Creation Functions	
WinCreateWindow	The most direct way of creating a window. The window is of class *ClassName* and returns *hwnd*.
WinCreateStdWindow	Creates a main window. Requires an anchor block.
WinInitialize	Initializes the PM programming interface facility.
Window Destruction Functions	
WinDestroyWindow	Destroys a window and its child windows, and releases all their resources.
Window Data Functions	
WinQueryWindowUShort	Obtains the unsigned short integer value of a given window at a specified offset from the reserved window word's memory.
WinSetWindowUShort	Sets an unsigned, short integer value into the memory of the reserved window words.
WinQueryWindowULong	Obtains the unsigned long integer value of a given window, at a specified offset, from the memory of a reserved window word.
WinSetWindowULong	Sets an unsigned, long integer value into the memory of the reserved window words.
WinQueryWindowPtr	Retrieves a pointer value from the memory of the reserved window word.
WinSetWindowPtr	Sets a pointer value into the memory of the reserved window words.
WinSetWindowBits	Sets a number of bits into the memory of the reserved window words.
Window Relationship Functions	
WinIsWindow	Determines if a window handle is valid.
WinIsThreadActive	Determines whether the active window belongs to the calling execution thread.
WinSetParent	Sets the parent for *hwnd* to NewParent.
WinQueryWindow	Returns the handle of a window that has a specified relationship to a specified window.
WinSetOwner	Changes the owner of a specified window.
WinBeginEnumWindows	Begins the enumeration process for all the immediate child windows of a specified window.

Table 2-5 (Page 2 of 3). Window Functions

WinGetNextWindow	Gets the window handle of the next window in a specified enumeration list.
WinEndEnumWindows	Ends the specified enumeration process.
WinIsChild	Tests to determine whether one window is a descendant of another.
WinQueryDesktopWindow	Returns the desktop window handle.
WinQueryObjectWindow	Returns the desktop-object window handle.
WinWindowFromID	Returns the handle of the child window with the specified ID.
WinWindowFromPoint	Finds the window, below a specified point, that is a descendant of a specified window.
WinMultWindowFromIDs	Finds the handles of child windows that belong to a specified window and that have window IDs within a specified range.
Window Size and Position Functions	
WinSetWindowPos	Facilitates the general positioning of a window.
WinQueryWindowPos	Obtains the size and position of a window.
WinSetMultWindowPos	An efficient means of repositioning multiple windows with one call, provided all windows being positioned have the same parent.
WinQueryWindowRect	Returns a window rectangle.
WinGetMaxPosition	Fills an SWP structure with the maximized-window size and position.
WinGetMinPosition	Returns the position to which a window is minimized.
Window Visibility Functions	
WinIsWindowShowing	Determines whether any part of the window, *hwnd*, is physically visible.
WinShowWindow	Sets the visibility state of a window.
WinIsWindowVisible	Returns the visibility state of a window.
Window Input Functions	
WinEnableWindow	Sets the window enabled state.
WinIsWindowEnabled	Returns the enabled or disabled state of a window.
WinQueryActiveWindow	Returns the active window for HWND_DESKTOP or other parent window.
WinSetActiveWindow	Sets the main window as the active window.
WinQueryFocus	Returns the focus window; NULL if there is no focus window.
WinSetFocus	Sets the focus window.

Table 2-5 (Page 3 of 3). Window Functions

WinQuerySysModalWindow	Returns the current system-modal window.
WinRequestMutexSem	Requests the ownership of a mutex semaphore or waits for a PM message.
WinSetSysModalWindow	Either sets a system-modal window or ends the system-modal state.
WinStartApp	Starts an application.
WinTerminate	Terminates an application thread's use of PM and releases all of its associated resources.
WinTerminateApp	Terminates an application started with WinStartApp.
WinWaitEventSem	Waits for an event semaphore to be posted or for a PM message.
WinWaitMuxWaitSem	Waits for a muxwait semaphore to clear or for a PM message.

Table 2-6. Window Messages

Message	Description
WM_ACTIVATE	Sent to a window as it gains or loses activation.
WM_ADJUSTWINDOWPOS	Sent to adjust a window's position. Not sent if SWP_NOADJUST is specified.
WM_CALCFRAMERECT	Occurs when an application uses the WinCalcFrameRect call.
WM_CALCVALIDRECTS	Sent from WinSetWindowPos and WinSetMultWindowPos to determine which areas of a window will be preserved if a window is sized and which should be redisplayed.
WM_CLOSE	Sent to a frame window to indicate that the window is being closed by the user.
WM_CREATE	Occurs when the application requests creation of a window.
WM_DESTROY	Occurs when the application requests destruction of a window.
WM_ENABLE	Sets the enable state of a window.
WM_MOVE	Occurs when a window with the style CS_MOVENOTIFY changes its absolute position.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_SETFOCUS	Occurs when a window is to receive or lose the input focus.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the window parameters.
WM_SHOW	Occurs when a window's WS_VISIBLE state is being changed.
WM_SIZE	Occurs when a window changes its size.
WM_SYSCOMMAND	Occurs when a control has a significant event to report to its owner.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Table 2-7. Window Data Structures

Data Structure	Description
CREATESTRUCT	Create window.
WNDPARAMS	Window parameters.

Chapter 3. Messages and Message Queues

The OS/2 operating system uses messages and message queues to communicate with applications and the windows belonging to those applications. This chapter explains how to create and use messages and message queues in PM applications.

About Messages and Message Queues

Unlike traditional applications that take complete control of the computer's keyboard, mouse, and screen, PM applications must share these resources with other applications that are running at the same time. All applications run independently and rely on the operating system to help them manage shared resources. The operating system does this by controlling the operation of each application, communicating with each application when there is keyboard or mouse input or when an application must move and size its windows.

Messages

A *message* is information, a request for information, or a request for an action to be carried out by a window in an application.

The operating system, or an application, sends or posts a message to a window so that the window can use the information or respond to the request.

There are three types of messages:

- User-initiated
- Application-initiated
- System-initiated.

A user-initiated message is the direct result of a user action, such as selecting a menu item or pressing a key. An application-initiated message is generated by one window in the application to communicate with another window. System-initiated messages are generated by the interface as the indirect result of a user action (for example, resizing a window) or as the direct result of a system event (such as creating a window).

A message that requires an immediate response from a window is sent directly to the window by passing the message data as arguments to the window procedure. The window procedure carries out the request or lets the operating system carry out default processing for the message.

A message that does not require an immediate response from a window is *posted* (the message data is copied) to the application's *message queue*. The message queue is a storage area that the application creates to receive and hold its posted messages. Then, the application can retrieve a message at the appropriate time, sending it to the addressed window for processing.

Every message contains a *message identifier*, which is a 16-bit integer that indicates the purpose of the message. When a window processes a message, it uses the message identifier to determine what to do.

Every message contains a *window handle*, which identifies the window the message is for. The window handle is important because most message queues and window procedures serve more than one window. The window handle ensures that the application forwards the message to the proper window.

A message contains two *message parameters*—32-bit values that specify data or the location of data that a window uses when processing the message. The meaning and value of a message parameter depend on the message. A message parameter can contain an integer, packed bit flags, a pointer to a structure that contains additional data, and so forth. Some messages do not use message parameters and, typically, set the parameters to NULL. An application always checks the message identifier to determine how to interpret the message parameters.

A *queue message* is a QMSG data structure that contains six data items, representing the window handle, message identifier, two message parameters, message time, and mouse-pointer position. The time and position are included because most queue messages are input messages, representing keyboard or mouse input from the user. The time and position also help the application identify the context of the message. The operating system posts a queue message by filling the QMSG structure and copying it to a message queue.

A *window message* consists of the window handle, the message identifier, and two message parameters. A window message does not include the message time and mouse-pointer position, because most window messages are requests to perform a task that is not related to the current time or mouse-pointer position. The operating system sends a window message by passing these values, as individual arguments, to a window procedure.

Message Queues

Every PM application must have a *message queue*. A message queue is the only means an application has to receive input from the keyboard or mouse. *Only applications that create message queues can create windows.*

An application creates a message queue by using the WinCreateMsgQueue function. This function returns a handle that the application can use to access the message queue. After an application creates a message queue, the system posts messages intended for windows in the application to that queue. The application can retrieve queue messages by specifying the message-queue handle in the WinGetMsg function. It also can examine messages, without retrieving them, by using the WinPeekMsg function. When an application no longer needs the message queue, it can destroy the queue by using the WinDestroyMsgQueue function.

One message queue serves all the windows in a thread. This means a queue can hold messages for several windows. A message specifies the handle of the window to which it belongs so the application can forward a message easily to the appropriate window. The message loop recognizes a NULL window handle and the message is processed within the message loop rather than passed to WinDispatchMessage. See Figure 3-1 on page 3-5 for an example of an input-message processing loop.

An application that has more than one thread can create more than one message queue. The system allows one message queue for each thread. A message queue created by a thread belongs to that thread and has no connection to other queues in the application. When an application creates a window in a given thread, the system associates the window with the message queue in that thread. The system then posts all subsequent messages intended for that window to that queue.

Note: The recommended way to structure PM applications is to have at least two threads and two message queues. The first thread and message queue control all the user-interface windows, and the second thread and message queue control all the object windows.

Several windows can use one message queue; it is important that the message queue be large enough to hold all messages that possibly can be posted to it. An application can set the size of the message queue when it creates the queue by specifying the maximum number of messages the queue can hold. The default maximum number of messages is 10.

To minimize queue size, several types of posted messages are not actually stored in a message queue. Instead, the operating system keeps a record in the queue of the message being posted and combines any information contained in the message with information from previous messages. Timer, semaphore, and paint messages are handled this way. For example, if more than one WM_PAINT message is posted, the operating system combines the *update regions* for each into a single update region. Although there is no actual WM_PAINT message in the queue, the operating system constructs one WM_PAINT message with the single update region when an application uses the WinGetMsg function.

The operating system handles mouse and keyboard input messages differently from the way it handles other types of messages. The operating system receives all keyboard and mouse events, such as keystrokes and mouse movements, into the system message queue. The operating system converts these events into messages and posts them, one at a time, to the appropriate application message queue. The application retrieves the messages from its queue and dispatches them to the appropriate window, which processes the messages.

The operating system message queue usually is large enough to hold all input messages, even if the user types or moves the mouse very quickly. If the operating system message queue does run out of space, the system *ignores* the most recent keyboard input (usually by beeping to indicate the input is ignored) and collects mouse motions into a WM_MOUSEMOVE message.

Every message queue has a corresponding MQINFO data structure that specifies the identifiers of the process and thread that own the message queue and gives a count of the maximum number of messages the queue can receive. An application can retrieve the structure by using the WinQueryQueueInfo function.

A message queue also has a current status that indicates the types of messages currently in the queue. An application can retrieve the queue status by using the WinQueryQueueStatus function. An application also can use the WinPeekMsg function to examine the contents of a message queue. WinPeekMsg checks for a specific message or range of messages in the queue and gives the application the option of removing messages from the queue. An application *can* call the WinQueryQueueStatus function to determine the contents of the queue before calling the WinPeekMsg or WinGetMsg function to remove a message from the queue.

Message Handling

To handle and process messages, an application can use a *message loop* and the *window procedure*. These terms are explained in the following two sections.

Message Loops

Every application with a message queue is responsible for retrieving the messages from that queue. An application can do this by using a message loop, usually in the application's main function, that retrieves messages from the message queue and dispatches them to the appropriate windows. The message loop consists of two calls: one to the WinGetMsg function; the other to the WinDispatchMsg function. The message loop has the following form:

```
HAB hab;
QMSG qmsg;

while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
    WinDispatchMsg(hab, &qmsg);
```

An application starts the message loop after creating the message queue and at least one application window. Once started, the message loop continues to retrieve messages from the message queue and to dispatch (send) them to the appropriate windows. WinDispatchMsg sends each message to the window specified by the window handle in the message.

Figure 3-1 on page 3-5 illustrates the typical routing of an input message through the operating system's and application's message loops.

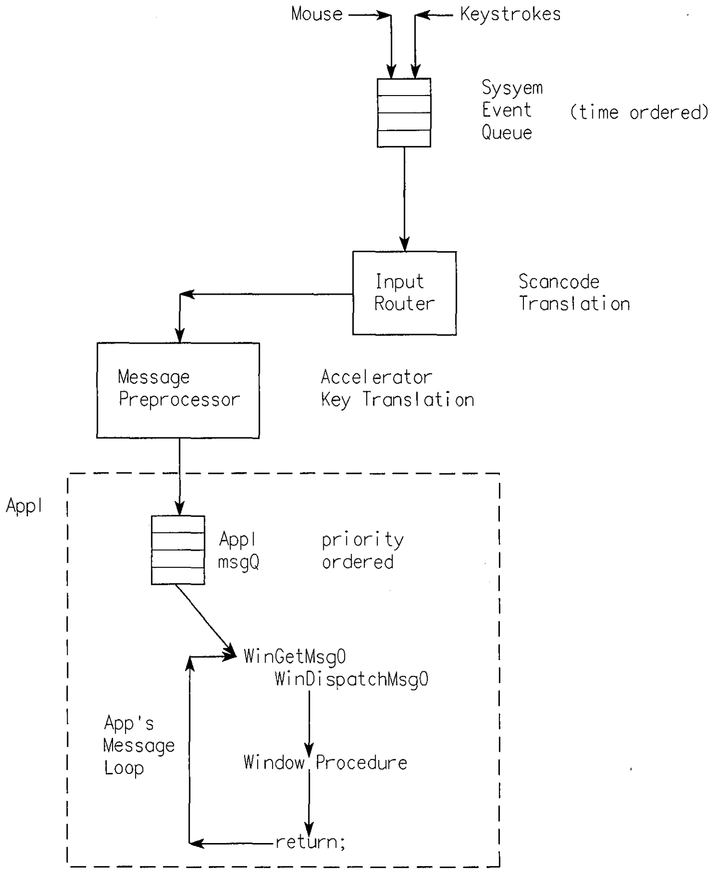


Figure 3-1. Input Message Processing Loop

Only one message loop is needed for a message queue, even if the queue contains messages for more than one window. Each queue message is a QMSG structure that contains the handle of the window to which the message belongs. WinDispatchMsg always dispatches the message to the proper window. WinGetMsg retrieves messages from the queue in first-in, first-out (FIFO) order, so the messages are dispatched to windows in the same order they are received.

If there are no messages in the queue, the operating system temporarily stops processing the WinGetMsg function until a message arrives. This means that processor time that, otherwise, would be spent waiting for a message can be given to the applications (or threads) that do have messages in their queues.

The message loop continues to retrieve and dispatch messages until WinGetMsg retrieves a WM_QUIT message. This message causes the function to return FALSE, terminating the loop. In most cases, terminating the message loop is the first step in terminating the application. An application can terminate its own loop by posting the WM_QUIT message in its own queue.

An application can modify its message loop in a variety of ways. For example, it can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages without specifying a window. (These messages apply to the application rather than a specific window; they have NULL window handles.) Also, an application can direct the WinGetMsg function to search for specific messages, leaving other messages in the queue. This is useful for applications that temporarily need to bypass the usual FIFO order of the message queue.

Window Procedures

A *window procedure* is a function that receives and processes all input and requests for action sent to the windows. Every window class has a window procedure; every window created using that class uses that window procedure to respond to messages.

The system sends a message to the window procedure by passing the message data as arguments. The window procedure takes the appropriate action for the given message. Most window procedures check the message identifier, then use the information specified by the message parameters to carry out the request. When it has completed processing the message, the window procedure returns a message result. Each message has a particular set of possible return values. The window procedure must return the appropriate value for the processing it performed.

A window procedure cannot ignore a message. If it does not process a message, it must pass the message back to the operating system for default processing. The window procedure does this by calling the WinDefWindowProc function to carry out a default action and return the message result. Then, the window procedure must return this value as its own message result.

A window procedure commonly processes messages for several windows. It uses the window handle specified in the message to identify the appropriate window. Most window procedures process just a few types of messages and pass the others on to the operating system by calling WinDefWindowProc.

Posting and Sending Messages

Any application can post and send messages. Like the operating system, an application *posts* a message by copying it to a message queue. It *sends* a message by passing the message data as arguments to a window procedure. To post and send messages, an application uses the WinPostMsg and WinSendMsg functions.

An application posts a message to notify a specific window to perform a task. The WinPostMsg function creates a QMSG structure for the message and copies the message to the message queue corresponding to the given window. The application's message loop eventually retrieves the message and dispatches it to the appropriate window procedure. For example, one message commonly posted is WM_QUIT. This message terminates the application by terminating the message loop.

An application sends a message to cause a specific window procedure to carry out a task immediately. The WinSendMsg function passes the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an entry-field control as its child window can set the text of the control by sending a message to the child window. The control can notify the parent window of changes to the text (carried out by the user) by sending messages back to the parent window.

Occasionally, an application might need to send or post a message to all windows in the system. For example, if the application changes a system value, it must notify all windows about the change by sending a WM_SYSVALUECHANGED message. An application can send or post messages to any number of windows by using the WinBroadcastMsg function. The options in WinBroadcastMsg determine whether the message is sent or posted and specify the windows that will receive the message.

Any thread in the application can post a message to a message queue, even if the thread has no message queue of its own. However, only a thread that has a message queue can send a message. Sending a message between threads is relatively uncommon. For one reason, sending a message is costly in terms of system performance. If an application posts a message between threads, it is likely to be a semaphore message, which permits window procedures to manage a shared resource jointly.

An application can post a message without specifying a window. If the application supplies a NULL window handle when it calls the WinPostMsg function, the function posts the message to the queue associated with the current thread. The application must process the message in the message loop. This is one way to create a message that applies to the entire application instead of to a specific window.

A window procedure can determine whether it is processing a message sent by another thread by using the WinInSendMsg function. This is useful when message processing depends on the origin of the message.

A common programming error is to assume that the WinPostMsg function always succeeds. It fails when the message queue is full. An application should check the return value of the WinPostMsg function to see whether the message was posted. In general, if an application intends to post many messages to the queue, it should set the message queue to an appropriate size when it creates the queue. The default message-queue size is 10 messages.

Message Types

This section describes the three types of OS/2 messages:

- System-defined
- Application-defined
- Semaphore.

System-Defined Messages

There are many *system-defined* messages that are used to control the operations of applications and to provide input and other information for applications to process. The system sends or posts a system-defined message when it communicates with an application. An application also can send or post system-defined messages. Usually, applications use these messages to control the operation of control windows created by using preregistered window classes.

Each system message has a unique message identifier and a corresponding symbolic constant. The symbolic constant, defined in the system header files, states the purpose of the message. For example, the WM_PAINT constant represents the paint message, which requests that a window paint its contents.

The symbolic constants also specify the *message category*. System-defined messages can belong to several categories; the prefix identifies the type of window that can interpret and process the messages. The following table lists the prefixes and their related message categories:

Table 3-1. Message Categories

Prefix	Message category
BKM_	Notebook control
BM_	Button control
CBM_	Combination-box control
CM_	Container control
EM_	Entry-field control
LM_	List-box control
MLM_	Multiple-line entry field control
MM_	Menu control
SBM_	Scroll-bar control
SLM_	Slider control
SM_	Static control
TBM_	Title-bar control
VM_	Value set control
WM_	General window

General window messages cover a wide range of information and requests, including:

- Mouse and keyboard-input
- Menu- and dialog-input
- Window creation and management
- Dynamic data exchange (DDE).

Application-Defined Messages

An application can create messages to use in its own windows. If an application does create messages, the window procedure that receives the messages must interpret them and provide the appropriate processing.

The operating system reserves the message-identifier values in the range *0x0000* through *0x0FFF* (the value of WM_USER - 1) for system-defined messages. Applications cannot use these values for their private messages.

In addition, the operating system uses certain message values higher than WM_USER. Applications should not use these message values. A partial listing of these messages is in the following figure:

```
From PMSTDDLG.H:

#define FDM_FILTER        WM_USER+40
#define FDM_VALIDATE      WM_USER+41
#define FDM_ERROR         WM_USER+42

#define FNTM_FACENAMECHANGED   WM_USER+50
#define FNTM_POINTSIZECHANGED  WM_USER+51
#define FNTM_STYLECHANGED      WM_USER+52
#define FNTM_COLORCHANGED      WM_USER+53
#define FNTM_UPDATEPREVIEW     WM_USER+54
#define FNTM_FILTERLIST        WM_USER+55
```

You should scan your header files to see if other messages have been defined with values higher than WM_USER.

Aside from the message values used by the operating system, values in the range *0x1000* (the value of WM_USER) through *0xBFFF* are available for message identifiers, defined by an application, for use in that application.

Warning: It is very important that applications do not broadcast messages in the *0x1000* through *0xBFFF* range due to the risk of misinterpretation by other applications.

Values in the range *0xC000* through *0xFFFF* are reserved for message identifiers that an application defines and registers with the system atom table; these can be used in any application. Values above *0xFFFF (0x00010000* through *0xFFFFFFFF)* are reserved for future use; applications must not use messages in this range.

Semaphore Messages

A *semaphore message* provides a way of signaling, through the message queue, the end of an event. An application uses a semaphore message the same way it uses system semaphore functions—to coordinate events by passing signals. A semaphore message often is used in conjunction with system semaphores.

There are four semaphore messages:

WM_SEM1
WM_SEM2
WM_SEM3
WM_SEM4.

An application posts one of these messages to signal the end of a given event. The window that is waiting for the given event receives the semaphore message when the message loop retrieves and dispatches the message.

Each semaphore message includes a bit flag that an application can use to uniquely identify the 32 possible semaphores for each semaphore message. The application passes the bit flag (with the appropriate bit set) as a message parameter with the message. The window procedure that receives the message then uses the bit flag to identify the semaphore.

To save space, the system does not store semaphore messages in the message queue. Instead, it sets a record in the queue, indicating that the semaphore message has been received, and then combines the bit flag for the message with the bit flags from previous messages. When the window procedure eventually receives the message, the bit flag specifies each semaphore message posted since the last message was retrieved.

Message Priorities

The WinGetMsg function retrieves messages from the message queue based on message priority. WinGetMsg retrieves messages with higher priority first. If it finds more than one message at a particular priority level, it retrieves the oldest message first. Messages have the following priorities:

Table 3-2. Message Priorities

Priority	Message
1	WM_SEM1
2	Messages posted using WinPostMsg
3	Input messages from the keyboard or mouse
4	WM_SEM2
5	WM_PAINT
6	WM_SEM3
7	WM_TIMER
8	WM_SEM4

Message Filtering

An application can choose specific messages to retrieve from the message queue (and ignore other messages) by specifying a message filter with the WinGetMsg or WinPeekMsg functions. The message filter is a range of message identifiers (specified by a first and last identifier), a window handle, or both. The WinGetMsg and WinPeekMsg functions use the *message filter* to select the messages to retrieve from the queue. Message filtering is useful if an application needs to search ahead in the message queue for messages that have a lower priority or that arrived in the queue later than other less important messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, filtering for a WM_CHAR message in a window that does not receive keyboard input prevents the WinGetMsg function from returning. Some messages, such as WM_COMMAND, are generated from other messages; filtering for them also can prevent WinGetMsg from returning.

To filter for mouse, button, and DDE messages, an application can use the following constants:

WM_MOUSEFIRST and WM_MOUSELAST
WM_BUTTONCLICKFIRST and WM_BUTTONCLICKLAST
WM_DDE_FIRST and WM_DDE_LAST.

Using Messages

This section explains how to perform the following tasks:

- Create a message queue and message loop.
- Examine the message queue.
- Post and send messages between windows.
- Broadcast a message to multiple windows.
- Use message macros.

Creating a Message Queue and Message Loop

An application needs a message queue and message loop to process messages for its windows. An application creates a message queue by using the WinCreateMsgQueue function. An application creates a message loop by using the WinGetMsg and WinDispatchMsg functions. The application must create and show at least one window after creating the queue but before starting the message loop. Figure 3-2 on page 3-12 shows how to create a message queue and message loop:

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2);

HAB hab;

int main(VOID)
{
    HMQ hmq;
    QMSG qmsg;
    HWND hwndFrame, hwndClient;
    ULONG flFrameFlags = FCF_TITLEBAR      | FCF_SYSMENU |
                         FCF_SIZEBORDER    | FCF_MINMAX  |
                         FCF_SHELLPOSITION | FCF_TASKLIST;

                                           /* Initialize the application for
                                     Presentation Manager interface.     */

    hab = WinInitialize(0);

                                           /* Create the application
                                           message queue.                */
    hmq = WinCreateMsgQueue(hab, 0);

                                           /* Register the window class for your
                                           client window.                */
    WinRegisterClass(hab,                  /* Anchor block handle        */
                     "MyClientClass",      /* Class name                 */
                     (PFNWP) ClientWndProc,/* Window procedure           */
                     CS_SIZEREDRAW,        /* Class style                */
                     0);                   /* Extra bytes to reserve     */

                                           /* Create a main window.      */
    hwndFrame = WinCreateStdWindow(
                    HWND_DESKTOP,          /* Parent window handle       */
                    WS_VISIBLE,            /* Style of frame window      */
                    &flFrameFlags,         /* Frame controls             */
                    "MyClientClass",       /* Window class for client    */
                    (PSZ) NULL,            /* No title-bar text          */
                    WS_VISIBLE,            /* Style of client window     */
                    (HMODULE) NULL,        /* Module handle for resources */
                    0,                     /* No resource identifier     */
                    &hwndClient);          /* Pointer to client handle   */

                                           /* Start the message loop.    */
    while (WinGetMsg(hab, &qmsg, (HWND) NULL, 0, 0))
        WinDispatchMsg(hab, &qmsg);

                                           /*. Destroy the main window.  */
    WinDestroyWindow(hwndFrame);

                                           /* Destroy the message queue. */
    WinDestroyMsgQueue(hmq);

                                           /* Terminate the application. */
    WinTerminate(hab);
}
```

Figure 3-2. Creating a Message Queue and Message Loop

Both the WinGetMsg and WinDispatchMsg functions take a pointer to a QMSG structure as a parameter. If a message is available, WinGetMsg copies it to the QMSG structure; WinDispatchMsg then uses the data in the structure as arguments for the window procedure.

Occasionally, an application might need to process a message before dispatching it. For example, if a message is posted but the destination window is not specified (that is, the message contains a NULL window handle), the application must process the message to determine which window should receive the message. Then the WinDispatchMsg function can forward the message to the proper window. The following code fragment shows how the message loop can process messages that have NULL window handles:

```
HAB hab;
QMSG qmsg;

while (WinGetMsg (hab, &qmsg, (HWND) NULL, 0, 0)) {
    if (qmsg.hwnd == NULL) {
        .
        . /* Process the message. */
        .
    }
    else
        WinDispatchMsg (hab, &qmsg);
    }
```

Examining the Message Queue

An application can examine the contents of the message queue by using the WinPeekMsg or WinQueryQueueStatus function. It is useful to examine the queue if the application starts a lengthy operation that additional user input might affect, or if the application needs to look ahead in the queue to anticipate a response to user input.

An application can use WinPeekMsg to check for specific messages in the message queue. This function is useful for extracting messages for a specific window from the queue. It returns immediately if there is no message in the queue. An application can use WinPeekMsg in a loop without requiring the loop to wait for a message to arrive. The following code fragment checks the queue for WM_CHAR messages:

```
HAB hab;
QMSG qmsg;

if (WinPeekMsg(hab, &qmsg, (HWND) NULL, WM_CHAR, WM_CHAR, PM_NOREMOVE)){
          .
          . /* Process the message. */
          .
}
```

An application also can use the WinQueryQueueStatus function to check for messages in the queue. This function is very fast and returns information about the kinds of messages available in the queue and which messages have been posted recently. Most applications use this function in message loops that need to be as fast as possible.

Posting a Message to a Window

An application can use the WinPostMsg function to post a message to a window. The message goes to the window's message queue. The following code fragment posts the WM_QUIT message.

```
HWND hwnd;

if (!WinPostMsg(hwnd, WM_QUIT, NULL, NULL)){

    /* Message was not posted. */
}
```

The WinPostMsg function returns FALSE if the queue is full, and the message cannot be posted.

Sending a Message to a Window

An application can use the WinSendMsg function to send a message directly to a window. An application uses this function to send messages to child windows. For example, the following code fragment sends an LM_INSERTITEM message to direct a list-box control to add an item to the end of its list:

```
HWND hwndListBox;
static CHAR szWeekday[] = "Tuesday";

WinSendMsg(hwndListBox,
           LM_INSERTITEM,
           (MPARAM)LIT_END,
           MPFROMP(szWeekday));
```

WinSendMsg calls the window's window procedure and waits for it to handle the message and return a result. An application can send a message to any window in the system, as long as the application has the handle of the target window. The message queue does not store the message; however, the thread making the call must have a message queue.

Broadcasting a Message

An application can send a message to multiple windows by using the WinBroadcastMsg function. Often this function is used to broadcast the WM_SYSVALUECHANGED message after an application changes a system value. The following code fragment shows how to broadcast this message to all frame windows in all applications:

```
HWND hwnd;

WinBroadcastMsg(
    hwnd,                                /* Window handle         */
    WM_SYSVALUECHANGED,                  /* Message identifier    */
    NULL,                                /* No message parameters */
    NULL,
    BMSG_FRAMEONLY | BMSG_POSTQUEUE);    /* All frame windows     */
```

An application can broadcast messages to all windows, just frame windows, or just the windows in the application.

Using Message Macros

The system header files define several macros that help create and interpret message parameters.

One set of macros helps you construct message parameters. These macros are useful for sending and posting messages. For example, the following code fragment uses the MPFROMSHORT macro to convert a 16-bit integer into the 32-bit message parameter:

```
HWND hwndButton;

WinSendMsg(hwndButton, BM_SETCHECK, MPFROMSHORT(1), NULL);
```

A second set of macros helps you extract values from a message parameter. These macros are useful for handling messages in a window procedure. The following code fragment determines whether the window receiving the WM_FOCUSCHANGE message is gaining or losing the keyboard focus. The fragment uses the SHORT1FROMMP macro to extract the focus-change flag, the SHORT2FROMMP macro to extract the focus flag, and the HWNDFROMMP macro to extract the window handle.

```
USHORT fsFocusChange;
MPARAM mp1, mp2;
HWND hwndGainFocus;


case WM_FOCUSCHANGE:
    fsFocusChange = SHORT2FROMMP(mp2);    /* Gets focus-change flags  */
    if (SHORT1FROMMP(mp2))                /* Gaining or losing focus? */
        hwndGainFocus = HWNDFROMMP(mp1);
```

A third set of macros helps you construct a message result. These macros are useful for returning message results in a window procedure, as the following code fragment illustrates:

```
return (MRFROM2SHORT(1, 2));
```

Related Functions

This section covers the functions that are related to Messages.

WinBroadcastMsg

This function broadcasts a message to multiple windows.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinBroadcastMsg  (HWND hwndParent, ULONG ulMsgId,
                       MPARAM mpParam1, MPARAM mpParam2,
                       ULONG flCmd)
```

Parameters

hwndParent (HWND) – input
Parent-window handle.

ulMsgId (ULONG) – input
Message identifier.

mpParam1 (MPARAM) – input
Parameter 1.

mpParam2 (MPARAM) – input
Parameter 2.

flCmd (ULONG) – input
Broadcast message command.

BMSG_POST
Post the message. This value is mutually exclusive with BMSG_SEND and BMSG_POSTQUEUE.

BMSG_SEND
Send the message. This value is mutually exclusive with BMSG_POST and BMSG_POSTQUEUE.

BMSG_POSTQUEUE
Post a message to all threads that have a message queue. This value is mutually exclusive with BMSG_POST and BMSG_SEND. The *hwnd* parameter of the QMSG structure is set to NULL.

BMSG_DESCENDANTS
Broadcast the message to all the descendants of the *hwndParent* parameter.

BMSG_FRAMEONLY
Broadcast the message only to windows with a style of CS_FRAME.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Message was sent or posted successfully to all applicable windows
FALSE	Error occurred.

WinCallMsgFilter

This function calls a message-filter hook.

Syntax

```
#define INCL_WINHOOKS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinCallMsgFilter  (HAB hab, PQMSG pqmsg, ULONG ulFilter)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pqmsg (PQMSG) – input
Message to be passed to the message-filter hook.

ulFilter (ULONG) – input
Filter.

MSGF_DIALOGBOX	Dialog-box mode loop.
MSGF_TRACK	Window-movement and size tracking. When this hook is used the TRACKINFO structure specified the *ptiTrackinfo* parameter of the WinTrackRect function is updated to give the current state before the hook is called. Only the *rclTrack* and the *fs* parameters are updated.
MSGF_DRAG	Direct manipulation mode loop.
MSGF_DDEPOSTMSG	DDE post message mode loop.

Returns

rc (BOOL) – returns
Message-filter hook return indicator.

TRUE	A message-filter hook returns TRUE
FALSE	All message-filter hooks return FALSE, or no message-filter hooks are defined.

WinCreateMsgQueue

This function creates a message queue.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

HMQ WinCreateMsgQueue  (HAB  hab, LONG  lQueuesize)
```

Parameters

hab (HAB) – input
Anchor-block handle.

lQueuesize (LONG) – input
Maximum queue size.

0	Use the system default queue size; that is 10 messages.
Other	Maximum queue size.

Returns

hmq (HMQ) – returns
Message-queue handle.

NULLHANDLE	Queue cannot be created.
Other	Message-queue handle.

WinDestroyMsgQueue

This function destroys the message queue.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
```

BOOL WinDestroyMsgQueue (HMQ hmq)

Parameters

hmq (HMQ) – input

Message-queue handle.

Returns

rc (BOOL) – returns

Queue-destroyed indicator.

TRUE	Queue destroyed
FALSE	Queue not destroyed.

WinDispatchMsg

This function invokes a window procedure.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
MRESULT WinDispatchMsg  (HAB  hab, PQMSG  pqmsgMsg)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pqmsgMsg (PQMSG) – input
Message structure.

Returns

mresReply (MRESULT) – returns
Message-return data.

WinGetMsg

This function gets, waiting if necessary, a message from the thread's message queue and returns when a message conforming to the filtering criteria is available.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinGetMsg  (HAB hab, PQMSG pqmsgmsg, HWND hwndFilter,
                 ULONG ulFirst, ULONG ulLast)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pqmsgmsg (PQMSG) – output
Message structure.

hwndFilter (HWND) – input
Window filter.

ulFirst (ULONG) – input
First message identity.

ulLast (ULONG) – input
Last message identity.

Returns

rc (BOOL) – returns
Continue message indicator.

TRUE	Message returned is not a WM_QUIT message
FALSE	Message returned is a WM_QUIT message.

WinInSendMsg

This function determines whether the current thread is processing a message sent by another thread.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinInSendMsg  (HAB  hab)
```

Parameters

hab (HAB) – input

Anchor-block handle.

Returns

rc (BOOL) – returns

Message-processing indicator.

TRUE Current thread is processing a message sent by another thread
FALSE Current thread is not processing a message, or an error occurred.

WinPeekMsg

This function inspects the thread's message queue and returns to the application with or without a message.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinPeekMsg  (HAB hab, PQMSG pqmsg, HWND hwndFilter,
                  ULONG ulFirst, ULONG ulLast, ULONG flOptions)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pqmsg (PQMSG) – output
Message structure.

hwndFilter (HWND) – input
Window filter.

ulFirst (ULONG) – input
First message identity.

ulLast (ULONG) – input
Last message identity.

flOptions (ULONG) – input
Options.

PM_REMOVE	Remove message from queue
PM_NOREMOVE	Do not remove message from queue.

Returns

rc (BOOL) – returns
Message-available indicator.

TRUE	Message available
FALSE	No message available.

WinPostMsg

This function posts a message to the message queue associated with the window defined by *hwnd*.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinPostMsg  (HWND hwnd, ULONG ulMsgid, MPARAM mpParam1,
                  MPARAM mpParam2)
```

Parameters

hwnd (HWND) – input
Window handle.

NULL The message is posted into the queue associated with the current thread. When the message is received by using the WinGetMsg or WinPeekMsg functions, the *hwnd* parameter of the QMSG structure is NULL.

Other Window handle.

ulMsgid (ULONG) – input
Message identity.

mpParam1 (MPARAM) – input
Parameter 1.

mpParam2 (MPARAM) – input
Parameter 2.

Returns

rc (BOOL) – returns
Message-posted indicator.

TRUE Message successfully posted
FALSE Message could not be posted; for example, because the message queue was full.

WinPostQueueMsg

This function posts a message to a message queue.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinPostQueueMsg  (HMQ hmq, ULONG msg, MPARAM mp1,
                       MPARAM mp2)
```

Parameters

hmq (HMQ) – input
Message-queue handle.

msg (ULONG) – input
Message identifier.

mp1 (MPARAM) – input
Parameter 1.

mp2 (MPARAM) – input
Parameter 2.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred, or the queue was full.

WinQueryMsgPos

This function returns the pointer position, in screen coordinates, when the last message obtained from the current message queue is posted.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryMsgPos  (HAB  hab, PPOINTL  pptl)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pptl (PPOINTL) – output
Pointer position in screen coordinates.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQueryQueueInfo

This function returns the information for the specified queue.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinQueryQueueInfo  (HMQ hmq, PMQINFO pmqiMqinfo, ULONG cbCopied)
```

Parameters

hmq (HMQ) – input
Queue handle.

pmqiMqinfo (PMQINFO) – output
Message queue information structure to contain the queue information.

cbCopied (ULONG) – input
Size of message queue information structure that is provided (in bytes).

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQueryQueueStatus

This function returns a code indicating the status of the message queue associated with the caller.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG WinQueryQueueStatus  (HWND  hwndDesktop)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle.

HWND_DESKTOP The desktop-window handle
Other Desktop-window handle returned by WinQueryDesktopWindow.

Returns

flStatus (ULONG) – returns
Status information.

Summary
Summary of message types existing on the queue.

This field contains a combination of the following values:

QS_KEY An input event (keyboard or journaling) has caused a WM_CHAR message to be placed in the queue.

QS_MOUSE An input event has caused a WM_MOUSEMOVE, WM_BUTTON1UP, WM_BUTTON1DOWN, WM_BUTTON1DBLCLK, WM_BUTTON2UP, WM_BUTTON2DOWN, WM_BUTTON2DBLCLK, WM_BUTTON3UP, WM_BUTTON3DOWN, or WM_BUTTON3DBLCLK message to be placed in the queue.

QS_MOUSEBUTTON An input event has caused a WM_BUTTON1UP, WM_BUTTON1DOWN, WM_BUTTON1DBLCLK, WM_BUTTON2UP, WM_BUTTON2DOWN, WM_BUTTON2DBLCLK, WM_BUTTON3UP, WM_BUTTON3DOWN, or WM_BUTTON3DBLCLK message to be placed in the queue.

QS_MOUSEMOVE An input event has caused a WM_MOUSEMOVE message to be placed in the queue.

QS_TIMER
A timer event has caused a WM_TIMER message to be placed in the queue.

QS_PAINT
A WM_PAINT message is available.

QS_SEM1
A WM_SEM1 message is available.

QS_SEM2
A WM_SEM2 message is available.

QS_SEM3
A WM_SEM3 message is available.

QS_SEM4
A WM_SEM4 message is available.

QS_POSTMSG
A message has been posted to the queue. Note that this message is probably not one of the messages listed above, but could be a WM_CHAR, WM_MOUSEMOVE or similar message if an application has posted one of these. In this case, the corresponding input status flag (QS_KEY, QS_MOUSE, and so on) is not set.

QS_SENDMSG
A message has been sent by another application to a window associated with the current queue.

Added
Message type additions.

Message types added to the queue since the last use of this function. The value of this field is a subset of the *Summary* field.

WinReleaseHook

This function releases an application hook from a hook chain.

Syntax

```
#define INCL_WINHOOKS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinReleaseHook  (HAB hab, HMQ hmq, LONG lHook, PFN pAddress,
                      HMODULE Module)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hmq (HMQ) – input
Handle of message queue from which the hook is to be released.

HMQ_CURRENT The hook is released from the message queue associated with the current thread (calling thread).

NULLHANDLE The hook is released from the system hook chain.

lHook (LONG) – input
Type of hook chain.

HK_CHECKMSGFILTER	See CheckMsgFilterHook.
HK_CODEPAGECHANGE	See CodePageChangedHook.
HK_DESTROYWINDOW	See DestroyWindowHook.
HK_HELP	See HelpHook.
HK_INPUT	See InputHook.
HK_JOURNALPLAYBACK	See JournalPlaybackHook.
HK_JOURNALRECORD	See JournalRecordHook.
HK_LOADER	See LoaderHook.
HK_MSGCONTROL	See MsgControlHook.
HK_MSGFILTER	See MsgFilterHook.
HK_SENDMSG	See SendMsgHook.

pAddress (PFN) – input
Address of the hook routine.

Module (HMODULE) – input
Module handle.

NULLHANDLE The hook procedure is in the application's .EXE file.

Module This is the module that contains the application procedure, as returned by the DosLoadModule or DosQueryModuleHandle call.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinRegisterUserMsg

This function registers a user message and defines its parameters.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinRegisterUserMsg  (HAB hab, ULONG msgid, LONG datatype1,
                          LONG dir1, LONG datatype2, LONG dir2,
                          LONG datatyper)
```

Parameters

hab (HAB) – input
Anchor-block handle.

msgid (ULONG) – input
Message identifier.

datatype1 (LONG) – input
Data type of message parameter 1.

DTYP_BIT16	See BIT16 data type.
DTYP_BIT32	See BIT32 data type.
DTYP_BIT8	See BIT8 data type.
DTYP_BOOL	See BOOL data type.
DTYP_LONG	See LONG data type.
DTYP_SHORT	See SHORT data type.
DTYP_UCHAR	See UCHAR data type.
DTYP_ULONG	See ULONG data type.
DTYP_USHORT	See USHORT data type.
DTYP_P*	A pointer to a system data type. Note that not all of the system data types that exist in the CPI are valid.
< -DTYP_USER	A pointer to a user data type. The user data type must have already been defined via WinRegisterUserDatatype.

dir1 (LONG) – input
Direction of message parameter 1.

RUM_IN	Input parameter (inspected by the recipient of the message, but not altered)

RUM_OUT	Output parameter (altered by the recipient of the message, without inspecting its value first)
RUM_INOUT	Input/output parameter (inspected by the recipient of the message, and then altered).

datatype2 (LONG) – input
Data type of message parameter 2.

dir2 (LONG) – input
Direction of message parameter 2.

datatyper (LONG) – input
Data type of message reply.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSendMsg

This function sends a message with identity *ulMsgid* to *hwnd*, passing *mpParam1* and *mpParam2* as the parameters to the window.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

MRESULT WinSendMsg  (HWND hwnd, ULONG ulMsgid, MPARAM mpParam1,
                     MPARAM mpParam2)
```

Parameters

hwnd (HWND) – input
Window handle.

ulMsgid (ULONG) – input
Message identity.

mpParam1 (MPARAM) – input
Parameter 1.

mpParam2 (MPARAM) – input
Parameter 2.

Returns

mresReply (MRESULT) – returns
Message-return data.

WinSetClassMsgInterest

This function sets the message interest of a window class.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetClassMsgInterest  (HAB hab, PSZ pszClassName,
                              ULONG ulMsgClass, LONG lControl)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pszClassName (PSZ) – input
Window-class name.

ulMsgClass (ULONG) – input
Message class to have interest level set.

msgid	A single message identity (for example, WM_SHOW).
SMIM_ALL	All messages (except for WM_QUIT if *lControl* is SMI_AUTODISPATCH or SMI_NOINTEREST).

lControl (LONG) – input
Interest identifier for the message class.

SMI_INTEREST	Interested in the message, or messages
SMI_NOINTEREST	Not interested in the message, or messages
SMI_AUTODISPATCH	Interested in the message or messages, but they are to be automatically dispatched to the window procedure.

Returns

rc (BOOL) – returns
Interest-changed indicator.

TRUE	Interest successfully changed
FALSE	Interest not successfully changed.

WinSetMsgInterest

This function sets a window's message interest.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetMsgInterest  (HWND hwnd, ULONG ulMsgClass, LONG lControl)
```

Parameters

hwnd (HWND) – input
Window handle.

ulMsgClass (ULONG) – input
Message class to have interest level set.

msgid	A single message identity (for example, WM_SHOW)
SMIM_ALL	All messages (except for WM_QUIT if *lControl* is SMI_AUTODISPATCH or SMI_NOINTEREST).

lControl (LONG) – input
Interest-identifier for the message class.

SMI_RESET	Revert to interest specified for the window class.
SMI_INTEREST	Interested in the messages.
SMI_NOINTEREST	Not interested in the messages.
SMI_AUTODISPATCH	Interested in the message or messages, but they are to be automatically dispatched to the window procedure.

Returns

rc (BOOL) – returns
Interest-changed indicator.

TRUE	Interest successfully changed
FALSE	Interest not successfully changed.

WinSetMsgMode

This function indicates the mode for the generation and processing of messages for the private window class of an application.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinSetMsgMode  (HAB hab, PSZ pszClassName, LONG lControl)
```

Parameters

hab (HAB) – input
Anchor block handle.

pszClassName (PSZ) – input
Window class name.

lControl (LONG) – input
Message mode identifier.

SMD_DELAYED	The generation of messages may be delayed
SMD_IMMEDIATE	The generation of messages will not be delayed.

Returns

rc (BOOL) – returns
Message delay indicator.

TRUE	Message mode successfully set
FALSE	Message mode not successfully set.

WinWaitMsg

This function waits for a filtered message.

Syntax

```
#define INCL_WINMESSAGEMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinWaitMsg  (HAB hab, ULONG ulFirst, ULONG ulLast)
```

Parameters

hab (HAB) – input
Anchor-block handle.

ulFirst (ULONG) – input
First message identity.

ulLast (ULONG) – input
Last message identity.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Messages.

WM_FOCUSCHANGE

This message occurs when the window possessing the focus is changed.

Parameters

param1

hwndFocus (HWND)
Focus window handle.

param2

usSetFocus (USHORT)
Focus flag.

TRUE The window is receiving the focus and *hwndFocus* identifies the window losing the focus.

FALSE The window is losing the focus and *hwndFocus* identifies the window receiving the focus.

fsFocusChange (USHORT)
Focus changing indicators.

The indicators are passed from the WinFocusChange function.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_QUIT

This message is posted to terminate the application.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SEM1

This message is sent or posted by an application.

Parameters

param1

flAccumBits (ULONG)
Semaphore value.

The semaphore values from all the WM_SEM1 messages posted to a queue, are accumulated by a logical-OR operation.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SEM2

This message is sent or posted by an application.

Parameters

param1

flAccumBits (ULONG)
Semaphore value.

The semaphore values from all the WM_SEM2 messages posted to a queue, are accumulated by a logical-OR operation.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SEM3

This message is sent or posted by an application.

Parameters

param1

flAccumBits (ULONG)
Semaphore value.

The semaphore values from all the WM_SEM3 messages posted to a queue, are accumulated by a logical-OR operation.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SEM4

This message is sent or posted by an application.

Parameters

param1

flAccumBits (ULONG)
Semaphore value.

The semaphore values from all the WM_SEM4 messages posted to a queue, are accumulated by a logical-OR operation.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SYSVALUECHANGED

This message is posted to all main windows when one of the settable system values is changed.

Parameters

param1

usChangedFirst (USHORT)
First system value.

The first of a contiguous set of system values that has been changed.

param2

usChangedLast (USHORT)
Last system value.

The last of a contiguous set of system values that has been changed.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Messages.

HMQ

Message-queue handle.

Syntax

```
typedef &handle. HMQ;
```

MQINFO

Message-queue information structure.

Syntax

```
typedef struct _MQINFO {
ULONG       cb;
PID         pid;
TID         tid;
ULONG       cmsgs;
PVOID       pReserved;
 } MQINFO;

typedef MQINFO *PMQINFO;
```

Fields

cb (ULONG)
Length of structure.

pid (PID)
Process identity.

tid (TID)
Thread identity.

cmsgs (ULONG)
Message count.

pReserved (PVOID)
Reserved.

QMSG

Message structure.

Syntax

```
typedef struct _QMSG {
HWND         hwnd;
ULONG        msg;
MPARAM       mp1;
MPARAM       mp2;
ULONG        time;
POINTL       ptl;
ULONG        reserved;
 } QMSG;

typedef QMSG *PQMSG;
```

Fields

hwnd (HWND)
Window handle.

msg (ULONG)
Message identity.

mp1 (MPARAM)
Parameter 1.

mp2 (MPARAM)
Parameter 2.

time (ULONG)
Message time.

ptl (POINTL)
Pointer position when message was generated.

reserved (ULONG)
Reserved.

Summary

Following are the functions and structures used with OS/2 messages and message queues.

Table 3-3. Commonly Used Message and Message Queue Functions

Function Name	Description
WinCreateMsgQueue	Creates a message queue.
WinDefDlgProc	Invokes the default dialog procedure.
WinDefWindowProc	Invokes the default window procedure.
WinDestroyMsgQueue	Destroys the message queue.
WinDispatchMsg	Invokes a window procedure.
WinGetMsg	Gets a message from the thread's message queue and returns *msg* when a message conforming to the filtering criteria is available.
WinPeekMsg	Inspects the thread's message queue and returns to the application with or without a message.
WinPostMsg	Posts a message to the message queue associated with the window defined by *hwnd*.
WinSendDlgItemMsg	Sends a message to the dialog item defined by *item* in the dialog window specified by *Dlg*
WinSendMsg	Sends a message with identity *Msgid* to *hwnd*.

Table 3-4. Seldom-Used Message and Message Queue Functions

Function Name	Description
WinBroadcastMsg	Broadcasts a message to multiple windows.
WinCallMsgFilter	Calls a message-filter hook.
WinInSendMsg	Determines whether the current thread is processing a message sent by another thread.
WinPostQueueMsg	Posts a message to a message queue.

Table 3-5. Almost-Never Used Message and Message Queue Functions

Function Name	Description
WinQueryMsgPos	Returns the pointer position, in screen coordinates, when the last message obtained from the current message queue is posted.
WinQueryQueueInfo	Returns the information for the specified queue.
WinQueryQueueStatus	Returns a code indicating the status of the message queue associated with the caller.
WinRegisterUserMsg	Registers a user message and defines its parameters.
WinReleaseHook	Releases an application hook from a hook chain.
WinSetClassMsgInterest	Sets the message interest of a message class.
WinSetMsgInterest	Sets a window's message interest.
WinSetMsgMode	Indicates the mode for the generation and processing of messages for the private window class of an application.
WinTranslateAccel	Translates a WM_CHAR message.
WinWaitMsg	Waits for a filtered message.

Table 3-6. Related Messages

Message	Description
WM_FOCUSCHANGE	This message occurs when the window possessing the focus is changed.
WM_QUIT	This message is posted to terminate the application.
WM_SEM1	This message is sent or posted by an application.
WM_SEM2	This message is sent or posted by an application.
WM_SEM3	This message is sent or posted by an application.
WM_SEM4	This message is sent or posted by an application.
WM_SYSVALUECHANGED	This message is posted to all main windows when one of the settable system values is changed.

Table 3-7. Message and Message Queue Structures

Structure Name	Description
HMQ	Message-queue handle.
MQINFO	Message-queue information structure.
QMSG	Message structure.

Chapter 4. Window Classes

A *window class* determines which styles and which window procedure are given to a window when it is created. This chapter explains how a PM application creates and uses window classes.

About Window Classes

Every window is a member of a window class. An application must specify a window class when it creates a window. Each window class has an associated *window procedure* that is used by all windows of the same class. The window procedure handles messages for all windows of that class and, therefore, controls the behavior and appearance of the window.

A window class must be *registered* before an application can create a window of that class. Registering a window class associates a window procedure and class styles with a class name. When an application specifies the class name in a window-creation function such as WinCreateWindow, the system creates a window that uses the window procedure and styles associated with the class name.

An application can register private classes or use preregistered public window classes.

Private Window Classes

A *private window class* is any class registered within an application. An application registers a private class by calling the WinRegisterClass function. A private class cannot be shared with other applications. When an application terminates, the system removes any data associated with the application's private window classes.

An application can register a private class anytime but, typically, does so as part of application initialization. To register a private class during application initialization, the application also must call WinInitialize and, usually, WinCreateMsgQueue before class registration.

An application cannot de-register a private window class; it remains registered and available until the application terminates.

When an application registers a private window class, it must supply the following information:

- Class name
- Class styles
- Window procedure
- Window data size.

Class Name

The *class name* identifies the window class. The application uses this name in the window-creation functions to specify the class of the window being created. The class name can be a character string or an atom, and it must be unique within the application. The system checks as to whether a public class or a class already registered by the application has the same name. If the class name is not unique to that application, the system returns an error.

Class Styles

Each window class has one or more values, called *class styles*, that tell the system which initial window styles to give a window created with that class. An application sets the class styles for a private window class when it registers the class. Once a class is registered, the application cannot change the styles.

An application can specify one or more of the following class styles in the WinRegisterClass function, combining them as necessary by using the bitwise OR operator:

Table 4-1 (Page 1 of 2). Class Styles

Style Name	Description
CS_CLIPCHILDREN	Prevents a window from painting over its child windows, but increases the time necessary to calculate the visible region. This style usually is not necessary, because if the parent and child windows overlap and are both invalidated, the operating system draws the parent window before drawing the child window. If the child window is invalidated independently of the parent window, the system redraws only the child window. If the update region of the parent window does not intersect the child window, drawing the parent window causes the child window to be redrawn. This style is useful to prevent a child window containing a complex graphic from being redrawn unnecessarily.
CS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows. This style protects sibling windows but increases the time necessary to calculate the visible region. This style is appropriate for windows that overlap and have the same parent window.
CS_FRAME	Identifies the window as a frame window.
CS_HITTEST	Directs the operating system to send WM_HITTEST messages to the window whenever the mouse pointer moves in the window.
CS_MOVENOTIFY	Directs the system to send WM_MOVE messages to the window whenever the user moves the window.
CS_PARENTCLIP	Extends a window's visible region to include that of its parent window. This style simplifies the calculation of the child window's visible region but, potentially, is dangerous, because the parent window's visible region is usually larger than the child window.
CS_SAVEBITS	Saves the screen area under a window as a bit map. When the user hides or moves the window, the system restores the image by copying the bits; there is no need to add the area to the uncovered window's update region. This style can improve system performance, but also can consume a great deal of memory. It is recommended only for transient windows such as menus and dialog windows—*not* for main application windows.

Table 4-1 (Page 2 of 2). Class Styles

Style Name	Description
CS_SIZEREDRAW	Causes the window to receive a WM_PAINT message and be completely invalidated whenever the window is resized, even if it is made smaller. (Typically, only the uncovered area of a window is invalidated when a window is resized.) This class style is useful when an application scales graphics to fill the window.
CS_SYNCPAINT	Causes the window to receive WM_PAINT messages immediately after a part of the window becomes invalid. Without this style, the window receives WM_PAINT messages only if no other message is waiting to be processed.

Window Procedure

The window procedure for a window class processes all messages sent or posted to all windows of that class. It is the chief component of the window class because it controls the appearance and behavior of each window created with the class. Window procedures are shared by all windows of a class, so an application must ensure that no conflicts arise when two windows of the same class attempt to access the same global data. In other words, the window procedure must protect global data and other shared resources.

Window Data Size

The system creates a window data structure for each window, which includes extra space that an application can use to store additional data about a window. An application specifies the number of extra bytes to allocate in the WinRegisterClass function. All windows of the same class have the same amount of window data space.

An application can store window data in a window's data structure by using the WinSetWindowUShort and WinSetWindowULong functions. It can retrieve data by using the WinQueryWindowUShort and WinQueryWindowULong functions.

Custom Window Styles

An application that registers a window class also can support its own set of styles for windows of that class. Standard window styles—for example, WS_VISIBLE and WS_SYNCPAINT—still apply to these windows. A window style is a 32-bit integer, and only the high 16 bits are used for the standard window styles; an application can use the low 16 bits for custom styles specific to a window class.

The operating system has unique window styles for all preregistered window classes. Styles such as FS_BORDER and BS_PUSHBUTTON are processed by the window procedure for the corresponding class. This means that an application can build the support for its own window styles into the window procedure for its private class. A window style designed for one window class will not work with another window class.

Public Window Classes

Public window classes are registered during system initialization. Their window procedures are in dynamic link libraries. Therefore, to use a public window class, an application need not register it. Nor does the application need to import the window procedure for a public window class because the system resolves references to the window procedure.

An application cannot use a public window class name when it registers a private window class.

System-Defined Public Window Classes

The system provides a number of public window classes that support menus, frame windows, control windows, and dialog windows. An application can create a window of a system-defined public window class by specifying one of the following class name constants in a call to WinCreateWindow:

Table 4-2 (Page 1 of 2). Public Window Classes

Class Name	Description
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_COMBOBOX	Creates a combination-box control, which combines a list-box control and an entry-field control. It enables the user to enter data either by typing in the entry field or by choosing from the list in the list box.
WC_CONTAINER	Creates a control in which the user can group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A composite window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as menu bars, or vertically as pull-down menus. Usually menus are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.
WC_SCROLLBAR	Consists of window scroll bars that let the user scroll the contents of the associated window.
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.

Table 4-2 (Page 2 of 2). Public Window Classes

Class Name	Description
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

Each system-defined public window class has a corresponding set of window styles that an application can use to customize a window of that class. For example, a window created with the WC_BUTTON class has styles that include BS_PUSHBUTTON and BS_CHECKBOX. Window styles enable you to customize aspects of a window's behavior and appearance. The application specifies the window styles in the WinCreateWindow function.

Custom Public Window Classes

An application can create a custom public window class, but it must do so during system initialization. Only the shell can register a public window class, and it can do so only when the system starts. Registering a public window class requires a special load entry in the os2.ini file. That entry instructs the shell to load a dynamic link library whose initialization routine registers the window class. Custom public window classes must be registered using WinRegisterClass and must have the class style CS_PUBLIC. If a custom public window class registered this way has the same name as an existing public window class, the custom class replaces the original class.

If a dynamic link library replaces an existing public window class, the library can save the address of the original window procedure and use the address to subclass the original window class. The dynamic link library retrieves the original window procedure address using the WinQueryClassInfo function. The custom window procedure then passes unprocessed messages to the original window procedure instead of calling WinDefWindowProc.

When subclassing a public window class, the custom public window procedure must not make the window data size smaller than the original window data size, because all public window classes that the operating system defines use 4 extra bytes for storing a pointer to custom window data. This size is guaranteed only for public window classes defined by the operating system dynamic link libraries.

Class Data

An application can examine public window class data by using the WinQueryClassInfo and WinQueryClassName functions. An application retrieves the name of the class for a given window by using the WinQueryClassName function. If the window is one of the preregistered public window classes, the name returned is in the form *#nnnnn*, where *nnnnn* is up to 5 digits, representing the value of the window class constant. Using this window class name, the application can call WinQueryClassInfo to retrieve the window class data. WinQueryClassInfo copies the class style, window procedure address, and window data size to a CLASSINFO data structure.

Using Window Classes

This section explains how to perform the following tasks:

- Register a private window class.
- Register an imported window procedure.

Registering a Private Window Class

An application can register a private window class at any time by using the WinRegisterClass function. You must define the window procedure in the application, choose a unique name, and set the window styles for the class. The following code fragment shows how to register the window class name “MyPrivateClass”:

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1, MPARAM mp2);

HAB hab;

WinRegisterClass(hab,      /* Anchor block handle           */
    "MyPrivateClass",      /* Name of class being registered */
    ClientWndProc,         /* Window procedure for class    */
    CS_SIZEREDRAW |        /* Class style                   */
    CS_HITTEST,            /* Class style                   */
    0);                    /* Extra bytes to reserve        */
```

Related Functions

This section covers the functions that are related to Windows Classes.

WinQueryClassInfo

This function returns window class information.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryClassInfo  (HAB hab, PSZ PSZClassName,
                         PCLASSINFO PclsiClassInfo)
```

Parameters

hab (HAB) – input
Anchor-block handle.

PSZClassName (PSZ) – input
Class name.

PclsiClassInfo (PCLASSINFO) – output
Class information structure.

Returns

rc (BOOL) – returns
Class-exists indicator.

TRUE	Class does exist
FALSE	Class does not exist.

WinQueryClassName

This function copies the window class name, as a null-terminated string, into a buffer.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinQueryClassName  (HWND  hwnd, LONG  lLength, PCH  PCHBuffer)
```

Parameters

hwnd (HWND) – input
  Window handle.

lLength (LONG) – input
  Length of *PCHBuffer*.

PCHBuffer (PCH) – output
  Class name.

Returns

lRetLen (LONG) – returns
  Returned class name length.

WinRegisterClass

This function registers a window class.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>

BOOL WinRegisterClass  (HAB hab, PSZ pszClassName, PFNWP pfnWndProc,
                        ULONG flStyle, ULONG cbWindowData)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pszClassName (PSZ) – input
Window-class name.

pfnWndProc (PFNWP) – input
Window-procedure identifier.

flStyle (ULONG) – input
Default-window style.

cbWindowData (ULONG) – input
Reserved storage.

Returns

rc (BOOL) – returns
Window-class-registration indicator.

TRUE	Window class successfully registered
FALSE	Window class not successfully registered.

Related Data Structures

This section covers the data structures that are related to Window Classes.

CLASSINFO

Class-information structure.

Syntax

```
typedef struct _CLASSINFO {
ULONG       flClassStyle;
PFNWP       pfnWindowProc;
ULONG       cbWindowData;
 } CLASSINFO;

typedef CLASSINFO *PCLASSINFO;
```

Fields

flClassStyle (ULONG)
Class-style flags.

pfnWindowProc (PFNWP)
Window procedure.

cbWindowData (ULONG)
Number of additional window words.

Summary

Following are the operating system functions and structure used with window classes.

Table 4-3. Window Class Functions

Function Name	Description
WinQueryClassInfo	Returns window class information.
WinQueryClassName	Copies, into a buffer, the window class name as a null-terminated string.
WinRegisterClass	Registers a window class.
WinSubclassWindow	Subclasses the indicated window by replacing its window procedure with another window procedure.

Table 4-4. Window Class Structure

Structure Name	Description
CLASSINFO	Class-information structure.

Chapter 5. Window Procedures

Windows have an associated *window procedure*—a function that processes all messages sent or posted to a window. Every aspect of a window's appearance and behavior depends on the window procedure's response to the messages. This chapter explains how window procedures function, in general, and describes the default window procedure.

About Window Procedures

Every window belongs to a window class that determines which window procedure a particular window uses to process its messages. All windows of the same class use the same window procedure. For example, the operating system defines a window procedure for the frame window class (WC_FRAME), and all frame windows use that window procedure.

An application typically defines at least one new window class and an associated window procedure. Then, the application can create many windows of that class, all of which use the same window procedure. This means that the same piece of code can be called from several sources simultaneously; therefore, you must be careful when modifying shared resources from a window procedure.

Dialog procedures have the same structure and function as window procedures. The primary difference between a dialog procedure and a window procedure is the absence of a client window in the dialog procedure; that is, the controls in a dialog procedure are the immediate child windows of the frame, whereas the controls in a normal window are the *grandchildren* of the frame. This makes significant differences in the code between the two; for example, WinSendDlgItemMsg does not work from a client window if you pass the client window handle as the first parameter.

Structure of a Window Procedure

A window procedure is a function that takes 4 arguments and returns a 32-bit pointer. The arguments of a window procedure consist of a window handle, a ULONG message identifier, and two arguments, called *message parameters*, that are declared with the MPARAM data type. The system defines an MPARAM as a 32-bit pointer to a VOID data type (a generic pointer). The message parameters actually might contain any of the standard data types. The message parameters are interpreted differently,depending on the value of the message identifier. The operating system includes several macros that enable the application to cast the information from the MPARAM values into the actual data type. SHORT1FROMMP, for example, extracts a 16-bit value from a 32-bit MPARAM.

The window-procedure arguments are described in the following table:

Table 5-1. Window Procedure Arguments

Argument	Description
hwnd	Handle of the window receiving the message.
msg	Message identifier. The message will correspond to one of the predefined constants (for example, WM_CREATE) defined in the system include files or be an application-defined message identifier. The value of an application-defined message identifier must be greater than the value of WM_USER, and less than or equal to *0xffff.*
mp1,mp2	Message parameters. Their interpretation depends on the particular message.

The return value of a window procedure is defined as an MRESULT data type. The interpretation of the return value depends on the particular message. Consult the description of each message to determine the appropriate return value.

Default Window Procedure

All windows in the system share certain fundamental behavior, defined in the default window-procedure function, WinDefWindowProc. The default window procedure provides the minimal functionality for a window. An application-defined window procedure should pass any messages it does not process to WinDefWindowProc for default processing.

Window-Procedure Subclassing

Subclassing enables an application to intercept and process messages sent or posted to a window before that window has a chance to process them. Subclassing most often is used to add functionality to a particular window or to alter a window's default behavior.

An application subclasses a window by using the WinSubclassWindow function to replace the window's original window procedure with an application-defined window procedure. Thereafter, the new window procedure processes any messages that are sent or posted to the window. If the new window procedure does not process a particular message, it must pass the message to the original window procedure, *not* to WinDefWindowProc, for default processing.

Using Window Procedures

This section explains how to:

- Design a window procedure
- Associate a window procedure with a window class
- Subclass a window.

Designing a Window Procedure

The following code fragment shows the structure of a typical window procedure and how to use the message argument in a switch statement, with individual messages handled by separate case statements. Notice that each case returns a specific value for each message. For messages that it does not handle itself, the window procedure calls WinDefWindowProc.

```
MRESULT ClientWndProc(
HWND hwnd,
ULONG msg,
MPARAM mp1,
MPARAM mp2)
{
    /* Define local variables here, if required. */
    switch (msg) {
        case WM_CREATE:

    /* Initialize private window data.           */
        return (MRESULT) FALSE;

        case WM_PAINT:

    /* Paint the window.                         */
        return 0;

        case WM_DESTROY:

    /* Clean up private window data.             */
        return 0;

        default:
        break;
     }
     return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

A dialog window procedure does not receive the WM_CREATE message; however, it does receive a WM_INITDLG message when all of its control windows have been created.

At the very least, a window procedure should handle the WM_PAINT message to draw itself. Typically, it should handle mouse and keyboard messages as well. Consult the descriptions of individual messages to determine whether your window procedure should handle them.

An application can call WinDefWindowProc as part of the processing of a message. In such a case, the application can modify the message parameters before passing the message to WinDefWindowProc or can continue with the default processing after performing its own operations.

Associating a Window Procedure with a Window Class

To associate a window procedure with a window class, an application must pass a pointer to that window procedure to the WinRegisterClass function. Once an application has registered the window procedure, the procedure automatically is associated with each new window created with that class.

The following code fragment shows how to associate the window procedure in the previous example with a window class:

```
HAB hab;
CHAR szClientClass[] = "My Window Class";

WinRegisterClass(hab,          /* Anchor-block handle  */
    szClientClass,             /* Class name           */
    ClientWndProc,             /* Pointer to procedure */
    CS_SIZEREDRAW,             /* Class style          */
    0);                        /* Window data          */
```

Subclassing a Window

To subclass a window, an application calls the WinSubclassWindow function, specifying the handle of the window to subclass and a pointer to the new window procedure. The WinSubclassWindow function returns a pointer to the original window procedure; the application can use this pointer to pass unprocessed messages to the original procedure.

The following code fragment subclasses a push button control window. The new window procedure generates a beep whenever the user clicks the push button.

```
    PFNWP pfnPushBtn;
    CHAR szCancel[] = "Cancel";
    HWND hwndClient;
    HWND hwndPushBtn;
        .
        .
        .

    /* Create a push button control.                */
    hwndPushBtn = WinCreateWindow(
        hwndClient,     /* Parent-window handle      */
        WC_BUTTON,      /* Window class              */
        szCancel,       /* Window text               */
        WS_VISIBLE    | /* Window style              */
        WS_SYNCPAINT.|  /* Window style              */
        BS_PUSHBUTTON,  /* Button style              */
        50, 50,         /* Physical position         */
        70, 30,         /* Width and height          */
        hwndClient,     /* Owner-window handle       */
        HWND_TOP,       /* Z-order position          */
        1,              /* Window identifier         */
        NULL,           /* No control data           */
        NULL);          /* No presentation parameters */

    /* Subclass the push button control.            */
    pfnPushBtn = WinSubclassWindow(hwndPushBtn,
        SubclassPushBtnProc);
        .
        .
        .
}
    /* This procedure subclasses the push button.   */
    MRESULT EXPENTRY SubclassPushBtnProc(HWND hwnd,ULONG msg,MPARAM mp1, MPARAM mp2)
    {
        switch (msg) {

    /* Beep when the user clicks the push button.   */
            case WM_BUTTON1DOWN:
                DosBeep(1000, 250);
                break;

            default:
                break;
        }

    /* Pass all messages to the original window procedure. */
        return (MRESULT) pfnPushBtn(hwnd, msg, mp1, mp2);
    }
```

Related Functions

This section covers the functions that are related to Windows Procedures.

WinDefDlgProc

This function invokes the default dialog procedure with *hwndDlg*, *msg*, *mp1*, and *mp2*.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
MRESULT WinDefDlgProc  (HWND hwndDlg, ULONG msg, MPARAM mp1,
                        MPARAM mp2)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

msg (ULONG) – input
Message identity.

mp1 (MPARAM) – input
Parameter 1.

mp2 (MPARAM) – input
Parameter 2.

Returns

mresReply (MRESULT) – returns
Message-return data.

WinDefWindowProc

This function invokes the default window procedure.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>

MRESULT WinDefWindowProc  (HWND hwnd, ULONG ulMsgid,
                           MPARAM mpParam1, MPARAM mpParam2)
```

Parameters

hwnd (HWND) – input
Window handle.

ulMsgid (ULONG) – input
Message identity.

mpParam1 (MPARAM) – input
Parameter 1.

mpParam2 (MPARAM) – input
Parameter 2.

Returns

mresReply (MRESULT) – returns
Message-return data.

WinSubclassWindow

This function subclasses the indicated window by replacing its window procedure with another window procedure, specified by *pNewWindowProc*.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

PFNWP WinSubclassWindow  (HWND hwnd, PFNWP pNewWindowProc)
```

Parameters

hwnd (HWND) – input
   Handle of window that is being subclassed.

pNewWindowProc (PFNWP) – input
   New window procedure.

Returns

pOldWindowProc (PFNWP) – returns
   Old window procedure.

Related Messages

This section covers the messages that are related to Windows Procedures.

WM_BUTTON1DBLCLK

This message occurs when the operator presses button 1 of the pointing device twice within a specified time, as detailed below.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON1DOWN

This message occurs when the operator presses pointer button one.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit test process, which determined the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON1UP

This message occurs when the operator releases button 1 of the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON2DBLCLK

This message occurs when the operator presses button 2 of the pointing device twice within a specified time, as detailed in "WM_BUTTON1DBLCLK" on page 5-9.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON2DOWN

This message occurs when the operator presses button 2 on the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit test process, which determined the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON2UP

This message occurs when the operator releases button 2 of the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON3DBLCLK

This message occurs when the operator presses button 3 of the pointing device twice within a specified time, as detailed in "WM_BUTTON1DBLCLK" on page 5-9.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON3DOWN

This message occurs when the operator presses button 3 on the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit test process, which determined the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_BUTTON3UP

This message occurs when the operator releases button 3 of the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

fsHitTestres (USHORT)
Hit-test result.

fsHitTestres provides the hit-test result. It contains the value returned from the hit-test process, which determines the window to be associated with this message. For details of the possible values, see "WM_HITTEST" on page 5-23.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_CHAR

This message is sent when an operator presses a key.

Parameters

param1

fsflags (USHORT)
Keyboard control codes.

KC_CHAR
Indicates that *usch* value is valid.

KC_SCANCODE
Indicates that *ucscancode* is valid.

Generally, this is set in all WM_CHAR messages generated from actual operator input. However, if the message has been generated by an application that has issued the WinSetHook function to filter keystrokes, or posted to the application queue, this may not be set.

KC_VIRTUALKEY
Indicates that *usvk* is valid.

Normally *usvk* should be given precedence when processing the message.

Note: For those using hooks, when this bit is set, KC_SCANCODE should usually be set as well.

KC_KEYUP
The event is a key-up transition; otherwise it is a down transition.

KC_PREVDOWN
The key has been previously down; otherwise it has been previously up.

KC_DEADKEY
The character code is a dead key. The application is responsible for displaying the glyph for the dead key without advancing the cursor.

KC_COMPOSITE
The character code is formed by combining the current key with the previous dead key.

KC_INVALIDCOMP
The character code is not a valid combination with the preceding dead key. The application is responsible for advancing the cursor past the dead-key glyph and then, if the current character is not a space, sounding the alarm and displaying the new character code.

KC_LONEKEY
Indicates if the key is pressed and released without any other keys being pressed or released between the time the key goes down and up.

KC_SHIFT
The SHIFT state is active when key press or release occurred.

KC_ALT
The ALT state is active when key press or release occurred.

KC_CTRL	The CTRL state was active when key press or release occurred.

ucrepeat (UCHAR)
Repeat count.

ucscancode (UCHAR)
Hardware scan code.

A keyboard-generated value that identifies the keyboard event. This is the raw scan code, not the translated scan code.

param2

usch (USHORT)
Character code.

The character value translation of the keyboard event resulting from the current code page that would apply if the CTRL or ALT keys were not depressed.

usvk (USHORT)
Virtual key codes.

A virtual key value translation of the keyboard event resulting from the virtual key code table. The low-order byte contains the **vk** value, and the high-order byte is always set to zero by the standard translate table.

0 This value applies if *fsflags* does not contain KC_VIRTUALKEY.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_COMMAND

This message occurs when a control has a significant event to notify to its owner, or when a key stroke has been translated by an accelerator table.

Parameters

param1

uscmd (USHORT)
Command value.

It is the responsibility of the application to be able to relate *uscmd* to an application function.

param2

ussource (USHORT)
Source type.

Identifies the type of control:

CMDSRC_PUSHBUTTON	Posted by a push-button control. *uscmd* is the window identity of the push button.
CMDSRC_MENU	Posted by a menu control. *uscmd* is the identity of the menu item.
CMDSRC_ACCELERATOR	Posted as the result of an accelerator. *uscmd* is the accelerator command value.
CMDSRC_FONTDLG	Font dialog. *uscmd* is the identity of the font dialog.
CMDSRC_FILEDLG	File dialog. *uscmd* is the identity of the file dialog.
CMDSRC_OTHER	Other source. *uscmd* gives further control-specific information defined for each control type.

uspointer (USHORT)
Pointer-device indicator.

TRUE	The message is posted as a result of a pointer-device operation.
FALSE	The message is posted as a result of a keyboard operation.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_CONTROLPOINTER

This message is sent to a owner window of a control when the pointing device pointer moves over the control window, allowing the owner to set the pointing device pointer.

Parameters

param1

usidCtl (USHORT)
Control identifier.

param2

hptrNew (HPOINTER)
Handle of the pointing device pointer that the control is to use.

Returns

hptrRet (HPOINTER)
Returned pointing device-pointer handle that is then used by the control.

WM_HELP

This message occurs when a control has a significant event to notify to its owner or when a key stroke has been translated by an accelerator table into a WM_HELP.

Parameters

param1

uscmd (USHORT)
Command value.

It is the responsibility of the application to be able to relate *uscmd* to an application function.

param2

ussource (USHORT)
Source type.

Identifies the type of control:

CMDSRC_PUSHBUTTON	Posted by a push-button control. *uscmd* is the window identity of the push button.
CMDSRC_MENU	Posted by a menu control. *uscmd* is the identity of the menu item.
CMDSRC_ACCELERATOR	Posted as the result of an accelerator. *uscmd* is the accelerator command value.
CMDSRC_OTHER	Other source. *uscmd* gives further control-specific information defined for each control type.

uspointer (USHORT)
Pointer-device indicator.

TRUE	If the message is posted as a result of a pointer-device operation
FALSE	If the message is posted as a result of a keyboard operation.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_HITTEST

This message is sent to determine which window is associated with an input from the pointing device.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulresult (ULONG)
Hit-test indicator.

HT_NORMAL	The message should be processed as normal. A WM_MOUSEMOVE, WM_BUTTON2DOWN, or WM_BUTTON1DOWN message is posted to the window.
HT_TRANSPARENT	The part of the window underneath the pointer is transparent; hit-testing should continue on windows underneath this window, as if the window did not exist.
HT_DISCARD	The message should be discarded; no message is posted to the application.
HT_ERROR	As HT_DISCARD, except that if the message is a button-down message, an alarm sounds and the window concerned is brought to the foreground.

WM_MENUSELECT

This message occurs when a menu item has been selected.

Parameters

param1

usItem (USHORT)
Identifier of selected item.

usPostCommand (USHORT)
Post-command flag.

TRUE	Indicates that either a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message is being posted by the menu control on return from the owner, subject to *rc*.
FALSE	Indicates that no message is being posted by the menu control on return from the owner, subject to *rc*.

param2

hwnd (HWND)
Menu-control window handle.

Returns

rc (BOOL)
Post indicator.

TRUE	Indicates that either a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message is to be posted by the menu control window procedure. The menu is dismissed if the selected item does not have a style of MIA_NODISMISS.
FALSE	Indicates that no message is to be posted by the menu control window procedure and that the menu is not dismissed.

WM_MOUSEMOVE

This message occurs when the pointing device pointer moves.

Parameters

param1

sxMouse (SHORT)
&Pdev. x-coordinate.

syMouse (SHORT)
&Pdev. y-coordinate.

param2

uswHitTest (USHORT)
Message result.

Zero	A pointing device capture is currently in progress
Other	The result of the WM_HITTEST message.

fsflags (USHORT)
Keyboard control codes.

In addition to the control codes described with the WM_CHAR message, the following keyboard control codes are valid.

KC_NONE	Indicates that no key is pressed

Returns

rc (BOOL)
Processed indicator.

TRUE	The window procedure did process the message.
FALSE	The window procedure did not process the message.

WM_QUERYCONVERTPOS

This message is sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.

Parameters

param1

pCursorPos (PRECTL)
Cursor position.

If *usCode* = QCP_CONVERT, *pCursorPos* should be updated to contain the position of the cursor in the window receiving this message. The position is specified as a rectangle in screen coordinates.

If *usCode* = QCP_NOCONVERT, *pCursorPos* should not be updated.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

usCode (USHORT)
Conversion code.

QCP_CONVERT	Conversion may be performed for the window with the input focus, *pCursorPos* has been updated to contain the position of the cursor.
QCP_NOCONVERT	Conversion should not be performed, the window with the input focus cannot receive DBCS characters, *pCursorPos* has not been updated.

WM_QUERYFOCUSCHAIN

This message is used to request the handle of a window in the focus chain.

Parameters

param1

fsCmd (USHORT)
Command to be performed.

This field contains a flag to indicate what action is to be performed:

QFC_NEXTINCHAIN
Return the next window in the focus chain.

The *hwndParent* parameter is not used.

QFC_ACTIVE
Return the handle of the frame window that would be activated or deactivated, if this window gains or loses the focus.

The window handle returned is a child of the window specified by the *hwndParent* parameter.

QFC_FRAME
Return the handle of the first frame window associated with this window.

The *hwndParent* parameter is not used.

QFC_SELECTACTIVE
Return the handle of the window from the group of owned windows to which this window belongs which either currently has the focus or, if no window has the focus, previously had the focus.

Return NULL, if no window in the owner group has had the focus.

The *hwndParent* parameter is not used.

QFC_PARTOFCHAIN
Return TRUE if the handle of the window identified by the *hwndParent* parameter is in the focus chain, otherwise return FALSE.

Because this message is passed along the focus chain, this is equivalent to returning TRUE, if the handle of the window receiving this message is *hwndParent* or to returning FALSE, if it is not.

param2

hwndParent (HWND)
Parent window.

Returns

hwndResult (HWND)

Handle of the window requested.

0	No window handle exists for this case of the *fsCmd* parameter This value is also to be interpreted as FALSE for the case when the *fsCmd* is set to QFC_PARTOFCHAIN.
Other	Handle of the window requested. This value is also to be interpreted as TRUE for the cases when the *fsCmd* is set to QFC_PARTOFCHAIN.

WM_SETSELECTION

This message occurs when a window is selected or deselected.

Parameters

param1

usselection (USHORT)
Selection flag.

TRUE	The window is selected.
FALSE	The window is deselected.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_TRANSLATEACCEL

This message is sent to the focus window whenever a WM_CHAR message occurs.

Parameters

param1

pqmsg (PQMSG)
Pointer to a QMSG structure.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Translated indicator.

TRUE	The character exists in the accelerator table and has been translated in the QMSG structure.
FALSE	The character does not exist in the accelerator table or the window does not have an accelerator table.

Summary

Following are the window-procedure functions and messages processed by the default window procedure.

Table 5-2. Window Procedure Functions

Function Name	Description
WinDefDlgProc	The default dialog procedure.
WinDefWindowProc	The default window procedure.
WinRegisterClass	Registers a window class.
WinSubclassWindow	Subclasses the indicated window by replacing its window procedure.

Table 5-3 (Page 1 of 2). Default Window Procedure Messages

Message	Description
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON1UP	Occurs when the user releases pointer button 1.
WM_BUTTON2DBLCLK	Occurs when the user presses button 2 of the pointing device twice.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON2UP	Occurs when the user releases pointer button 2.
WM_BUTTON3DBLCLK	Occurs when the user presses button 3 of the pointing device twice.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON3UP	Occurs when the user releases pointer button 3.
WM_CALCVALIDRECTS	Sent to determine which areas of a window can be preserved if a window is sized and which can be redisplayed.
WM_CHAR	Occurs when the user presses a key.
WM_CLOSE	Sent to a frame window to indicate that the window is being closed by the user.
WM_COMMAND	Sent when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointer moves over the control window, allowing the user to set the pointer.
WM_FOCUSCHANGE	Occurs when the focus window is changed.

Table 5-3 (Page 2 of 2). Default Window Procedure Messages

Message	Description
WM_HELP	Occurs when a control has a significant event to notify to its owner, or when a key stroke has been translated into a WM_HELP by an accelerator table.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_MENUSELECT	Occurs when a menu item is selected.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin DBCS conversion.
WM_QUERYFOCUSCHAIN	Requests the handle of a window in the focus chain.
WM_QUERYFRAMECTLCOUNT	Sent to the frame window in response to receipt of a WM_SIZE or WM_UPDATEFRAME message.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_SETSELECTION	Occurs when a window is selected or deselected.
WM_TIMER	Posted when a timer times out.
WM_TRANSLATEACCEL	Sent to the focus window when a WM_CHAR message occurs.

Chapter 6. Frame Windows

A *frame window* is the basic window used by most Presentation Manager applications to enable the user to perform manipulation functions. This chapter explains how to create and use frame windows in PM applications.

About Frame Windows

An application nearly always starts with a frame window to create a *composite window* (for example, a main window) that consists of the frame window, several frame-control windows, and a client window. The frame controls conform to the Common User Access (CUA) user interface guidelines. The frame window coordinates the actions of the frame controls and client window, enabling the composite window to act as a single unit.

Frame windows have the preregistered public window class WC_FRAME. The frame-window class, like the preregistered control classes, defines the appearance and behavior of the frame window.

Main Window

The *main window* of an application, typically, is composed of a frame window and a client window. The frame window usually includes control windows such as a title bar, system menu, menu bar (*action bar* or *menu* in user terminology), and scroll bars. Figure 6-1 on page 6-2 is an example of a typical frame window.

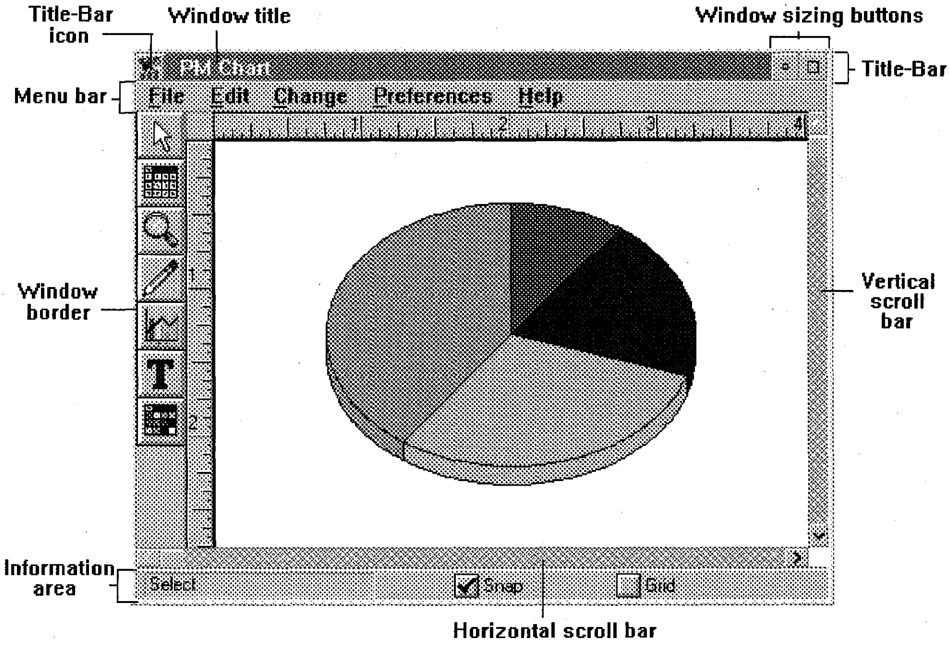


Figure 6-1. Typical Frame Window and Its Components

A frame window provides the standard services the user expects from a window—for example, moving, sizing, minimizing, and maximizing. The frame window receives input from the control windows (called *frame controls*) and sends messages to both the frame controls and the client window.

Frame Controls

When creating a frame window, an application also can create one or more frame controls as child windows of the frame window. Most frame windows contain at least a system menu and title bar. Other optional controls might include a menu bar and scroll bar as shown above.

An application can create a frame window with specified frame controls by calling WinCreateStdWindow with the appropriate frame-control flags.

The frame window owns the child frame-control windows, which can send notification messages that tell the frame window what the user is doing with the frame controls. For example, using a mouse, a user can move a window by clicking the title bar and dragging the window to a new position. The title-bar control responds to the click by sending a message to the frame window, notifying it of the user's request to move the window. Then the frame window tracks the mouse motion and moves the frame window and all of its child windows to the new position.

PM, rather than the application, handles the processing of the frame controls, thus providing the user a consistent interface for manipulating and interacting with windowed applications on the screen. Frame controls are described in individual chapters. For more information about control windows, see Chapter 16, "Control Windows."

Client Window

Every main window has a *client window*, which is the window in which the application displays output and receives mouse and keyboard input from the user. What an application displays in the client window, how it displays it, and how it interprets input to the window are controlled by the client's application-defined window procedure.

An application creates the client window when it creates the frame window. The client window, which is specific to the application, is nearly always created using a *private window class* (a class registered by the application). Like a frame control, the client window is a child window and is owned by the frame window. This means, for example, that the client window is moved when the frame window moves, is clipped to the frame-window size, and is destroyed when the frame window is destroyed.

The relationship between the frame window and the client window allows the frame window to pass messages between other frame controls and the client window. For example, a client window can send a message to the frame window requesting that the frame window change the window title. The frame window, in turn, sends a message to the title-bar control, telling it to change the title of the window.

Additional Frame-Window Items

In addition to its frame controls, a frame window also can contain a sizing border and the minimize and maximize buttons (also known as minimize and maximize *icons*). These items are not frame controls, because the frame window draws and maintains them. (*Frame controls* are windows that draw and maintain themselves.)

The sizing border encloses the frame window and lets the user change the size of the window using a mouse. The minimize button, at the right end of the title bar, lets the user reduce the frame window to an icon. The maximize button, to the right of the minimize button, lets the user enlarge the window so that it fills the screen. An application can add these items to a frame window by using the FCF_SIZEBORDER, FCF_MAXBUTTON, and FCF_MINBUTTON (or FCF_MINMAX) styles. (The FCF_MINMAX style adds both a maximize button and a minimize button.)

Frame-Control Identifiers

A frame window uses a set of standard constants to identify the frame controls and the client window. The *frame-control identifiers* all begin with the prefix FID_ and can be used in functions such as WinWindowFromID to uniquely identify a given control or the client window. The frame controls also use these identifiers in notification messages sent to the frame window. The following table describes the frame-control identifiers:

Table 6-1. Frame-Control Identifiers

Identifier	Description
FID_CLIENT	Identifies a client window.
FID_HORZSCROLL	Identifies a horizontal scroll bar.
FID_MENU	Identifies a menu.
FID_MINMAX	Identifies the minimize and maximize (*window-sizing*) buttons.
FID_SYSMENU	Identifies a system menu.
FID_TITLEBAR	Identifies a title bar.
FID_VERTSCROLL	Identifies a vertical scroll bar.

Frame-Window Creation

An application typically creates a frame window by using WinCreateStdWindow, which creates a frame window, a client window, and the specified frame controls. The application also can call WinCreateWindow with the WC_FRAME window class, which creates the frame window and controls but not the client window. To create the client, the application can call WinCreateWindow, specifying the original frame window as the parent and owner.

An application also can use a frame window to create a dialog window. For a dialog window, the frame window contains control windows but no client window. The application creates the dialog window by using WinLoadDlg or WinCreateDlg. These functions require an appropriate dialog template from the application's resource-definition file. The dialog template specifies the styles and dimensions for the frame window and for the control windows that compose the dialog window.

Frame Window Controls and Styles

An application uses frame-control flags in WinCreateStdWindow to specify which frame controls to give to the frame window. Frame-control flags are constants that have the FCF_ prefix.

The frame-window class (WC_FRAME), like other public window classes, provides many class-specific window styles that applications can use to adapt the appearance and behavior of a frame window. To specify the frame-window styles, an application can use either frame-control flags or the frame-window style constants, which have the FS_ prefix. Each style constant has a corresponding frame-control flag. Both produce exactly the same styles in a frame window. Typically, if an application is creating a frame window that uses frame controls, the application uses frame-control flags to specify the frame-window styles—if not, the application uses frame-style constants. An application can combine the frame-style constants with the standard window styles when creating a frame window.

When an application calls WinCreateStdWindow without setting any frame-control flags, the function creates a standard window that is invisible and behind all its sibling windows, has a width and height of 0, and is positioned at the lower-left corner of its parent window. After the call to WinCreateStdWindow returns, the application can use WinSetWindowPos to change the window's size, coordinates, z-order position, and visibility.

If an application calls WinCreateStdWindow with the FCF_SHELLPOSITION frame-control flag, the function creates the window so that it is in front of its sibling windows and has a standard size and coordinates determined by the system.

Frame-Window Resources

If an application specifies FCF_ACCELTABLE, FCF_ICON, FCF_MENU, FCF_STANDARD, FS_ACCELTABLE, FS_ICON, or FS_STANDARD when creating a frame window, the application must provide the resources to support the specified style. Failure to do so causes the window creation to fail. Depending on the style, a frame window might attempt to load one or more resources from the application's executable files.

The following table shows the frame-control flags and frame-window styles that require resources:

Table 6-2. Frame Window Flags and Styles Requiring Resources

Flag	Style	Description
FCF_ACCELTABLE	**FS_ACCELTABLE**	Requires an accelerator-table resource. The frame window uses the accelerator table to translate WM_CHAR messages to WM_COMMAND, WM_HELP, or WM_SYSCOMMAND messages.
FCF_ICON	**FS_ICON**	Requires an icon resource. The frame window draws the icon when the user minimizes the window.
FCF_MENU	**FS_MENU**	Requires a menu-template resource. A frame window uses the menu template to create a menu containing the commands and menus specified by the resource.
FCF_STANDARD	**FS_STANDARD**	Requires a menu-template resource (FCF_STANDARD only), an accelerator-table resource, and an icon resource.

You can use the resource compiler to add icon, menu, and accelerator-table resources to the application's executable file. Each resource must have a resource identifier that matches the resource identifier specified in the FRAMECDATA structure passed to WinCreateWindow or in the *idResources* parameter of WinCreateStdWindow.

Note: For detailed information about icon, menu, and accelerator-table resources, see Chapter 10, "Mouse Pointers and Icons," Chapter 13, "Menus," and Chapter 14, "Keyboard Accelerators," respectively.

The following sample code illustrates how to use WinCreateStdWindow to load and set up certain resources for a frame window. Normally the first step is to set up a header file defining the the IDs of the applicable resources:

```
#define ID_RESOURCE 001

#define IDM_OPTIONS  50
#define IDM_SHIFT    51
#define IDM_EXIT     52
```

Figure 6-2. Defining Resources for Header File

Then, make a resource (.RC) file, defining each resource:

```
/* Sample Resource */

#include <os2.h>

POINTER ID_RESOURCE  sampres.ico              /* Icon              */

ACCELTABLE ID_RESOURCE
BEGIN                                        /* Accelerator table */
     VK_F10,   IDM_SHIFT,   VIRTUALKEY
     VK_F3 ,   IDM_EXIT,    VIRTUALKEY
END

MENU ID_RESOURCE                             /* Menu              */
BEGIN
     SUBMENU "~Options", IDM_OPTIONS
      BEGIN
         MENUITEM "~Shift Colors\tF10", IDM_SHIFT
         MENUITEM "~Exit\tF3",          IDM_EXIT
      END
END
```

Figure 6-3. Defining Resources for Resource (.RC) File

When using WinCreateStdWindow with more than one resource, each resource can have the same ID, as in the above example (ID_RESOURCE or *1*), *but only if each resource is of a different type*. Resources of the same type must have unique IDs. Use FCF flags to indicate what resources to load:

```
ULONG flFrameFlags=
                    FCF_TITLEBAR      |  /*  Title bar             */
                    FCF_SIZEBORDER    |  /*  Size border           */
                    FCF_MINMAX        |  /*  Min & Max buttons     */
                    FCF_SYSMENU       |  /*  System menu           */
                    FCF_SHELLPOSITION |  /*  System size & position */
                    FCF_TASKLIST      |  /*  Add name to task list */
                    FCF_ICON          |  /***Add icon              */
                    FCF_ACCELTABLE    |  /***Add accel. table      */
                    FCF_MENU          ;  /***Add menu              */
```

Figure 6-4. Using FCF Flags to Indicate What Resources to Load

Use 0 (or NULL) in the seventh parameter of WinCreateStdWindow to indicate that the resource is stored in the application file, as follows:

```
hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,      /* Parent is desktop window.          */
    WS_VISIBLE,        /* Make frame window visible.         */
    &flFrameFlags,     /* Frame controls                     */
    "ResSamClient",    /* Window class for client            */
    NULL,              /* No window title                    */
    WS_VISIBLE,        /* Make client window visible .       */
    (HMODULE) 0,       /* Resources in application module    */
    ID_RESOURCE,       /* Resource identifier                */
    NULL);             /* Pointer to client window handle    */
```

Figure 6-5. Indicating that a Resource is Stored in the Application File

Following is the full listing of the sample program:

```
#define INCL_PM
#include <os2.h>

MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2);

int main(int argc, char *argv, char *envp)
{
    HWND hwndFrame;
    HWND hwndClient;
    HMQ  hmq;
    QMSG qmsg;
    HAB  hab;


    ULONG flFrameFlags=
                        FCF_TITLEBAR      |  /*  Title bar             */
                        FCF_SIZEBORDER    |  /*  Size Border           */
                        FCF_MINMAX        |  /*  Min & Max Buttons     */
                        FCF_SYSMENU       |  /*  System Menu           */
                        FCF_SHELLPOSITION |  /*  System size & position */
                        FCF_TASKLIST      |  /*  Add name to task list. */
                        FCF_ICON          |  /***Add icon.             */
                        FCF_ACCELTABLE    |  /***Add accelerator table. */
                        FCF_MENU;            /***Add menu.             */

    hab = WinInitialize(0);

    hmq = WinCreateMsgQueue(hab, 0);

    WinRegisterClass(
        hab,                    /* Anchor block handle            */
        "ResSamClient",         /* Name of class being registered */
        (PFNWP)ClientWndProc,   /* Window procedure for class     */
        CS_SIZEREDRAW |         /* Class style                    */
        CS_HITTEST,             /* Class style                    */
        0);                     /* Extra bytes to reserve         */


    hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,      /* Parent is desktop window.           */
        WS_VISIBLE,        /* Make frame window visible.          */
        &flFrameFlags,     /* Frame controls                      */
        "ResSamClient",    /* Window class for client             */
        NULL,              /* No window title                     */
        WS_VISIBLE,        /* Make client window visible .        */
        (HMODULE) 0,       /* Resources in application module     */
        ID_RESOURCE,       /* Resource identifier                 */
        NULL);             /* Pointer to client window handle     */


    while (WinGetMsg(hab, &qmsg, 0, 0, 0))
           WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);

    return 0;
}
```

Figure 6-6 (Part 1 of 2). Sample Program for Loading Resources in a Frame Window

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2)
{
    RECTL rcl;
    HPS   hps;
    static LONG  lColor=CLR_RED;
    switch (msg) {

        case WM_PAINT:
              hps=WinBeginPaint(hwnd,(HPS) NULL, &rcl);     /* Get hps          */
              WinFillRect(hps,&rcl,lColor);                 /* Fill the window  */
              WinEndPaint(hps);                             /* Free hps         */
        return 0;

        case WM_COMMAND:

             switch (SHORT1FROMMP(mp1)) {
             case IDM_SHIFT:                                /* Shift selected   */
                if (lColor==CLR_RED) lColor=CLR_BLUE;       /* Change the       */
                else lColor=CLR_RED;                        /* color            */
                WinInvalidateRect(hwnd,(PRECTL)NULL,0UL);   /* Paint Window     */
                return 0;
             case IDM_EXIT:                                 /* Exit selected    */
                WinPostMsg(hwnd,WM_CLOSE,MPVOID,MPVOID);    /* Exit program.    */
                return 0;
             }

     }
     return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

Figure 6-6 (Part 2 of 2). Sample Program for Loading Resources in a Frame Window

Frame-Window Class Data

An application can specify class-specific data for a frame window by passing to WinCreateWindow a pointer to the FRAMECDATA structure. The class-specific data contains the frame-control flags (FCF_ flags), resource-module handle, and resource identifier to be used when creating the frame window. The resource-module handle and the resource identifier specify where to find resources for the frame window.

Supplying class-specific data with WinCreateWindow is similar to using WinCreateStdWindow without creating a client window.

Frame-Window Data

Frame-window data specifies the state of the frame window at a given time. An application can retrieve the frame-window data by calling WinQueryWindowUShort. A frame window has the following state flags:

Table 6-3. Frame Window State Flags and Their Meanings

Flag	Description
FF_ACTIVE	Indicates that the frame window is active.
FF_DLGDISMISSED	Indicates that a dialog window has been dismissed by a call to WinDismissDlg.
FF_FLASHHILITE	Indicates that the frame window is flashing and its flash state is TRUE.
FF_FLASHWINDOW	Indicates that the frame window flashes as the result of either a call to WinFlashWindow or a WM_FLASHWINDOW message.
FF_NOACTIVATESWP	Indicates that the system should do no z-ordering on this frame window.
FF_OWNERDISABLE	Indicates whether the owner window was enabled or disabled when the dialog window was loaded, for a frame window that is part of a dialog window,
FF_OWNERHIDDEN	Indicates that the frame window's owner window is hidden or minimized, in which case the frame window also is hidden.
FF_SELECTED	Indicates that the frame window has been selected.
FI_ACTIVATEOK	Indicates that the window can be activated.
FI_FRAME	Indicates that the window is a frame window.
FI_NOMOVEWITHOWNER	Indicates that the window should move when its owner window moves.
FI_OWNERHIDE	Indicates that the frame window should be hidden or shown as a result of its owner window being hidden, shown, minimized, or maximized.

Frame-Window Operation

The frame window maintains the size, position, and visibility of itself, its frame controls, and its client window. The frame window responds to user requests to move, size, minimize, maximize, and redraw itself. It also responds to requests to close (destroy) itself and to change the focus and activation state.

The frame window, when being moved or sized, maintains the position of each owned window relative to its owner window's lower-left corner.

Whenever the frame window redraws itself (for example, after being moved or sized), it draws the frame controls and then lets the application draw the client window. This order ensures that the rapidly drawn frame controls are drawn before the client window.

The order in which the frame controls are drawn depends on the z-order position of the controls. The following list specifies the z-order position of the frame controls (from top to bottom):

FID_SYSMENU
FID_TITLEBAR
FID_MENU
FID_VERTSCROLL
FID_HORZSCROLL
FID_CLIENT

Although an application can change the z-order position of any window, changing the relative positions of frame controls is not recommended.

When the user maximizes the frame window, the size of the frame window increases to the size of its parent window, plus an additional amount on each of its four sides equal to the width of its sizing border. A window always is clipped to its parent window; a maximized standard frame window does not show its sizing border in its normal maximized position.

Frame controls owned by a frame window or windows owned by child windows of a frame window are destroyed automatically when the frame window processes the WM_DESTROY message.

Nonstandard Frame Windows

Although most applications use frame windows to create their main windows and dialog windows, they are not limited to frame windows. Applications can create nonstandard frame windows and still use the standard frame controls, such as the title bar and system menu, within the nonstandard windows.

An application can create a nonstandard frame window either by subclassing a frame window or by creating a private frame-window class. An application that subclasses a frame window can intercept the messages sent to the window and process them in new ways. An application that creates private frame-window classes essentially rewrites the frame-window procedure. In either case, by creating nonstandard frame windows, the application gains much more control over the arrangement of frame controls in the frame window.

The messages WM_FORMATFRAME, WM_UPDATEFRAME, and WM_CALCVALIDRECTS control the arrangement of frame controls for applications that subclass the frame-window procedure. By intercepting these messages, an application can rearrange the frame controls in a frame window.

To maintain the size and position of frame controls, an application that creates private frame-window classes can use WinCreateFrameControls and WinCalcFrameRect. These functions provide capabilities that are similar to those provided by frame windows.

Default Frame-Window Behavior

The following table lists all the messages specifically handled by the window procedure of the predefined frame-window class (WC_FRAME) and describes how the window procedure responds to each message.

Table 6-4 (Page 1 of 2). Default Frame-Window Messages and Behavior

Message	Description
WM_ACTIVATE	Sets the highlighted state of the title bar or border so that it matches the frame window's activation state.
WM_BUTTON1DOWN	If the frame window is minimized, captures the mouse; otherwise, activates the frame window.
WM_BUTTON2DOWN	Activates the frame window.
WM_BUTTON3DOWN	Activates the frame window.
WM_BUTTON1UP	Processes messages from minimized window frames.
WM_BUTTON1DBLCLK	If the frame window is minimized, posts a WM_SYSCOMMAND message to itself; otherwise, activates the frame window.
WM_CALCVALIDRECTS	If the frame window has no client window or if the client window has the CS_SIZEREDRAW style, returns the CVR_REDRAW flag to invalidate the entire window.
WM_CLOSE	If the frame window has a client window, passes this message to the client; otherwise, returns the result of WinDefWindowProc.
WM_CREATE	Creates the specified frame controls by calling WinCreateFrameControls. Also creates any accelerator tables, loads icons, and adds itself to the Window List. These actions depend on the frame-window styles and frame-control flags specified for the window.

Table 6-4 (Page 2 of 2). Default Frame-Window Messages and Behavior

Message	Description
WM_DESTROY	If the focus is held by a child window of the frame window, sets the focus to the frame window's parent window, destroys any owned windows or child windows, destroys any icons created by using the FS_ICON style, and destroys any accelerator tables created by using the FS_ACCELTABLE style.
WM_ENABLE	Returns the result of WinDefWindowProc.
WM_ERASEBACKGROUND	Returns TRUE, signaling that the window should erase the client-window area. The frame window sends this message to itself during WM_PAINT processing.
WM_FORMATFRAME	Calculates the sizes and positions of the frame controls and the client window.
WM_HITTEST	If the frame window is minimized and disabled, returns HT_ERROR; otherwise, returns TF_MOVE.
WM_MINMAXFRAME	If the frame window has a client window, passes this message to the client window; otherwise, passes this message to WinDefWindowProc.
WM_MOUSEMOVE	Determines the correct mouse pointer to use and returns the result of WinDefWindowProc.
WM_PAINT	If the frame window is minimized, sends WM_QUERYICON and WM_ERASEBACKGROUND to itself and draws the icon; otherwise, paints the control windows, sends a WM_ERASEBACKGROUND message to the client window, and paints the client window.
WM_QUERYTRACKINFO	Starts track-move processing of the title-bar control window.
WM_SHOW	Returns the result of WinDefWindowProc.
WM_SIZE	Sends a WM_FORMATFRAME message to itself.
WM_SYSCOMMAND	If the frame window has captured the mouse, ignores the system command; otherwise, uses one of the following commands: SC_APPMENU, SC_CLOSE, SC_MOVE, SC_NEXT, SC_NEXTFRAME, SC_RESTORE, SC_SIZE, SC_SYSMENU, SC_TASKMANAGER.
WM_UPDATEFRAME	Reformats and updates the appearance of the frame window. Sent after a frame control has been added to or removed from the frame window.

Using Frame Windows

This section explains how to:

- Create a main window
- Retrieve a frame-control handle

Creating a Main Window

An application can create a main window by using WinCreateStdWindow. The following code fragment creates a typical main window—a frame window that has a system menu, title bar, menu, vertical and horizontal scroll bars, minimize and maximize (window-sizing) buttons, and a sizing border:

```
#define IDM_MENU 1

HWND hwndFrame;

ULONG flFrameControlFlags =
FCF_SYSMENU    | FCF_TITLEBAR | FCF_SIZEBORDER |
FCF_MENU       | FCF_MINMAX   | FCF_HORZSCROLL |
FCF_VERTSCROLL | FCF_SHELLPOSITION;

hwndFrame = WinCreateStdWindow(
HWND_DESKTOP,           /* Frame-window parent       */
WS_VISIBLE,             /* Make window visible       */
&flFrameControlFlags,   /* Frame-control flags       */
"MyClass",              /* Client-window class       */
"Main Window",          /* Window title              */
0,                      /* No client-window styles   */
(HMODULE)NULL,          /* App. module has resources */
IDM_MENU,               /* Resource ID               */
0);                     /* Client-window handle      */
```

An application also can create a *standard* main window by creating a frame window with the FCF_STANDARD flag. The application must include icon, menu, and accelerator-table resources if it uses the FCF_STANDARD flag.

The application creates the standard window by using WinCreateStdWindow, as shown in the following code fragment:

```
#define IDM_RESOURCES 1

HWND hwndFrame;

/* Set the frame-control flags.                               */
ULONG flFrameControlFlags = FCF_STANDARD;

/* Create the standard main window.                           */
hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
&flFrameControlFlags,
"MyClass", "Main Window", 0, (HMODULE) NULL,
IDM_RESOURCES, 0);
```

Another way to create a main window and its frame controls is to use WinCreateWindow to create the frame window and the frame controls, then call WinCreateWindow again to create the client window. One advantage of this approach is that, when creating the frame window, the application can specify the window's initial size and position. Figure 6-7 on page 6-16 illustrates this approach.

```
#define ID_RESOURCES 1
#define ID_FRAME     1

ULONG flFrameControlFlags =
FCF_ACCELTABLE | FCF_ICON       | FCF_MENU    |
FCF_MINMAX     | FCF_SIZEBORDER | FCF_SYSMENU |
FCF_TASKLIST   | FCF_TITLEBAR;

FRAMECDATA fcdata;
HWND hwndFrame;
HWND hwndClient;
SWP swp;

fcdata.cb = sizeof(FRAMECDATA);
fcdata.flCreateFlags = flFrameControlFlags;
fcdata.hmodResources = (HMODULE) NULL;
fcdata.idResources  = ID_RESOURCES;

/* Create the frame and client windows.             */
hwndFrame = WinCreateWindow(
HWND_DESKTOP,         /* Frame-window parent        */
WC_FRAME,             /* Frame-window class         */
"Main Window",        /* Window title               */
0,                    /* Initially invisible        */
0,0,0,0,              /* Size and position = 0      */
NULL,                 /* No owner                   */
HWND_TOP,             /* Top z-order position       */
ID_FRAME,             /* Frame-window ID            */
&fcdata,              /* Pointer to class data      */
NULL);                /* No presentation parameters */

hwndClient = WinCreateWindow(
hwndFrame,            /* Client-window parent       */
"MyClass",            /* Client-window class        */
NULL,                 /* No title for client window */
0,                    /* Initially invisible        */
0,0,0,0,              /* Size and position = 0      */
hwndFrame,            /* Owner is frame window      */
HWND_BOTTOM,          /* Bottom z-order position    */
FID_CLIENT,           /* Standard client-window ID  */
NULL,                 /* No class data              */
NULL);                /* No presentation parameters */

.
. /* Continue with initialization.                  */
.
```

Figure 6-7 (Part 1 of 2). Using WinCreateWindow to Create Frame, Control, and Client Windows

```
/* Set the size and position of the frame window.   */
WinQueryWindowPos(HWND_DESKTOP, &swp);
WinSetWindowPos(hwndFrame, HWND_TOP, swp.x, swp.cy / 2,
swp.cx, swp.cy / 2, SWP_MOVE | SWP_SIZE);

/* Set the size and position of the client window.  */
WinQueryWindowPos(hwndFrame, &swp);
WinSetWindowPos(hwndClient, HWND_TOP, SV_CXSIZEBORDER,
SV_CYSIZEBORDER - 1, swp.cx - SV_CXSIZEBORDER * 2,
(swp.cy - SV_CYSIZEBORDER * 2) + 1, SWP_MOVE | SWP_SIZE);

/* Make the frame and client windows visible.       */
WinShowWindow(hwndFrame, TRUE);
WinShowWindow(hwndClient, TRUE);
```

Figure 6-7 (Part 2 of 2). Using WinCreateWindow to Create Frame, Control, and Client Windows

Retrieving a Frame Handle

An application can retrieve a frame-control handle by using WinWindowFromID. The following code fragment retrieves the handle of a title-bar control:

```
HWND hwndTitleBar,hwndFrame;

hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

Given a frame-control handle, an application can retrieve its parent frame-window handle by using WinQueryWindow:

```
HWND hwndFrame,hwndTitleBar;

hwndFrame = WinQueryWindow(hwndTitleBar, QW_PARENT);
```

By using identifiers to identify frame controls, rather than using window classes, an application can create its own controls to replace the predefined controls.

Related Functions

This section covers the functions that are related to Frame Windows.

WinCalcFrameRect

This function calculates a client rectangle from a frame rectangle, or a frame rectangle from a client rectangle.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinCalcFrameRect  (HWND  hwndFrame, PRECTL  prcl, BOOL  fClient)
```

Parameters

hwndFrame (HWND) – input
Frame-window handle.

prcl (PRECTL) – in/out
Window rectangle.

fClient (BOOL) – input
Frame indicator.

TRUE	Frame rectangle provided
FALSE	Client-area rectangle provided.

Returns

rc (BOOL) – returns
Rectangle-calculated indicator.

TRUE	Rectangle successfully calculated
FALSE	Error occurred, or the calculated rectangle is empty.

WinCreateFrameControls

This function creates the standard frame controls for a specified window.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinCreateFrameControls  (HWND hwndFrame, PFRAMECDATA pfcdata,
                              PSZ pszTitle)
```

Parameters

hwndFrame (HWND) – input

Frame-window handle.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

pfcdata (PFRAMECDATA) – input

Frame-control data.

pszTitle (PSZ) – input

Title string.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Frame Windows.

WM_ADJUSTFRAMEPOS

This message is sent to a frame window whose position or size is to be adjusted.

Parameters

param1

pswp (PSWP)

New frame window state.

This points to a SWP structure.

The structure has been filled in by the WinSetWindowPos or WinSetMultWindowPos functions with the proposed move or size data for the frame window.

param2

hsavewphsvwp (HSAVEWP)

Identifier of the frame window repositioning process.

Returns

ulReserved (ULONG)

Reserved value, should be 0.

WM_ERASEBACKGROUND

This message causes a client window to be filled with the background, should this be appropriate.

Parameters

param1

hpsFrame (HPS)
Presentation-space handle for the frame window.

param2

pprcPaint (PRECTL)
Rectangle structure of rectangle to be painted.

This points to a RECTL structure.

Returns

rc (BOOL)
Processed indicator.

TRUE If a FID_CLIENT window exists, the area of the frame covered by the FID_CLIENT window is erased in the system-window background color.

If no FID_CLIENT window exists, the entire frame window is erased in the system-window background color.

FALSE The client window did process the message.

WM_FLASHWINDOW

An application has issued a WinFlashWindow function.

Parameters

param1

usFlash (USHORT)
Flash indicator.

TRUE	Start the window border flashing
FALSE	Stop the window border flashing.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_FORMATFRAME

This message is sent to a frame window to calculate the sizes and positions of all of the frame controls and the client window.

Parameters

param1

pswp (PSWP)
Structure array.

This points to an array that is to hold the SWP structures.

param2

pprectl (PRECTL)
Pointer to client window rectangle.

This is typically the window rectangle of *pswp*, but where the window has a wide border, as specified by FCF_DLGBORDER for example, the rectangle is inset by the size of the border.

Returns

ccount (USHORT)
Count of the number of SWP arrays returned.

WM_MINMAXFRAME

This message is sent to a frame window that is being minimized, maximized, or restored.

Parameters

param1

pswp (PSWP)
Set window position structure.

This points to a SWP structure. The structure has the appropriate SWP_* indicators set to describe the operation that is occurring to the window.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Processed indicator.

TRUE The message has been processed; the default system actions for the operation specified by the *pswp* parameter to the window are not to be performed.

FALSE The message has been ignored; the default system actions for the operation specified by the *pswp* parameter to the window are to be performed.

WM_NEXTMENU

This message occurs when either the beginning or the end of the menu is reached by use of the cursor control keys.

Parameters

param1

hwndMenu (HWND)
Menu-control window handle.

param2

usPrev (USHORT)
Previous-menu indicator.

TRUE	Beginning of the menu has been reached
FALSE	End of the menu has been reached.

Returns

hwndNewMenu (HWND)
New menu window handle.

NULLHANDLE	No new menu
Other	New menu window handle.

WM_QUERYFRAMECTLCOUNT

This message is sent to the frame window in response to the receipt of a WM_SIZE or a WM_UPDATEFRAME (in Frame Controls) message.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sControlCount (SHORT)
Count of frame controls.

WM_QUERYFRAMEINFO

This message enables an application to query information about frame windows.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

flFlags (ULONG)
Frame information flags.

FI_FRAME	Identifies a frame window.
FI_OWNERHIDE	The frame window is hidden when its owner is hidden.
FI_NOMOVEWITHOWNER	The frame window does not move with its owner.
FI_ACTIVATEOK	The frame window may be activated. This means, for example, that the frame window is not disabled.

WM_QUERYICON

This message is sent to a frame window to query its associated icon.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

hptrIcon (HPOINTER)
Handle to the icon.

WM_QUERYTRACKINFO

The frame control generates this message on receiving a WM_TRACKFRAME (in Frame Controls) message.

Parameters

param1

ustflags (USHORT)
Tracking flags.

Contains a combination of one or more TF_* flags as defined in the TRACKINFO structure.

param2

ptrackinfo (PTRACKINFO)
Track information structure.

This points to a TRACKINFO structure. The receiver of this message must modify this structure.

Returns

rc (BOOL)
Continue indicator.

TRUE	Continue sizing or moving
FALSE	Terminate sizing or moving.

WM_SETBORDERSIZE

This message is sent to the frame window to change the width and height of the border.

Parameters

param1

uscx (USHORT)
Width of border.

param2

uscy (USHORT)
Height of border.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_SETICON

This message is sent to a frame window to set its associated icon.

Parameters

param1

hptrIcon (HPOINTER)
New icon handle.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_SYSCOMMAND

This message occurs when a control window has a significant event to notify to its owner, or when a key stroke has been translated by an accelerator table into a WM_SYSCOMMAND.

Parameters

param1

uscmd (USHORT)
Command value.

The frame control takes the action described on these *uscmd* values:

SC_SIZE
Sends a WM_TRACKFRAME (in Frame Controls) to the frame window.

SC_MOVE
Sends a WM_TRACKFRAME (in Frame Controls) to the frame window.

SC_MINIMIZE
If a control with the identifier FID_MINMAX is present, minimizes the frame window, or restores it to a remembered size and position.

SC_MAXIMIZE
If a control with the identifier FID_MINMAX is present, maximizes the frame window, or restores it to a remembered size and position.

When a window is moved or sized in the normal way at least one border should remain on the screen. When a window is maximized and the maximum size is as large as the screen, all borders should be positioned just outside the screen.

SC_RESTORE
If a control with the identifier FID_MINMAX is present, restores a maximized frame window to its previous size and position.

SC_NEXT
Cycles the active window status to the next main window.

SC_APPMENU
Sends a MM_STARTMENUMODE message to the control with the identifier FID_MENU.

SC_SYSMENU
Sends a MM_STARTMENUMODE message to the control with the identifier FID_SYSMENU.

SC_CLOSE
If Close is not enabled in the system menu, this message is ignored. Otherwise the frame posts a WM_CLOSE message to the client if it exists or to itself, if not.

SC_NEXTFRAME
The next frame window that is a child of the desktop window is activated.

SC_NEXTWINDOW
The next window with the same owner window is activated.

SC_TASKMANAGER The Task List is activated.

SC_HELPEXTENDED The frame manager sends HM_EXT_HELP to the associated Help Manager Object Window. If there is no such associated window, the original message is sent to the client.

SC_HELPKEYS The frame manager sends HM_KEYS_HELP to the associated Help Manager Object Window. If there is no such associated window, the original message is sent to the client.

SC_HELPINDEX The frame manager sends HM_HELP_INDEX to the associated Help Manager Object Window. If there is no such associated window, the original message is sent to the client.

SC_HIDE Sets the visibility state of the frame window to off causing it to appear hidden or invisible.

param2

ussource (USHORT)
Source type.

Identifies the type of control:

CMDSRC_PUSHBUTTON Posted by a push-button control: *uscmd* is the window identifier of the push button.

CMDSRC_MENU Posted by a menu control: *uscmd* is the identifier of the menu item.

CMDSRC_ACCELERATOR Posted as the result of an accelerator: *uscmd* is the accelerator command value.

CMDSRC_OTHER Other source: *uscmd* gives further control-specific information defined for each control type.

fpointer (BOOL)
Pointing-device indicator.

TRUE The message is posted as a result of a pointing-device operation.
FALSE The message is posted as a result of a keyboard operation.

ulReserved (ULONG)
Reserved value, should be 0.

WM_TRACKFRAME

This message is sent to a window whenever it is to be moved or sized.

Parameters

param1

fsTrackFlags (USHORT)
Tracking flags.

Contains a combination of one or more TF_* flags; for details, see the TRACKINFO data structure description.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator

TRUE	The operation is successful.
FALSE	The operation is unsuccessful, or the operation is terminated.

WM_UPDATEFRAME

This message is sent by an application after frame controls have been added or removed from the window frame.

Parameters

param1

flCreateFlags (ULONG)

Frame-creation flags.

Contains the FCF_* flags that indicate which frame controls have been added or removed.

param2

ulReserved (ULONG)

Reserved value, should be 0.

Returns

rc (BOOL)

Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

Related Data Structures

This section covers the data structures that are related to Frame Windows.

FRAMECDATA

Frame-control data structure.

Syntax

```
typedef struct _FRAMECDATA {
USHORT      cb;
ULONG       flCreateFlags;
USHORT      hmodResources;
USHORT      idResources;
 } FRAMECDATA;

typedef FRAMECDATA *PFRAMECDATA;
```

Fields

cb (USHORT)

Length.

flCreateFlags (ULONG)

Frame-creation flags.

Possible values are described in the following list:

FCF_TITLEBAR
FCF_SYSMENU
FCF_MENU
FCF_SIZEBORDER
FCF_MINBUTTON
FCF_MAXBUTTON
FCF_MINMAX
FCF_VERTSCROLL
FCF_HORZSCROLL
FCF_DLGBORDER
FCF_BORDER
FCF_SHELLPOSITION
FCF_TASKLIST
FCF_NOBYTEALIGN
FCF_NOMOVEWITHOWNER
FCF_ICON
FCF_ACCELTABLE
FCF_SYSMODAL
FCF_SCREENALIGN
FCF_MOUSEALIGN

FCF_HIDEBUTTON

FCF_HIDEMAX

FCF_AUTOICON

FCF_DBE_APPSTAT

FCF_STANDARD — The standard setting is equivalent to setting FCF_TITLEBAR, FCF_SYSMENU, FCF_MENU, FCF_SIZEBORDER, FCF_MINMAX, FCF_ICON, FCF_ACCELTABLE, FCF_SHELLPOSITION, and FCF_TASKLIST.

hmodResources (USHORT)
Identifier of required resource.

This is supplied in an environment-dependent manner.

idResources (USHORT)
Resource identifier.

HSAVEWP

Frame window-repositioning process handle.

Syntax

```
typedef &handle. HSAVEWP;
```

Summary

Following are the OS/2 functions, messages, and structures used with frame windows.

Table 6-5. Frame-Window Functions

Function Name	Description
WinCalcFrameRect	Calculates a client rectangle from a frame rectangle or a frame rectangle from a client rectangle.
WinCreateFrameControls	Creates the standard frame controls for a specified window.

Table 6-6 (Page 1 of 2). Frame-Window Messages

Message	Description
WM_ACTIVATE	Occurs when an application causes the activation or deactivation of a window.
WM_ADJUSTFRAMEPOS	Sent by the WinSetWindowPos call to enable the window to adjust its new position or size whenever it is about to be moved.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON1UP	Occurs when the user releases point button 1.
WM_CALCVALIDRECTS	Sent to determine which areas of a window can be preserved and which can be displayed when a window is sized.
WM_CLOSE	Sent to a frame window to indicate that the user is closing the window.
WM_CREATE	Occurs when the application requests creation of a window.
WM_DESTROY	Occurs when an application requests destruction of a window.
WM_ENABLE	Sets the enable state of a window.
WM_ERASEBACKGROUND	Causes a client window to be filled with the background, if appropriate.
WM_FLASHWINDOW	Occurs when an application has issued a WinFlashWindow call.
WM_FOCUSCHANGE	Occurs when the window possessing the focus is changed.
WM_FORMATFRAME	Sent to a frame window to calculate the sizes and positions of all the frame controls and the client window.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.

Table 6-6 (Page 2 of 2). Frame-Window Messages

Message	Description
WM_MINMAXFRAME	Sent to a frame window that is being minimized, maximized, or restored.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_NEXTMENU	Occurs when either the beginning or the end of the menu is reached using the cursor control keys.
WM_PAINT	Occurs when a window needs painting.
WM_QUERYFRAMECTLCOUNT	Sent to the frame window in response to the receipt of a WM_SIZE or WM_UPDATEFRAME message.
WM_QUERYFRAMEINFO	Enables an application to query information about frame windows.
WM_QUERYICON	Sent to a frame window to query its associated icon.
WM_QUERYTRACKINFO	The frame control and title bar generate this message after receiving a WM_TRACKFRAME message.
WM_SETACCELTABLE	Establishes the window accelerator table to be used for translation when the window is active.
WM_SETBORDERSIZE	Sent to the frame window to change the width and height of the border.
WM_SETICON	Sent to a frame window to set its associated icon.
WM_SHOW	Occurs when a window's WS_VISIBLE state is changing.
WM_SIZECLIPBOARD	Sent when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and the clipboard application window has changed size.
WM_SYSCOMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_SYSCOMMAND message.
WM_TRACKFRAME	Sent to a window whenever it is to be moved or sized.
WM_TRANSLATEACCEL	Sent to the focus window whenever a WM_CHAR message occurs.
WM_UPDATEFRAME	Sent by an application after frame controls have been added or removed from the window frame.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Table 6-7. Frame-Window Structures

Structure Name	Description
FRAMECDATA	Frame-control data structure.
HSAVEWP	Frame window repositioning handle.

Chapter 7. Painting and Drawing

This chapter describes presentation spaces, device contexts, and window regions, explaining how a PM application uses them for painting and drawing in windows.

About Painting and Drawing

An application typically maintains an internal representation of the data that it is manipulating. The information displayed in a screen, window, or printed copy is a visual representation of some portion of that data. This chapter introduces the concepts and strategies necessary to make your PM application function smoothly and cooperatively in the OS/2 display environment.

Presentation Spaces and Device Contexts

A *presentation space* is a data structure, maintained by the operating system, that describes the drawing environment for an application. An application can create and hold several presentation spaces, each describing a different drawing environment. All drawing in a PM application must be directed to a presentation space.

Normally each presentation space is associated with a *device context* that describes the physical device where graphics commands are displayed. The device context translates graphics commands made to the presentation space into commands that enable the physical device to display information. Typical device contexts are the screen, printers and plotters, and off-screen memory bit maps.

Figure 7-1 shows how graphics commands from an application go through a presentation space, to a device context, and then to the physical device.

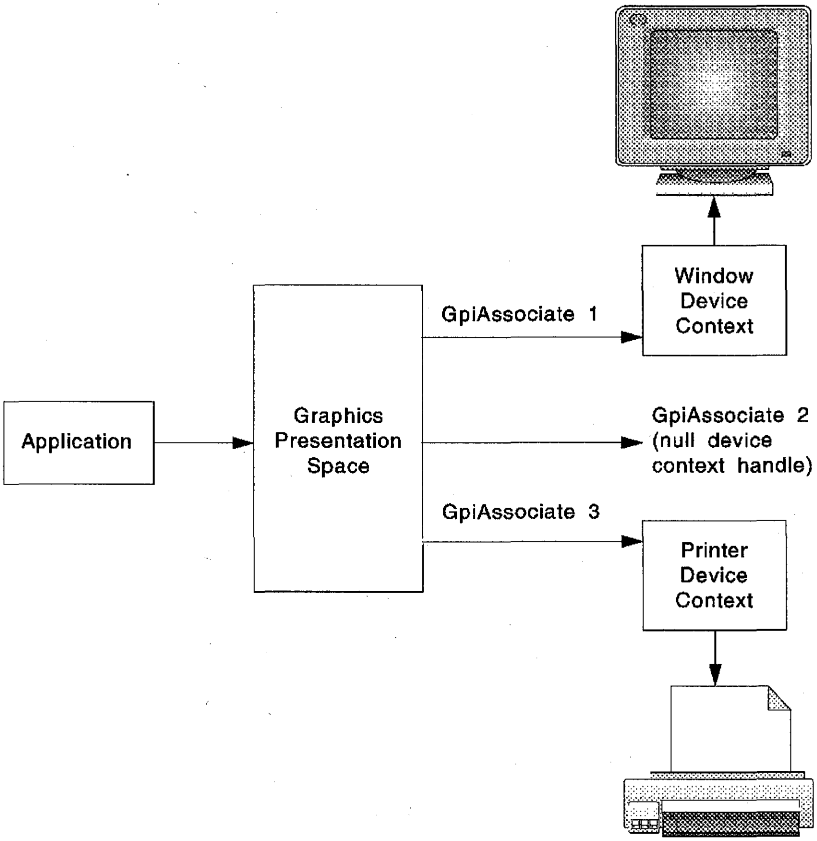


Figure 7-1. Application's Flow of Graphics Commands

By creating presentation spaces and associating them with particular device contexts, an application can control where its graphics output appears. Typically, a presentation space and device context isolate the application from the physical details of displaying graphics, so the same graphics commands can be used for many types of displays. This virtualization of output can reduce the amount of display code an application must include to support multiple output devices.

This chapter describes how an application sets up its presentation spaces and device contexts before drawing, and how to use window-drawing functions. Refer to the *Graphics Programming Interface Programming Guide* for the graphics functions available to PM applications.

Window Regions

A window and its associated presentation space have three regions that control where drawing takes place in the window. These regions ensure that the application does not draw outside the boundaries of the window or intrude into the space of an overlapping window.

Table 7-1. Window Regions

Region	Description
Update Region	This region represents the area of the window that needs to be redrawn. This region changes when overlapping windows change their z-order or when an application explicitly adds an area to the update region to force a window to be painted.
Clip Region	This region and the visible region determine where drawing takes place. Applications can change the clip region to limit drawing to a particular portion of a window. Typically, a presentation space is created with a clip region equal to NULL, which makes this region equivalent to the update region.
Visible Region	This region and the clip region determine where drawing takes place. The system changes the visible region to represent the portion of a window that is visible. Typically, the visible region is used to mask out overlapping windows. When an application calls the WinBeginPaint function in response to a WM_PAINT message, the system sets the visible region to the intersection of the visible region and the update region to produce a new visible region. Applications cannot change the visible region directly.

Whenever drawing occurs in a window's presentation space, the output is clipped to the intersection of the visible region and clip region. Figure 7-2 shows how the intersection of the visible region and the clip region of a window that is behind another window prevents the drawing in the back window from intruding into the front window.

The clip region includes the overlapped part of the back window, but the visible region excludes that portion of the back window. The system maintains the visible region to protect other windows on the screen; the application maintains the clip region to specify the portion of the window in which it draws. Together, these two regions provide safe and controllable clipping.

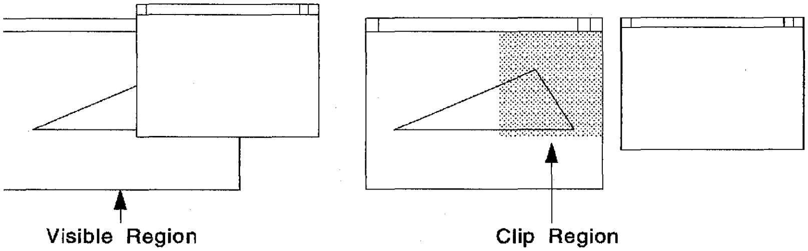


Figure 7-2. Clip Region and Visible Region of a Window's Presentation Space

To further control drawing, both the system and the application manipulate the update region. For example, if the windows shown in Figure 7-2 switch positions front to back, several changes occur in the regions of both windows. The system adds the lower-right corner of the new front window to that window's visible region. The system also adds that corner area to the window's update region.

Window Styles for Painting

Most of the styles relating to window drawing can be set either for the window class (CS_ prefix) or for an individual window (WS_ prefix). The styles described in this section control how the system manipulates the window's regions and how the window is notified when it must be painted or redrawn.

WS_CLIPCHILDREN, CS_CLIPCHILDREN

All the windows with this style are excluded from their parent's visible region. This style protects windows but increases the amount of time necessary to calculate the parent's visible region. This style normally is not necessary, because if the parent and child windows overlap and both are invalidated, the parent window is drawn before the child window. If the child window is invalidated independently from its parent window, only the child window is redrawn. If the update region of the parent window does not intersect the child window, drawing the parent window does not disturb the child window.

WS_CLIPSIBLINGS, CS_CLIPSIBLINGS

Windows with this style are excluded from the visible region of sibling windows. This style protects windows with the same parent from being drawn accidentally, but increases the amount of time necessary to calculate the visible region. This style is appropriate for sibling windows that overlap.

WS_PARENTCLIP, CS_PARENTCLIP

The visible region for a window with this style is the same as the visible region of the parent window. This style simplifies the calculation of the visible region but is potentially hazardous, because the parent window's visible region usually is larger than the child window. Windows with this style should not draw outside their boundaries.

WS_SAVEBITS, CS_SAVEBITS

The system saves the bits beneath a window with this style when the window is displayed. When the window moves or is hidden, the system simply restores the uncovered bits. This operation can consume a great deal of memory; it is recommended only for transient windows such as menus and dialog boxes—not for main application windows. This style also is inappropriate for windows that are updated dynamically, such as clocks.

WS_SYNCPAINT, CS_SYNCPAINT

Windows that have these styles receive WM_PAINT messages as soon as their update regions contain something; they are updated immediately (synchronously).

CS_SIZEREDRAW

A window with this class style receives a WM_PAINT message; the window is completely invalidated whenever it is resized, even if it is made smaller. (Typically, only the uncovered area of a window is invalidated when a window is resized.) This class style is useful when an application scales graphics to fill the current window.

Strategies for Painting and Drawing

A PM application shares the screen with other windows and applications; therefore, painting and drawing must not interfere with those other applications and windows. When you follow these strategies, your application can coexist with other applications and still take full advantage of the graphics capabilities of the operating system.

Drawing in a Window

Ideally, all drawing in a window occurs as a result of an application's processing a WM_PAINT message. Applications maintain an internal representation of what must be displayed in the window, such as text or a linked list of graphics objects, and use the WM_PAINT message as a cue to display a visual representation of that data in the window.

To route all display output through the WM_PAINT message, an application must not draw on the screen at the time its data changes. Instead, it must update the internal representation of the data and call the WinInvalidateRect or WinInvalidateRegion functions to invalidate the portion of the window that must be redrawn. Sometimes it is much more efficient to draw directly in a window without relying on the WM_PAINT message—for example, when drawing and redrawing an object for a user who is using the mouse to drag or size the object.

If a window has the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes a WM_PAINT message to be sent to the window immediately. Essentially, sending a message is like making a function call; the actions corresponding to the WM_PAINT message are carried out before the call that caused the invalidation returns—that is to say, the painting is synchronous.

If the window does not have the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes the invalidated region to be added to the window's update region. The next time the application calls the WinGetMsg or WinPeekMsg functions, the application is sent a WM_PAINT message. If there are many messages in the queue, the painting occurs after the invalidation—that is, the painting is asynchronous. A WM_PAINT message is not posted to the queue in this case, so all invalidation operations since the last WM_PAINT message are consolidated into a single WM_PAINT message the next time the application has no messages in the queue.

There are advantages to both synchronous and asynchronous painting. Windows that have simple painting functions should be painted synchronously. Most of the system-defined control windows, such as buttons and frame controls, are painted synchronously because they can be painted quickly without interfering with the responsiveness of the program. Windows that require more time-consuming painting operations should be painted asynchronously so that the painting can be initiated only when there are no other pending messages that might otherwise be blocked while waiting for the window to be painted. Also, a window that uses an incremental approach to invalidating small portions of itself usually should allow those operations to consolidate into a single asynchronous WM_PAINT message, rather than a series of synchronous WM_PAINT messages.

If necessary, an application can call the WinUpdateWindow function to cause an asynchronous window to update itself without going through the event loop. WinUpdateWindow sends a WM_PAINT message directly to the window if the window's update region is not empty.

The WM_PAINT Message

A window receives a WM_PAINT message whenever its update region is not NULL. A window procedure responds to a WM_PAINT message by calling the WinBeginPaint function, drawing to fill in the update areas, then calling the WinEndPaint function.

The WinBeginPaint function returns a handle to a presentation space that is associated with the device context for the window and that has a visible region equal to the intersection of the window's update region and its visible region. This means that only those portions of the window that need to be redrawn are drawn. Attempts to draw outside this region are clipped and do not appear on the screen.

If the application maintains its own presentation space for the window, it can pass the handle of that presentation space to WinBeginPaint, which modifies the visible region of the presentation space and passes the presentation-space handle back to the caller. If the application does not have its own presentation space, it can pass a NULL presentation-space handle and the system will return a cached-micro presentation space for the window. In either case, the application can use the presentation space to draw in the window.

The WinBeginPaint function takes a pointer to a RECTL structure, filling in this structure with the coordinates of the rectangle that encloses the area to be updated. The application can use this rectangle to optimize drawing, by drawing only those portions of the window that intersect with the rectangle. If an application passes a NULL pointer for the rectangle argument, the application draws the entire window and relies on the clipping mechanism to filter out the unneeded areas.

After the WinBeginPaint function sets the update region of a window to NULL, the application does the necessary drawing to fill the update areas. If an application handles a WM_PAINT message and does not call WinBeginPaint, or otherwise empty the update region, the application continues to receive WM_PAINT messages as long as the update region is not empty.

After the application finishes drawing, it calls the WinEndPaint function to restore the presentation space to its former state. When a cached-micro presentation space is returned by WinBeginPaint, the presentation space is returned to the system for reuse. If the application supplies its own presentation space to WinBeginPaint, the presentation space is restored to its previous state.

Drawing the Minimized View

When an application creates a standard frame window, it has the option of specifying an icon that the system uses to represent the application in its minimized state. Typically, if an icon is supplied, the system draws it in the minimized window and labels it with the name of the window. If the application does not specify the FS_ICON style for the window, the window receives a WM_PAINT message when it is minimized. The code in the window procedure that handles the WM_PAINT message can determine whether the frame window currently is minimized and draw accordingly. Notice that because the WS_MINIMIZED style is relevant only for the frame window, and not for the client window, the window procedure checks the frame window rather than the client window.

The following code fragment shows how to draw a window in both the minimized and normal states:

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2)
{
    HPS hps;
    RECTL rcl;
    ULONG flStyle;

    switch (msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd, (HPS) NULL, &rcl);

                     /* Check whether the frame window
                      (client's parent window)
                       is minimized.            */

            flStyle = WinQueryWindowULong(WinQueryWindow(hwnd,
                QW_PARENT), QWL_STYLE);

            if (flStyle & WS_MINIMIZED) {
                .
                .    /* Paint the minimized state. */
                .
            }
            else {
                .
                .    /* Paint the normal state.    */
                .
            }
            WinEndPaint(hps);
            return 0;
    }
}
```

Drawing Without the WM_PAINT Message

An application can draw in a window's presentation space without having received a WM_PAINT message. As long as there is a presentation space for the window, an application can draw into the presentation space and avoid intruding into other windows or the desktop. Applications that draw without using the WM_PAINT message typically call the WinGetPS function to obtain a cached-micro presentation space for the window and call the WinReleasePS function when they have finished drawing. An application also can use any of the other types of presentation spaces described in the following sections.

Three Types of Presentation Spaces

All drawing must take place within a presentation space.

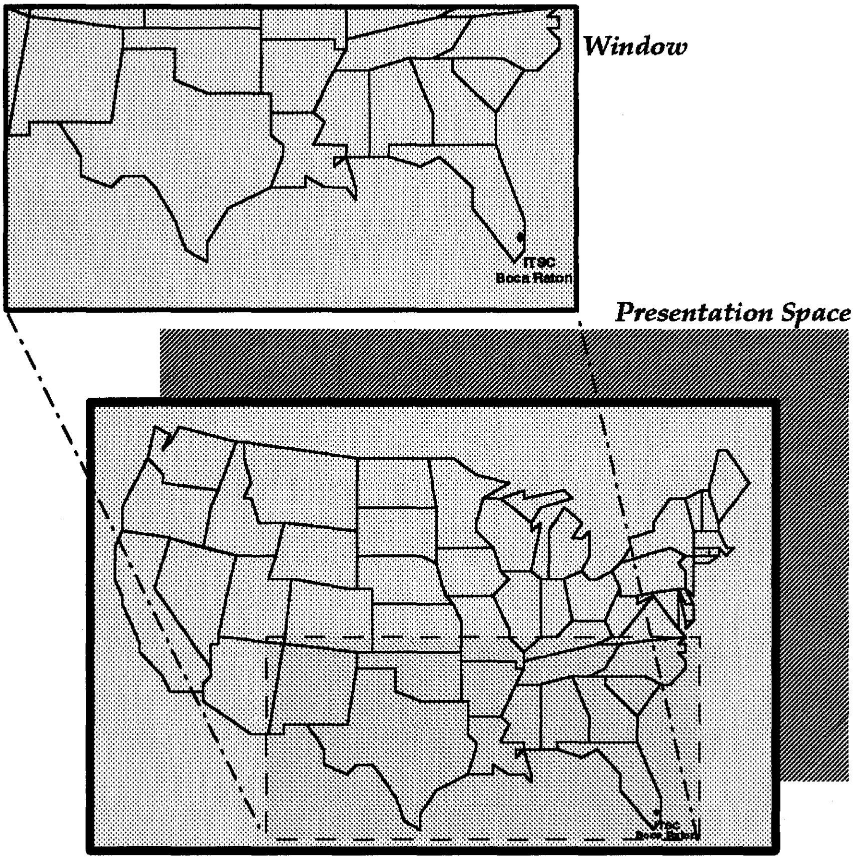


Figure 7-3. Presentation Space versus Window

The operating system provides three types of presentation spaces for drawing: normal, micro, and cached-micro presentation spaces.

The *normal presentation space* provides the most functionality, allowing access to all the graphics functions of the operating system and enabling the application to draw to all device types. The normal presentation space is more difficult to use than the other two kinds of presentation spaces and it uses more memory. It is created by using the GpiCreatePS function and is destroyed by using the GpiDestroyPS function.

The *micro presentation space* allows access to only a subset of the operating system graphics functions, but it uses less memory and is faster than a normal presentation space. The micro presentation space also enables the application to draw to all device types. It is created by using the GpiCreatePS function and destroyed by using the GpiDestroyPS function.

The *cached-micro presentation space* provides the least functionality of the three kinds of presentation spaces, but it is the most efficient and easiest to use. The cached-micro presentation space draws only to the screen. It is created and destroyed by using either the WinBeginPaint and WinEndPaint functions or the WinGetPS and WinReleasePS functions.

The following sections describe each of the types of presentation spaces, in detail, and suggest strategies for using each type in an application. All three kinds of presentation spaces can be used in a single application. Some windows, especially if they never will be printed, are best served by cached-micro presentation spaces. Other windows might require the more flexible services of micro or normal presentation spaces.

Normal Presentation Spaces

The normal presentation space supports the full power of the operating system graphics, including retained graphics. The primary advantages of a normal presentation space over the other two presentation-space types are its support of all graphics functions and its ability to be associated with many kinds of device contexts.

A normal presentation space can be associated with many different device contexts. Typically, this means that an application creates a normal presentation space and associates it with a window device context for screen display. When the user asks to print, the application associates the same presentation space with a printer device context. Later, the application can reassociate the presentation space with the window device context. A presentation space can be associated with only one device context at a time, but the normal presentation space enables the application to change the device context whenever necessary.

Figure 7-4 shows how an application typically routes graphics through one normal presentation space into another device context:

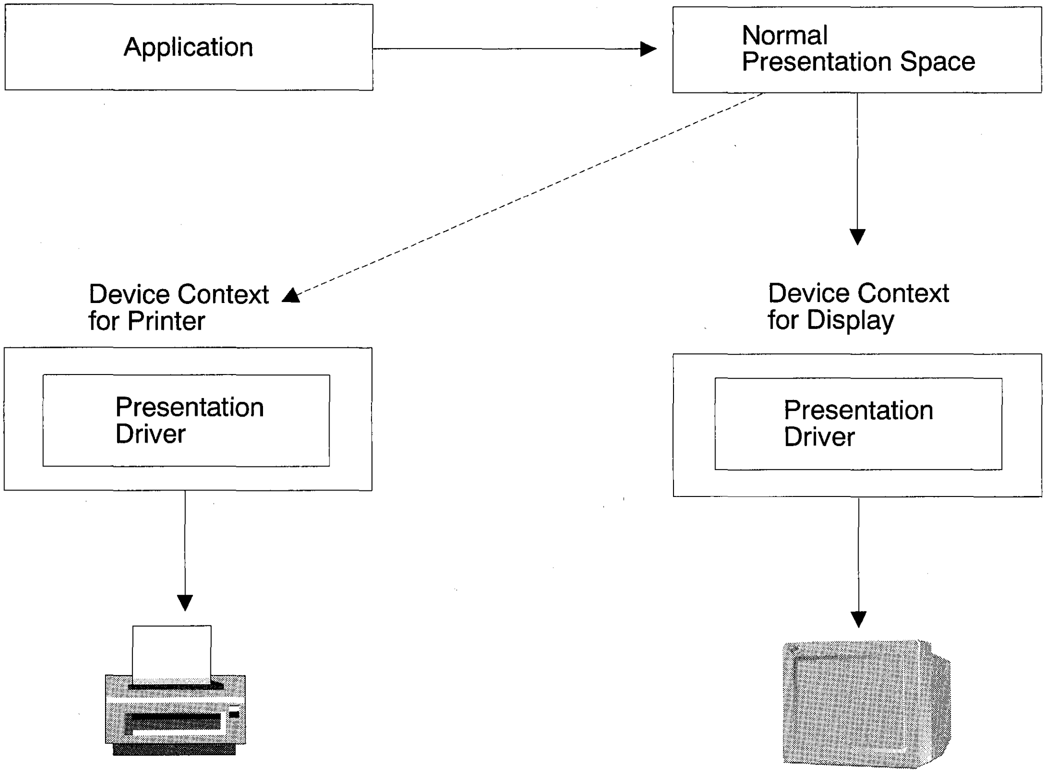

Application
Normal Presentation Space
Device Context for Printer
Device Context for Display
Presentation Driver
Presentation Driver

Figure 7-4. Normal Presentation Space

When creating a normal presentation space, an application can associate it with a device context or defer the association to a later time. The GpiAssociate function associates a device context with a normal presentation space after the presentation space has been created. An application typically associates the normal presentation space with a device context when calling the GpiCreatePS function and, later, associates the presentation space with a different device context by calling GpiAssociate. To obtain a device context for a window, call the WinOpenWindowDC function. To obtain a device context for a device other than the screen, call the DevOpenDC function.

An application typically creates a normal presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it passes the handle of the normal presentation space as an argument to WinBeginPaint; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied normal presentation space and returns the presentation space to the application. This method enables the application to use the same presentation space for all the drawing in a specified window.

Normal presentation spaces created using GpiCreatePS must be destroyed by calling GpiDestroyPS before the application terminates. Do not call WinReleasePS to release a presentation space obtained using GpiCreatePS. Before terminating, applications also must use DevCloseDC to close any device contexts opened using DevOpenDC. No action is necessary for device contexts obtained using WinOpenWindowDC, because the system automatically closes these device contexts when destroying the associated windows.

Micro Presentation Spaces

The primary advantage of a micro presentation space over a cached-micro presentation space is that it can be used for printing as well as painting in a window. An application that uses a micro presentation space must explicitly associate it with a device context. This makes the micro presentation space useful for painting to a printer, a plotter, or an off-screen memory bit map.

A micro presentation space does not support the full set of OS/2 graphics functions. Unlike a normal presentation space, a micro presentation space does not support retained graphics.

An application that must display graphics or text in a window and print to a printer or plotter typically maintains two presentation spaces: one for the window and one for the printing device. Figure 7-5 on page 7-13 shows how an application's graphics output can be routed through separate presentation spaces to produce a screen display and printed copy.

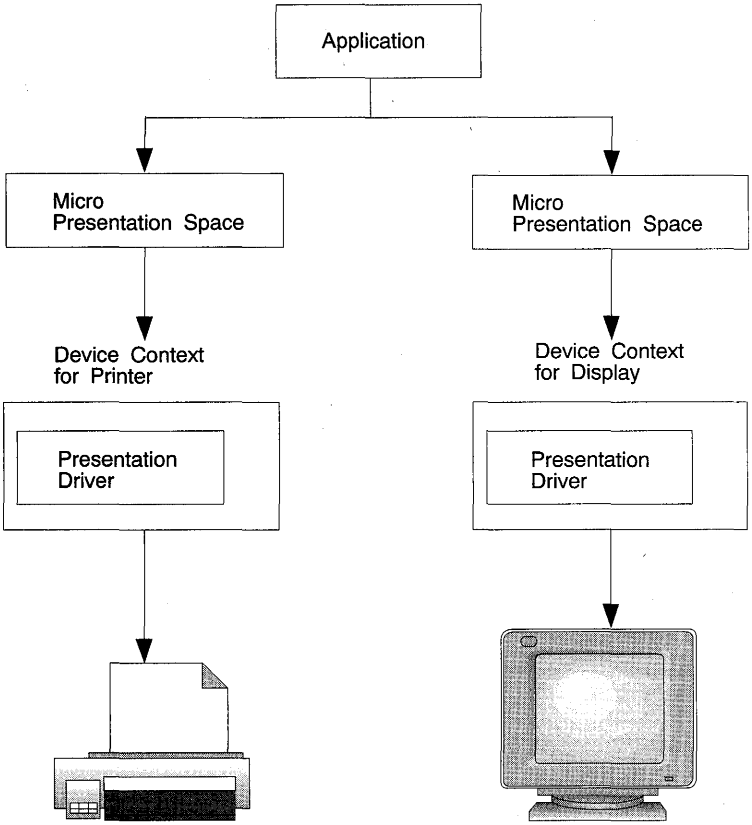


Figure 7-5. Micro Presentation Space

An application creates a micro presentation space by calling the GpiCreatePS function. A device context must be supplied at the time the micro presentation space is created. An application typically creates a device context and then a presentation space. The following code fragment demonstrates this by obtaining a device context for a window and associating it with a new micro presentation space:

```
    hdc = WinOpenWindowDC(...);
    hps = GpiCreatePS(..., hdc, ..., GPIA_ASSOC);
```

To create a micro presentation space for a device other than the screen, replace the call to the WinOpenWindowDC function with a call to the DevOpenDC function, which obtains a device context for a device other than the screen. Then the device context that is obtained by this call can be used as an argument to GpiCreatePS.

An application typically creates a micro presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it should pass the handle of the micro presentation space as an argument to the WinBeginPaint function; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied micro presentation space and returns the presentation space to the application. This method enables the application to use the same presentation space for all drawing in a specified window.

Micro presentation spaces created by using GpiCreatePS should be destroyed by calling GpiDestroyPS before the application terminates. Do not call the WinReleasePS function to release a presentation space obtained by using GpiCreatePS. Before terminating, applications must use the DevCloseDC function to close any device contexts opened using the DevOpenDC function. No action is necessary for device contexts obtained using WinOpenWindowDC, because the system automatically closes these device contexts when destroying the associated windows.

Cached-Micro Presentation Spaces

The cached-micro presentation space provides the simplest and most efficient drawing environment. It can be used only for drawing on the screen, typically in the context of a window. It is most appropriate for application tasks that require simple window-drawing functions that will not be printed. Cached-micro presentation spaces do not support retained graphics.

After an application draws to a cached-micro presentation space, the drawing commands are routed through an implied device context to the current display. The application does not require information about the actual device context, because the device context is assumed to be the display. This process makes cached-micro presentation spaces easy for applications to use.

The following code fragment illustrates this process:

```
HPS    hps;

     case WM_PAINT:
         hps = WinBeginPaint(hwnd,NULL,NULL);

         /*
          * Use PS.
          */

         WinEndPaint (hps);
```

or

```
 HPS    hps;

     case WM_PAINT:

          hps = WinGetPS(hwnd);

          /*
          * Use PS.
          */

          WinReleasePS(hps);
```

There are two common strategies for using cached-micro presentation spaces in an application. The simplest strategy is to call the WinBeginPaint function during the WM_PAINT message, use the resulting cached-micro presentation space to draw in the window, then return the presentation space to the system by calling the WinEndPaint function. By using this method, the application interacts with the presentation space only when drawing in the presentation space. This method is most appropriate for simple drawing. A disadvantage of this method is that the application must set up any special attributes for the presentation space, such as line color and font, each time a new presentation space is obtained.

A second strategy is for the application to allocate a cached-micro presentation space during initialization, by calling the WinGetPS function and saving the resulting presentation-space handle in a static variable. Then the application can set attributes in the presentation space that exist for the life of the program. The presentation-space handle can be used as an argument to the WinBeginPaint function each time the window gets a WM_PAINT message; the system modifies the visible region and returns the presentation space to the application with its attributes intact. This strategy is appropriate for applications that need to customize their window-drawing attributes.

A presentation space that is obtained by calling the WinGetPS function must be released by calling WinReleasePS when the application has finished using it, typically during program termination. A presentation space that is obtained by calling WinBeginPaint must be released by calling WinEndPaint, typically as the last part of processing a WM_PAINT message.

Related Functions

This section covers the functions that are related to Painting and Drawing.

WinBeginPaint

This function obtains a presentation space whose associated update region is set ready for drawing in a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
HPS WinBeginPaint  (HWND hwnd, HPS hps, PRECTL prclPaint)
```

Parameters

hwnd (HWND) – input

Handle of window where drawing is going to occur.

HWND_DESKTOP	The desk top window.
Other	Specified window.

hps (HPS) – input

Presentation-space handle.

NULLHANDLE	Obtain a cache presentation space.
Other	Presentation-space handle. This function sets its clipping region to the update region of the *hwnd* parameter.

prclPaint (PRECTL) – output

Bounding rectangle.

NULL	No bounding rectangle; that is, there is no need of changing any point.
Other	Specifies the smallest rectangle bounding the update region, in window coordinates.

Returns

hpsPaintPS (HPS) – returns

Presentation-space handle.

NULLHANDLE	Error occurred
Other	Presentation-space handle.

WinEndPaint

This function indicates that the redrawing of a window is complete, generally as part of the processing of a WM_PAINT message.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinEndPaint  (HPS  hps)
```

Parameters

hps (HPS) – input
Presentation-space handle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinExcludeUpdateRegion

This function subtracts the update region (invalid region) of a window from the clipping region of a presentation space.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinExcludeUpdateRegion  (HPS hps, HWND hwnd)
```

Parameters

hps (HPS) – input
Presentation-space handle whose clipping region is to be updated.

hwnd (HWND) – input
Window handle.

Returns

lComplexity (LONG) – returns
Complexity value.

RGN_NULL	Null Region
RGN_RECT	Rectangle region
RGN_COMPLEX	Complex region
RGN_ERROR	Error.

WinGetClipPS

This function obtains a clipped cache presentation space.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

HPS WinGetClipPS  (HWND hwnd, HWND hwndClipWindow, ULONG ulClipflags)
```

Parameters

hwnd (HWND) – input

Handle of window for which the presentation space is required.

hwndClipWindow (HWND) – input

Handle of window for clipping.

HWND_BOTTOM — Clip the last window in the sibling chain and continue clipping until the next window is *hwnd* or NULLHANDLE.

HWND_TOP — Clip the first window in the sibling chain and continue clipping until the next window is *hwnd* or NULLHANDLE.

NULLHANDLE — Clip all siblings to the window *hwnd*.

ulClipflags (ULONG) – input

Clipping control flags.

PSF_CLIPSIBLINGS — Clip out all siblings of *hwnd*.

PSF_CLIPCHILDREN — Clip out all children of *hwnd*.

PSF_CLIPUPWARDS — Taking *hwndClipWindow* as a reference window, clip out all sibling windows before *hwndClipWindow*. This value may not be used with PSF_CLIPDOWNWARDS.

PSF_CLIPDOWNWARDS — Taking *hwndClipWindow* as a reference window, clip out all sibling windows after *hwndClipWindow*. This value may not be used with PSF_CLIPUPWARDS.

PSF_LOCKWINDOWUPDATE — Calculate a presentation space that keeps a visible region even though output may be locked by the WinLockWindowUpdate function.

PSF_PARENTCLIP — Calculate a presentation space that uses the visible region of the parent of *hwnd* but with an origin calculated for *hwnd*.

Returns

hps (HPS) – returns

Presentation-space handle that can be used for drawing.

WinGetPS

This function gets a cache presentation space.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
```

HPS WinGetPS (HWND hwnd)

Parameters

hwnd (HWND) – input

Handle of window for which the presentation space is required.

HWND_DESKTOP	The desktop-window handle; a presentation space for the whole of the desktop window is returned
Other	Handle of window for which the presentation space is required.

Returns

hps (HPS) – returns

Presentation-space handle that can be used for drawing in the window.

WinGetScreenPS

This function returns a presentation space that can be used for drawing anywhere on the screen.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HPS WinGetScreenPS  (HWND  hwndDeskTop)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hpsScreenPS (HPS) – returns

Presentation-space handle.

NULLHANDLE	*hwndDeskTop* is not HWND_DESKTOP or a desktop window handle obtained from the WinQueryDesktopWindow function.
Other	Presentation space handle.

WinInvalidateRect

This function adds a rectangle to a window's update region.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinInvalidateRect  (HWND hwnd, PRECTL pwrc, BOOL fIncludeChildren)
```

Parameters

hwnd (HWND) – input
Handle of window whose update region is to be changed.

HWND_DESKTOP	This function applies to the whole screen (or desktop).
Other	Handle of window whose update region is to be changed.

pwrc (PRECTL) – input
Update rectangle.

NULL	The whole window is to be added into the window's update region.
Other	Rectangle to be added to the window's update region.

fIncludeChildren (BOOL) – input
Invalidation-scope indicator.

TRUE	Include the descendants of *hwnd* in the invalid rectangle.
FALSE	Include the descendants of *hwnd* in the invalid rectangle, but only if the parent does not have a WS_CLIPCHILDREN style.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinInvalidateRegion

This function adds a region to a window's update region.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

BOOL WinInvalidateRegion (HWND hwnd, HRGN hrgn, BOOL fIncludeChildren)

Parameters

hwnd (HWND) – input

Handle of window whose update region is to be changed.

HWND_DESKTOP This function applies to the whole screen (or desktop).
Other Handle of window whose update region is to be changed.

hrgn (HRGN) – input

Handle of the region to be added to the update region of the window.

NULLHANDLE The whole window is to be added into the window's update region.
Other Handle of the region to be added to the window's update region.

fIncludeChildren (BOOL) – input

Invalidation-scope indicator.

TRUE Include the descendants of *hwnd* in the invalid rectangle.
FALSE Include the descendants of *hwnd* in the invalid rectangle, but only if the parent does not have a WS_CLIPCHILDREN style.

Returns

rc (BOOL) – returns

Success indicator.

TRUE Successful completion
FALSE Error occurred.

WinLockVisRegions

This function locks or unlocks the visible regions of all the windows on the screen, preventing any of the visible regions from changing.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinLockVisRegions  (HWND hwndDesktop, BOOL fLock)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle or HWND_DESKTOP.

fLock (BOOL) – input
Indicates whether the visible regions are being locked or unlocked.

TRUE	Lock the visible regions
FALSE	Unlock the visible regions.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful.
FALSE	An error occurred.

WinLockWindowUpdate

This function disables or enables output to a window and its descendants.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinLockWindowUpdate  (HWND hwndDeskTop, HWND hwndLockUpdate)
```

Parameters

hwndDeskTop (HWND) – input

Desktop handle of the screen containing the window to be locked.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

hwndLockUpdate (HWND) – input

Handle of window in which output is to be prevented.

NULLHANDLE	Enable output in the locked window and its descendants.
Other	Handle of the window in which output is to be prevented. Output is also prevented in the descendants of the window.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful operation.
FALSE	Error occurred.

WinOpenWindowDC

This function opens a device context for a window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

HDC WinOpenWindowDC  (HWND  hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

hdc (HDC) – returns
Device-context handle.

WinQueryUpdateRect

This function returns the rectangle that bounds the update region of a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryUpdateRect  (HWND hwnd, PRECTL prclPrc)
```

Parameters

hwnd (HWND) – input

Handle of window whose update rectangle is to be queried.

prclPrc (PRECTL) – output

Update region that bounds the rectangle (in window coordinates).

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred, or window has no update region; it is wholly valid, therefore *prclPrc* is NULL.

WinQueryUpdateRegion

This call obtains an update region of a window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinQueryUpdateRegion  (HWND  hwnd, HRGN  hrgn)
```

Parameters

hwnd (HWND) – input

Handle of window whose update region is to be queried.

hrgn (HRGN) – input

Handle of the window's update region.

Returns

lComplexity (LONG) – returns

Complexity of resulting region/error indicator.

RGN_NULL	Null region
RGN_RECT	Rectangular region
RGN_COMPLEX	Complex region
RGN_ERROR	Error.

WinQueryWindowDC

This function returns the device context for a given window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HDC WinQueryWindowDC  (HWND  hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

hdc (HDC) – returns
Device-context handle.

NULLHANDLE   Either WinOpenWindowDC has not been called for this window, or an error has occurred.

Other   Device context handle.

WinReleasePS

This function releases a cache presentation space obtained using the WinGetPS, the WinGetScreenPS, or the WinGetClipPS call.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinReleasePS  (HPS hps)
```

Parameters

hps (HPS) – input
Handle of the cache presentation space to release, as returned by the WinGetPS, the WinGetScreenPS, or the WinGetClipPS function.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinUpdateWindow

This function forces the update of a window and its associated child windows.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinUpdateWindow  (HWND  hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

rc (BOOL) – returns
Window-updated indicator.

TRUE	Window successfully updated
FALSE	Window not successfully updated.

WinValidateRect

This function subtracts a rectangle from the update region of an asynchronous paint window, marking that part of the window as visually valid.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinValidateRect  (HWND hwnd, PRECTL prclRect,
                       BOOL fIncludeClippedChildren)
```

Parameters

hwnd (HWND) – input
Handle of window whose update region is changed.

prclRect (PRECTL) – input
Rectangle to be subtracted from the window's update region.

fIncludeClippedChildren (BOOL) – input
Validation-scope indicator.

TRUE	Include descendants of *hwnd* in the valid rectangle
FALSE	Include descendants of *hwnd* in the valid rectangle, only if parent is not WS_CLIPCHILDREN.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinValidateRegion

This function subtracts a region from the update region of an asynchronous paint window, marking that part of the window as visually valid.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinValidateRegion  (HWND hwnd, HRGN hrgn,
                         BOOL fIncludeClippedChildren)
```

Parameters

hwnd (HWND) – input
Handle of window whose update region is changed.

hrgn (HRGN) – input
Handle of subtracted region.

fIncludeClippedChildren (BOOL) – input
Validation-scope indicator.

TRUE	Include descendants of *hwnd* in the valid region
FALSE	Include descendants of *hwnd* in the valid region, only if parent is not WS_CLIPCHILDREN.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinWindowFromDC

This function returns the handle of the window corresponding to a particular device context.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinWindowFromDC  (HDC hdc)
```

Parameters

hdc (HDC) – input
Device-context handle.

Returns

hwnd (HWND) – returns
Window handle.

NULLHANDLE Error occurred. For example, the device context has not been opened by the WinOpenWindowDC function.

Other Window handle.

Related Messages

This section covers the messages that are related to Drawing and Painting.

WM_PAINT

This message occurs when a window needs repainting.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Painting and Drawing.

RECTL

Rectangle structure.

Syntax

```
typedef struct _RECTL {
LONG       xLeft;
LONG       yBottom;
LONG       xRight;
LONG       yTop;
  } RECTL;

typedef RECTL *PRECTL;
```

Fields

xLeft (LONG)
X-coordinate of left-hand edge of rectangle.

yBottom (LONG)
Y-coordinate of bottom edge of rectangle.

xRight (LONG)
X-coordinate of right-hand edge of rectangle.

yTop (LONG)
Y-coordinate of top edge of rectangle.

Summary

Following are the OS/2 functions used with presentation spaces, device contexts, and window regions.

Table 7-2 (Page 1 of 2). Presentation Space, Device Context, and Window Region Functions

Function Name	Description
DevCloseDC	Closes a device context.
DevOpenDC	Opens a device context.
GpiAssociate	Associates a graphics presentation space with, or disassociates it from, a device context.
GpiCreatePs	Creates a presentation space
GpiDestroyPS	Destroys a presentation space.
WinBeginPaint	Obtains a presentation space whose associated update region is set to draw in a specified window.
WinEnableWindowUpdate	Sets the visibility state for subsequent drawing.
WinEndPaint	Indicates that the redrawing of a window is complete.
WinExcludeUpdateRegion	Subtracts the update region of a window from the clipping region of a presentation space.
WinGetClipPS	Obtains a clipped cache presentation space.
WinGetPS	Gets a cache presentation space.
WinGetScreenPS	Returns a presentation space that can be used for drawing anywhere on the screen.
WinInvalidateRect	Adds a rectangle to a window's update region.
WinInvalidateRegion	.Adds a region to a window's update region.
WinLockVisRegions	Locks or unlocks the visible regions or all the windows
WinLockWindowUpdate	Disables or enables output to a window and its descendants.
WinOpenWindowDC	Opens a device context for a window.
WinQueryUpdateRect	Returns the rectangle that bounds the update region of a specified window.
WinQueryUpdateRegion	Obtains an update region of a window.
WinQueryWindowDC	Returns the device context for a given window.
WinReleasePS	Releases a cache presentation space obtained using the WinGetPS or WinGetScreenPS calls.
WinUpdateWindow	Forces the update of a window and its associated child windows.
WinValidateRect	Subtracts a rectangle from the update region of an asynchronous paint window, marking that part of the window as visually valid.

Table 7-2 (Page 2 of 2). Presentation Space, Device Context, and Window Region Functions

Function Name	Description
WinValidateRegion	Subtracts a region from the update region of an asynchronous paint window, marking that part of the widow as visually valid.
WinWindowFromDC	Returns the handle of the window corresponding to a particular device context.

Table 7-3. Presentation Space, Device Context, and Window Region Messages

Message	Description
WM_PAINT	Sent when a window needs repainting.

Table 7-4. Presentation Space, Device Context, and Window Region Structures

Structure name	Description
RECTL	Rectangle structure.

Chapter 8. Drawing in Windows

This chapter describes, at a high level, the functions specifically intended for drawing in PM windows. For information on the complete set of drawing functions, see the *Graphics Programming Interface Programming Guide*.

About Window-Drawing Functions

The functionality of the PM window-drawing functions overlaps that of similar Graphic Programming Interface (GPI) drawing functions in OS/2. These window-drawing functions are less general than the GPI functions and are somewhat easier to use, but they also offer fewer capabilities than the complete set of GPI functions. Programmers requiring optimum functionality should use the GPI functions.

Points

All drawing in a window takes place in the context of the window's coordinate system. Locations of points in the window are described by POINTL structures, which contain an x and a y coordinate for the point. The lower-left corner of a window always has the coordinates (0,0).

The WinMapWindowPoints function converts the coordinates of points from one window-coordinate space to those of another window-coordinate space. If one of the specified windows is HWND_DESKTOP, the function uses screen coordinates. This function is useful for converting window coordinates to screen coordinates or the other way around.

Rectangles

Locations of window rectangles are described by RECTL structures, which contain the coordinates of two points that define the lower-left and upper-right corners of the rectangle. An empty rectangle is one that has no area: either its right coordinate is less than or equal to its left coordinate, or its top coordinate is less than or equal to its bottom coordinate.

There are two types of rectangles in OS/2: inclusive-inclusive and inclusive-exclusive. In inclusive-exclusive rectangles, the lower-left coordinate of the rectangle is included within the rectangle area, while the upper-right coordinate is excluded from the rectangle area. In an inclusive-inclusive rectangle, both the lower-left and upper-right coordinates are included in the rectangle. Figure 8-1 on page 8-2 shows both types of rectangles:

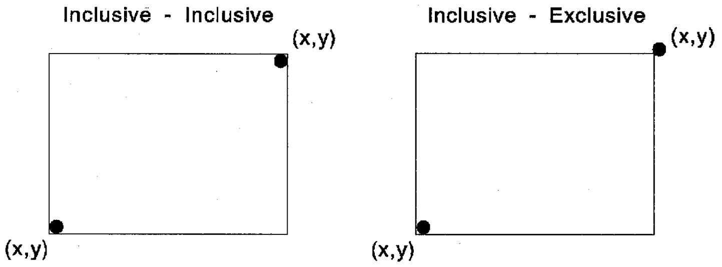


Figure 8-1. Types of Rectangles

In general, graphics operations involving device coordinates (such as regions, bit maps and bit blts, and window management) use inclusive-exclusive rectangles. All other graphics operations, such as GPI functions that define paths, use inclusive-inclusive rectangles.

Using Window-Drawing Functions

This section explains how to use drawing functions to fill (paint) a rectangle with color, scroll the contents of a window, draw bit maps and text, and determine the dimensions of a rectangle.

Working with Points and Rectangles

The operating system includes functions for manipulating rectangles, many of which change the rectangle coordinates. Other functions draw in a presentation space, using a rectangle to position the drawing operation.

The rest of the rectangle functions are mathematical and do not draw. They are used to manipulate and combine rectangles to produce new rectangles that you then can use in drawing operations.

Determining the Dimensions of a Rectangle

You can calculate the dimensions of an inclusive-exclusive rectangle as follows:

```
cx = rcl.xRight - rcl.xLeft; /* width  */
cy = rcl.yTop - rcl.yBottom; /* height */
```

You can calculate the dimensions of an inclusive-inclusive rectangle as follows:

```
cx = (rcl.xRight - rcl.xLeft) + 1; /* width  */
cy = (rcl.yTop - rcl.yBottom) + 1; /* height */
```

Filling a Rectangle

The WinFillRect function fills (paints) a rectangle with a specified color. For example, to fill an entire window with blue in response to a WM_PAINT message, you could use the following code fragment, which is taken from a window procedure:

```
HPS    hps;
RECTL  rcl;


case WM_PAINT:
    hps = WinBeginPaint(hwnd, (HPS) NULL, (PRECTL) NULL);
    WinQueryWindowRect(hwnd, &rcl);
    WinFillRect(hps, &rcl, CLR_BLUE);
    WinEndPaint(hps);
    return 0;
```

A more efficient way of painting a client window is to pass a rectangle to the WinBeginPaint function. The rectangle is set to the coordinates of the rectangle that encloses the update region of the window. Drawing in this rectangle updates the window, which can make drawing faster if only a small portion of the window needs to be painted. This method is shown in the following code fragment. Notice that WinFillRect uses the presentation space and a rectangle defined in window coordinates to guide the paint operation.

```
HPS    hps;
RECTL  rcl;

case WM_PAINT:
    hps = WinBeginPaint(hwnd, (HPS) NULL, &rcl);
    WinFillRect(hps, &rcl, CLR_BLUE);
    WinEndPaint(hps);
    return 0;
```

You could draw the entire window during the WM_PAINT message, but the graphics output would be clipped to the update region.

The default method of indicating that a particular portion of a window has been selected is using the WinInvertRect function to invert the rectangle's bits.

Scrolling the Contents of a Window

An application typically responds to a click in a scroll bar by scrolling the contents of the window. This operation has three parts. First, the application changes its internal data-representation state to show what portion of the image must now be in the window. Next, the application moves the current image in the window. Finally, the application draws in the area that has been uncovered by the scrolling operation.

For example, a simple text editor might display a small portion of several pages of text in a window. When the user clicks the Down arrow of the vertical scroll bar, the application moves all the text up one line and displays the next line at the bottom of the window.

This clicking also causes a message to be sent to the client window of the frame window that owns the scroll bar. The application responds to this message by changing its internal data-representation state to show which line of text is topmost in the window, scrolling the text in the window up one line, and drawing the new line at the bottom of the window. There normally is no need to completely redraw the entire window, because the scrolled portion of the image remains valid.

You can use the WinScrollWindow function to scroll the contents of your application windows. WinScrollWindow scrolls a specified rectangular area of the window by a specified x and y offset (in window coordinates). If you set the SW_INVALIDATERGN flag for this function, the areas you uncover by scrolling are added to the window's update region automatically, causing a WM_PAINT message for the areas to be sent to the window.

For example, as used in the simple text editor described previously, the following call scrolls the text up one line (assuming that the *iVScrollInc* parameter specifies the height of the current font) and adds the uncovered area at the bottom of the window to the update region.

```
HWND hwnd;
LONG iVScrollInc;

/* Scroll, adding a new area to the update region.          */

WinScrollWindow(hwnd,  /* Window handle                     */
    0,                 /* x displacement                    */
    -(iVScrollInc),    /* y displacement                    */
    (PRECTL) NULL,     /* Scroll rectangle is entire window */
    (PRECTL) NULL,     /* Clip rectangle is entire window   */
    (HRGN) NULL,       /* Update region                     */
    (PRECTL) NULL,     /* Update rectangle                  */
    SW_INVALIDATERGN); /* Scroll-window flag                */
```

When the uncovered area is added to the window's update region, a WM_PAINT message is sent to the window. Upon receiving the message, the window draws the line of text at the bottom of the window. If the window has the WS_SYNCPAINT style, the WM_PAINT message is sent to the window before WinScrollWindow returns.

To optimize scrolling speed for repeated scrolling operations, you can omit the SW_INVALIDATERGN flag from the call to WinScrollWindow, which prevents the function from adding the invalid region (uncovered by the scroll) to the window's update region. If you omit the SW_INVALIDATERGN flag, you must pass a region or rectangle to WinScrollWindow. The rectangle or region will contain the area that must be updated after scrolling.

Drawing a Bit Map

The WinDrawBitmap function draws a bit map, identified by a bit map handle, in a specified rectangle. This function enables you to reduce or enlarge the bit map from the source rectangle to the destination rectangle. WinDrawBitmap also can draw in several different copy modes, including using the OR operator to combine source and destination pels.

Drawing Text

There are many ways to draw text in a window in an OS/2 application. The simplest way is to use the WinDrawText function, which draws a single line of text in a specified rectangle, using a variety of alignment methods.

WinDrawText allows you to set a flag so that the function does not draw any text; instead, the function returns the number of characters in the string that will fit in the specified rectangle. For a section of running text, an application can alternate between computation and calls to WinDrawText to draw successive lines of text. When performing this kind of repetitive operation, you can set the DT_WORDBREAK flag in the WinDrawText function to put line breaks on word boundaries rather than between arbitrary characters.

Related Functions

This section covers the functions that are related to Drawing in Windows.

WinCopyRect

This function copies a rectangle from *prclSrc* to *prclDst*.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinCopyRect  (HAB  hab, PRECTL  prclDst, PRECTL  prclSrc)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prclDst (PRECTL) – output
Destination rectangle.

prclSrc (PRECTL) – input
Source rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinDrawBorder

This function draws the borders and interior of a rectangle.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinDrawBorder  (HPS hps, PRECTL prcl, LONG cx, LONG cy,
                     LONG clrFore, LONG clrBack, ULONG flCmd)
```

Parameters

hps (HPS) – input

Presentation-space handle.

prcl (PRECTL) – input

Bounding rectangle for the border.

cx (LONG) – input

Width of border rectangle vertical sides.

cy (LONG) – input

Width of border rectangle horizontal sides.

clrFore (LONG) – input

Color of edge of border.

clrBack (LONG) – input

Color of interior of border.

flCmd (ULONG) – input

Flags controlling the way in which the border is drawn.

DB_ROP

A group of flags that specify the mix to be used, for both the border and the interior.

DB_PATCOPY

Use the ROP_PATCOPY raster operation (see "GpiBitBlt" in the *GPI*). This is a copy of the pattern to the destination.

DB_PATINVERT

Use the ROP_PATINVERT raster operation (see "GpiBitBlt" in the *GPI*). This is an exclusive-OR of the pattern with the destination.

DB_DESTINVERT

Use the ROP_DESTINVERT raster operation (see "GpiBitBlt" in the *GPI*). This inverts the destination.

DB_AREAMIXMODE

Map the current area foreground mix attribute into a Bitblt raster operation (see "GpiBitBlt" in the *GPI*). The area background mix mode is ignored.

DB_INTERIOR
The area contained within the given rectangle, and not included within the borders (as given by *cx* and *cy*), is drawn.

DB_AREAATTRS

- If this is specified:

  For any border, the pattern used is the pattern as currently defined in the area attribute.

  For any interior, the pattern used is the same as if GpiSetAttrs for the area attributes is made with the background color of the area attribute being passed for the foreground color, and the foreground color of the area attribute being passed as the background color.

- If this is not specified (default):

  For any border, the pattern used is the same as if GpiSetAttrs for the area attributes is made with a foreground color of *clrFore*, and a background color of *clrBack*.

  For any interior, the pattern used is the same as if GpiSetAttrs for the area attributes is made with a foreground color of *clrBack*, and a background color of *clrFore*.

DB_STANDARD
cx and *cy* are multiplied by the system SV_CXBORDER and SV_CYBORDER constants to produce the widths of the vertical and horizontal sides of the border.

DB_DLGBORDER
A standard dialog border is drawn, in the active titlebar color if DB_PATCOPY is specified, or the inactive titlebar color if DB_PATINVERT is specified. Other DB_ROP options, and DB_AREAATTRS, are ignored.

DB_ROP and DB_AREAATTRS are also ignored for the interior. The interior is drawn in the color specified by *clrBack*.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinDrawText

This function draws a single line of formatted text into a specified rectangle.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinDrawText  (HPS hps, LONG cchText, PCH lpchText, PRECTL prcl,
                   LONG clrFore, LONG clrBack, ULONG flCmd)
```

Parameters

hps (HPS) – input
Presentation-space handle.

cchText (LONG) – input
Count of the number of characters in the string.

-1L	The string is null-terminated and its length is to be calculated by this function.
Other	Count of the number of characters in the string.

lpchText (PCH) – input
Character string to be drawn.

prcl (PRECTL) – in/out
Text rectangle.

clrFore (LONG) – input
Foreground color.

clrBack (LONG) – input
Background color.

flCmd (ULONG) – input
An array of flags that determines how the text is drawn.

DT_LEFT	Left-justify the text.
DT_CENTER	Center the text.
DT_RIGHT	Right-justify the text.
DT_VCENTER	Vertically center the text.
DT_TOP	Top-justify the text.
DT_BOTTOM	Bottom-justify the text.
DT_HALFTONE	Halftone the text display.

DT_MNEMONIC

If a mnemonic prefix character is encountered, the next character is drawn with mnemonic emphasis.

DT_QUERYEXTENT

The height *prcl* is changed to a rectangle that bounds the string if it were drawn with WinDrawText.

DT_WORDBREAK

Only words that fit completely within the supplied rectangle are drawn. A *word* is defined as:

Any number of leading spaces followed by one or more visible characters and terminated by a space, carriage return, or line-feed character.

When calculating whether a particular word fits within the given rectangle, this function does not consider the trailing blanks. Only the length of the visible part of the word is tested against the right edge of the rectangle.

Also, note that this function always tries to draw at least one word, even if that word does not fit in the passed rectangle. This is so that progress is always made when drawing multiline text.

DT_EXTERNALLEADING

This flag causes the "external leading" value for the current font to be added to the bottom of the bounding rectangle before returning. It has an effect only when both DT_TOP and DT_QUERYEXTENT are also specified.

DT_TEXTATTRS

If this is specified, text is drawn using the character foreground and background colors of the presentation space, and *clrFore* and *clrBack* are ignored.

DT_ERASERECT

If this is specified, the rectangle defined by *prcl* is erased before drawing the text. Otherwise, the background of the characters themselves can be erased if the character background mix (see "GpiSetAttrs" and "GpiSetBackMix" in the *GPI*) is set to BM_OVERPAINT.

DT_UNDERSCORE

Underscore the characters. See FATTR_SEL_UNDERSCORE in the FATTRS datatype.

DT_STRIKEOUT

Overstrike the characters. See FATTR_SEL_STRIKEOUT in the FATTRS datatype.

Returns

lChars (LONG) – returns

Count of characters drawn within the rectangle.

0	Error occurred
Other	Count of characters drawn within the rectangle.

WinEqualRect

This function compares two rectangles for equality.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinEqualRect  (HAB hab, PRECTL prcl1, PRECTL prcl2)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prcl1 (PRECTL) – input
First rectangle.

prcl2 (PRECTL) – input
Second rectangle.

Returns

rc (BOOL) – returns
Equality indicator.

TRUE	Rectangles are identical
FALSE	Rectangles are not identical, or an error occurred.

WinFillRect

This function draws a filled rectangular area.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinFillRect  (HPS  hps, PRECTL  prcl, LONG  lColor)
```

Parameters

hps (HPS) – input
Presentation-space handle.

prcl (PRECTL) – input
Rectangle to be filled, in window coordinates.

lColor (LONG) – input
Color with which to fill the rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinIntersectRect

This function calculates the intersection of the two source rectangles and returns the result in the destination rectangle.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinIntersectRect  (HAB hab, PRECTL pcrlDst, PRECTL pcrlSrc1,
                        PRECTL pcrlSrc2)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pcrlDst (PRECTL) – output
Intersection rectangle.

pcrlSrc1 (PRECTL) – input
First rectangle.

pcrlSrc2 (PRECTL) – input
Second rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Source rectangles intersect
FALSE	Source rectangles do not intersect, or an error occurred.

WinInvertRect

This function inverts a rectangular area.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinInvertRect  (HPS hps, PRECTL prclRect)
```

Parameters

hps (HPS) – input
Presentation-space handle.

prclRect (PRECTL) – input
Rectangle to be inverted.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinIsRectEmpty

This function checks whether a rectangle is empty.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsRectEmpty  (HAB hab, PRECTL prclprc)
```

Parameters

hab (HAB) – input

Anchor-block handle.

prclprc (PRECTL) – input

Rectangle to be checked.

Returns

rc (BOOL) – returns

Empty indicator.

TRUE	Rectangle is empty
FALSE	Rectangle is not empty.

WinMakeRect

This function converts a rectangle to a graphics rectangle.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinMakeRect (HAB hab, PRECTL pwrc)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pwrc (PRECTL) – in/out
Rectangle to be converted.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinMapWindowPoints

This function maps a set of points from a coordinate space relative to one window into a coordinate space relative to another window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinMapWindowPoints  (HWND hwndFrom, HWND hwndTo,
                          PPOINTL prgptl, LONG cwpt)
```

Parameters

hwndFrom (HWND) – input
Handle of the window from whose coordinates points are to be mapped.

HWND_DESKTOP	Points are mapped from screen coordinates
Other	Points are mapped from window coordinates.

hwndTo (HWND) – input
Handle of the window to whose coordinates points are to be mapped.

HWND_DESKTOP	Points are mapped into screen coordinates
Other	Points are mapped into window coordinates.

prgptl (PPOINTL) – in/out
Points to be mapped to the new coordinate system.

cwpt (LONG) – input
Number of points to be mapped.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinOffsetRect

This function offsets a rectangle.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinOffsetRect  (HAB hab, PRECTL prcl, LONG cx, LONG cy)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prcl (PRECTL) – in/out
Rectangle to be offset.

cx (LONG) – input
x-value of offset.

cy (LONG) – input
y-value of offset.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinPtInRect

This function queries whether a point lies within a rectangle.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinPtInRect  (HAB hab, PRECTL prcl, PPOINTL pptl)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prcl (PRECTL) – input
Rectangle to be queried.

pptl (PPOINTL) – input
Point to be queried.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	*pptl* lies within *prcl*.
FALSE	*pptl* does not lie within *prcl*, or an error occurred.

WinSetRect

This function sets rectangle coordinates.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetRect  (HAB hab, PRECTL prclrect, LONG lLeft, LONG lBottom,
                  LONG lRight, LONG lTop)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prclrect (PRECTL) – in/out
Rectangle to be updated.

lLeft (LONG) – input
Left edge of rectangle.

lBottom (LONG) – input
Bottom edge of rectangle.

lRight (LONG) – input
Right edge of rectangle.

lTop (LONG) – input
Top edge of rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetRectEmpty

This function sets a rectangle empty.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetRectEmpty  (HAB hab, PRECTL prclrect)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prclrect (PRECTL) – in/out
Rectangle to be set empty.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinShowTrackRect

This function hides or shows the tracking rectangle.

Syntax

```
#define INCL_WINTRACKRECT /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinShowTrackRect  (HWND hwnd, BOOL fShow)
```

Parameters

hwnd (HWND) – input
Window handle.

fShow (BOOL) – input
Show indicator.

TRUE	Show the tracking rectangle
FALSE	Hide the tracking rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSubtractRect

This function subtracts one rectangle from another.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSubtractRect  (HAB hab, PRECTL prclDest, PRECTL prclSrc1,
                       PRECTL prclSrc2)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prclDest (PRECTL) – output
Result.

prclSrc1 (PRECTL) – input
First source rectangle.

prclSrc2 (PRECTL) – input
Second source rectangle.

Returns

rc (BOOL) – returns
Not-empty indicator.

TRUE	Rectangle is not empty
FALSE	Rectangle is empty or an error occurred.

WinTrackRect

This function draws a tracking rectangle.

Syntax

```
#define INCL_WINTRACKRECT /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinTrackRect  (HWND hwnd, HPS hps, PTRACKINFO ptiTrackinfo)
```

Parameters

hwnd (HWND) – input

Window handle where tracking is to take place.

HWND_DESKTOP	Track over the entire screen
Other	Track over specified window only.

hps (HPS) – input

Presentation-space handle.

NULLHANDLE — The *hwnd* parameter is used to calculate a presentation space for tracking. It is assumed that tracking takes place within *hwnd* and that the style of this window is not WS_CLIPCHILDREN. Thus, when the drag rectangle appears, it is not clipped by any children within the window. If the window style is WS_CLIPCHILDREN and the application causes the drag rectangle to be clipped, it must explicitly pass an appropriate presentation space.

Other — Specified presentation-space handle.

ptiTrackinfo (PTRACKINFO) – in/out

Track information.

Returns

rc (BOOL) – returns

Success indicator.

TRUE — Tracking successful.

FALSE — Tracking canceled, or the pointing device was already captured when this function was called.

Only one tracking rectangle can be in use at one time.

WinUnionRect

This function calculates a rectangle that bounds the two source rectangles.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinUnionRect  (HAB hab, PRECTL prclDest, PRECTL prclSrc1,
                    PRECTL prclSrc2)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prclDest (PRECTL) – output
Bounding rectangle.

prclSrc1 (PRECTL) – input
First source rectangle.

prclSrc2 (PRECTL) – input
Second source rectangle.

Returns

rc (BOOL) – returns
Nonempty indicator.

TRUE	*prclDest* is a nonempty rectangle
FALSE	Error, or *prclDest* is an empty rectangle.

Related Data Structures

This section covers the data structures that are related to Drawing in Windows.

FATTRS

Font-attributes structure.

Syntax

```
typedef struct _FATTRS {
USHORT      usRecordLength;
USHORT      fsSelection;
LONG        lMatch;
CHAR        szFacename[FACESIZE];
USHORT      idRegistry;
USHORT      usCodePage;
LONG        lMaxBaselineExt;
LONG        lAveCharWidth;
USHORT      fsType;
USHORT      fsFontUse;
 } FATTRS;

typedef FATTRS *PFATTRS;
```

Fields

usRecordLength (USHORT)
Length of record.

fsSelection (USHORT)
Selection indicators.

Flags causing the following features to be simulated by the system.

Note: If an italic flag is applied to a font that is itself defined as italic, the font is slanted further by italic simulation.

Underscore or strikeout lines are drawn using the appropriate attributes (for example, color) from the character bundle (see the CHARBUNDLE datatype), not the line bundle (see LINEBUNDLE). The width of the line, and the vertical position of the line in font space, are determined by the font. Horizontally, the line starts from a point in font space directly above or below the start point of each character, and extends to a point directly above or below the escapement point for that character.

For this purpose, the start and escapement points are those applicable to left-to-right or right-to-left character directions (see GpiSetCharDirection in GPI), even if the string is currently being drawn in a top-to-bottom or bottom-to-top direction.

For left-to-right or right-to-left directions, any white space generated by the character extra and character break extra attributes (see GpiSetCharExtra and GpiSetCharBreakExtra in GPI), as well as increments provided by the vector of increments on GpiCharStringPos and GpiCharStringPosAt, are also underlined/overstruck, so that in these cases the line is continuous for the string.

FATTR_SEL_ITALIC	Generate *italic* font.
FATTR_SEL_UNDERSCORE	Generate underscored font.
FATTR_SEL_BOLD	Generate **bold** font. (Note that the resulting characters are wider than those in the original font.)
FATTR_SEL_STRIKEOUT	Generate font with ~~overstruck~~ characters.
FATTR_SEL_OUTLINE	Use an outline font with hollow characters. If this flag is not set, outline font characters are filled. Setting this flag normally gives better performance, and for sufficiently small characters (depending on device resolution) there may be little visual difference.

lMatch (LONG)
Matched-font identity.

szFacename[FACESIZE] (CHAR)
Typeface name.

The typeface name of the font, for example, Tms Rmn.

idRegistry (USHORT)
Registry identifier.

Font registry identifier (zero if unknown).

usCodePage (USHORT)
Code page.

If zero, the current Gpi code page (see GpiSetCp in GPI) is used. A subsequent GpiSetCp function changes the code page used for this logical font.

lMaxBaselineExt (LONG)
Maximum baseline extension.

For raster fonts, this should be the height of the required font, in world coordinates.

For outline fonts, this should be zero.

lAveCharWidth (LONG)
Average character width.

For raster fonts, this should be the width of the required font, in world coordinates.

For outline fonts, this should be zero.

fsType (USHORT)
Type indicators.

FATTR_TYPE_KERNING	Enable kerning (PostScript** only).
FATTR_TYPE_MBCS	Font for mixed single- and double-byte code pages.
FATTR_TYPE_DBCS	Font for double-byte code pages.

FATTR_TYPE_ANTIALIASED Antialiased font required. Only valid if supported by the device driver.

fsFontUse (USHORT)
Font-use indicators.

These flags indicate how the font is to be used. They affect presentation speed and font quality.

FATTR_FONTUSE_NOMIX Text is not mixed with graphics and can be written without regard to any interaction with graphics objects.

FATTR_FONTUSE_OUTLINE Select an outline (vector) font. The font characters can be used as part of a path definition. If this flag is not set, an outline font might or might not be selected. If an outline font is selected, however, character widths are rounded to an integral number of pels.

FATTR_FONTUSE_TRANSFORMABLE Characters can be transformed (for example, scaled, rotated, or sheared).

POINTL

Point structure (long integers).

Syntax

```
typedef struct _POINTL {
LONG       x;
LONG       y;
 } POINTL;

typedef POINTL *PPOINTL;
```

Fields

x (LONG)
X-coordinate.

y (LONG)
Y-coordinate.

RECTL

Rectangle structure.

Syntax

```
typedef struct _RECTL {
LONG       xLeft;
LONG       yBottom;
LONG       xRight;
LONG       yTop;
 } RECTL;

typedef RECTL *PRECTL;
```

Fields

xLeft (LONG)
X-coordinate of left-hand edge of rectangle.

yBottom (LONG)
Y-coordinate of bottom edge of rectangle.

xRight (LONG)
X-coordinate of right-hand edge of rectangle.

yTop (LONG)
Y-coordinate of top edge of rectangle.

Summary

Following are the OS/2 functions and structures used for drawing in windows.

Table 8-1 (Page 1 of 2). Window-Drawing Functions

Function Name	Description
WinCalcFrameRect	Calculates a client rectangle from a frame rectangle or a frame rectangle from a client rectangle.
WinCopyRect	Copies a rectangle from *prclSrc* to *prclDest*.
WinDrawBitmap	Draws a bit map using the current image colors and mixes.
WinDrawBorder	Draws the borders and interior of a rectangle.
WinDrawText	Draws a single line of formatted text into a specified rectangle.
WinEqualRect	Compares two rectangles for equality.
WinFillRect	Draws a filled rectangular area.
WinInflateRect	Expands a rectangle.
WinIntersectRect	Calculates the intersection of the two source rectangles and returns the result in the destination rectangle.
WinInvalidateRect	Adds a rectangle to a window's update region.
WinInvertRect	Inverts a rectangular area.
WinIsRectEmpty	Determines whether a rectangle is empty.
WinMakeRect	Converts points to graphics points.
WinMapWindowPoints	Maps points from dialog coordinates to window coordinates or from window coordinates to dialog coordinates.
WinOffsetRect	Offsets a rectangle.
WinPtInRect	Queries whether a point lies within a rectangle.
WinQueryUpdateRect	Returns the rectangle that bounds the update region of a specified window.
WinQueryWindowRect	Returns a window rectangle.
WinScrollWindow	Scrolls the contents of a window rectangle.
WinSetRect	Sets rectangle coordinates.
WinSetRectEmpty	Sets a rectangle empty.
WinShowTrackRect	Hides or shows the tracking rectangle.
WinSubtractRect	Subtracts one rectangle from another.
WinTrackRect	Draws a tracking rectangle.
WinUnionRect	Calculates a rectangle that bounds the two source rectangles.

Table 8-1 (Page 2 of 2). Window-Drawing Functions

Function Name	Description
WinValidateRect	Subtracts a rectangle from the update region of an asynchronous paint window, marking that part of the window as visually valid.

Table 8-2. Window-Drawing Structures

Structure Name	Description
FATTRS	Font-attributes structure.
POINTL	Point structure (long integer).
RECTL	Rectangle structure.

Chapter 9. Mouse and Keyboard Input

An OS/2 Presentation Manager application can accept input from both a mouse (or other pointing device) and the keyboard. This chapter explains how these *input events* should be received and processed.

About Mouse and Keyboard Input

Only one window at a time can receive keyboard input, and only one window at a time can receive mouse input; but they do not have to be the same window. All keyboard input goes to the window with the input focus, and, normally, all mouse input goes to the window under the mouse pointer.

System Message Queue

The operating system routes all keystrokes and mouse input to the system message queue, converting these input events into messages, and posts them, one at a time, to the proper application-defined message queues. An application retrieves messages from its queue and dispatches them to the appropriate window procedures, which process the messages.

Mouse and keyboard input events in the system message queue are strictly ordered so that a new event cannot be processed until all previous events are fully processed: the system cannot determine the destination window of an input event until then. For example, if a user types a command in one window, clicks the mouse to activate another window, then types a command in the second window, the destination of the second command depends on how the application handles the mouse click. The second command would go to the second window only if that window became active as a result of the mouse click.

It is important for an application to process all messages quickly to avoid slowing user interaction with the system. A message must be responded to immediately in the current thread, but the processing it initiates should be done asynchronously in another thread that has no windows in the desktop tree.

The OS/2 operating system can display multiple windows belonging to several applications at the same time. To manage input among these windows, the system uses the concepts of *window activation* and *keyboard focus*.

Window Activation

Although the operating system can display windows from many different applications simultaneously during a PM session, the user can interact with only one application at a time—the *active* application. The other applications continue to run, but they cannot receive user input until they become active.

To enable the user to easily identify the active application, the system activates all frames in the tree between HWND_DESKTOP and the window with input focus. That is, the system positions the active frame window above all other top-level windows on the screen. If the active window is a standard frame window, the window's title bar and sizing border are highlighted.

The user can control which application is active by clicking on a window or by pressing the Alt+Tab or Alt+Esc key combinations. An application can set the active frame window by calling WinSetActiveWindow; it also can obtain the handle of the active frame window by using WinQueryActiveWindow.

When one window is deactivated and another activated, the system sends a WM_ACTIVATE message, first to the window being deactivated, then to the window being activated. The *fActive* parameter of the WM_ACTIVATE message is set to FALSE for the window being deactivated and set to TRUE for the window being activated. An application can use this message to track the activation state of a client window.

Keyboard Focus

The *keyboard focus* is a temporary attribute of a window; the window that has the keyboard focus receives all keyboard input until the focus changes to a different window. The system converts keyboard input events into WM_CHAR messages and posts them to the message queue of the window that has the keyboard focus.

An application can set the keyboard focus to a particular window by calling WinSetFocus. If the application does not use WinSetFocus to explicitly set the keyboard-focus window, the system sets the focus to the active frame window.

The following events occur when an application uses WinSetFocus to shift the keyboard focus from one window (the *original* window) to another (the *new* window):

1. The system sends the original window a WM_SETFOCUS message (with the *fFocus* parameter set to FALSE), indicating that that window has lost the keyboard focus.
2. The system then sends the original window a WM_SETSELECTION message, indicating that the window should remove the highlight from the current selection.
3. If the original (frame) window is being deactivated, the system sends it a WM_ACTIVATE message (with the *fActive* parameter set to FALSE), indicating that the window is no longer active.
4. The system then sends the new application a WM_ACTIVATE message (with *fActive* set to TRUE), indicating that the new application is now active.
5. If the new (main) window is being activated, the system sends it a WM_ACTIVATE message (with *fActive* set to TRUE), indicating that the main window is now active.
6. The system sends the new window a WM_SETSELECTION message, indicating that the window should highlight the current selection.
7. Finally, the system sends the new window a WM_SETFOCUS message (with *fFocus* set to TRUE), indicating that the new window has the keyboard focus.

If, while processing a WM_SETFOCUS message, an application calls WinQueryActiveWindow, that function returns the handle of the previously-active window until the application establishes a new active window. Similarly, if the application, while processing WM_SETFOCUS, calls WinQueryFocus, that function returns the handle of the previous keyboard-focus window until the application establishes a new keyboard-focus window. In other words, even though the system has sent WM_ACTIVATE and WM_SETFOCUS messages (with the *fActive* and *fFocus* parameters set to FALSE) to the previous windows, those windows are considered the active and focus windows until the system establishes new active and focus windows.

If the application calls WinSetFocus while processing a WM_ACTIVATE message, the system does not send a WM_SETFOCUS message (with *fFocus* set to FALSE), because no window has the focus.

A client window receives a WM_ACTIVATE message when its parent frame window is being activated or deactivated. The activation or deactivation message usually is followed by a WM_SETFOCUS message that specifies whether the client window is gaining or losing the keyboard focus. Therefore, if the client window needs to change the keyboard focus, it should do so during the WM_SETFOCUS message, not during the WM_ACTIVATE message.

Keyboard Messages

The system sends keyboard input events as WM_CHAR messages to the message queue of the keyboard-focus window. If no window has the keyboard focus, the system posts WM_CHAR messages to the message queue of the active frame window. Following are two typical situations in which an application receives WM_CHAR messages:

> An application has a client window or custom control window, either of which can have the keyboard focus. If the window procedure for the client or control window does not process WM_CHAR messages, it should pass them to WinDefWindowProc, which will pass them to the owner. Dialog control windows, in particular, should pass unprocessed WM_CHAR messages to the WinDefDlgProc function, because this is how the user interface implements control processing for the Tab and Arrow keys.
>
> An application window owns a control window whose window procedure can handle some, but not all, WM_CHAR messages. This is common in dialog windows. If the window procedure of a control in a dialog window cannot process a WM_CHAR message, the procedure can pass the message to the WinDefDlgProc function. This function sends the message to the control window's owner, which usually is a dialog frame window. The application's dialog procedure then receives the WM_CHAR message. This also is the case when an application client window owns a control window.

A WM_CHAR message can represent a key-down or key-up transition. It might contain a character code, virtual-key code, or scan code. This message also contains information about the state of the Shift, Ctrl, and Alt keys.

Each time a user presses a key, at least two WM_CHAR messages are generated: one when the key is pressed, and one when the key is released. If the user holds down the key long enough to trigger the keyboard repeat, multiple WM_CHAR key-down messages are generated. If the keyboard repeats faster than the application can retrieve the input events from its message queue, the system combines repeating character events into one

WM_CHAR message and increments a count byte that indicates the number of keystrokes represented by the message. Generally, this byte is set to 1, but an application should check each WM_CHAR message to avoid missing any keystrokes.

An application can ignore the repeat count. For example, an application might ignore the repeat count on Arrow keys to prevent the cursor from skipping characters when the system is slow.

Message Flags

Applications decode WM_CHAR messages by examining individual bits in the flag word contained in the first message parameter (*mp1*) that the system passes with every WM_CHAR message. The type of flag word indicates the nature of the message. The system can set the bits in the flag word in various combinations. For example, a WM_CHAR message can have the KC_CHAR, KC_SCANCODE, and KC_SHIFT attribute bits all set at the same time. An application can use the following list of flag values to test the flag word and determine the nature of a WM_CHAR message:

Table 9-1 (Page 1 of 2). Keyboard Character Flags

Flag Name	Description
KC_ALT	Indicates that the Alt key was down when the message was generated.
KC_CHAR	Indicates that the message contains a valid character code for a key, typically an ASCII character code.
KC_COMPOSITE	In combination with the KC_CHAR flag, this flag indicates that the character code is a combination of the key that was pressed and the previous dead key. This flag is used to create characters with diacritical marks.
KC_CTRL	Indicates that the Ctrl key was down when the message was generated.
KC_DEADKEY	In combination with the KC_CHAR flag, this flag indicates that the character code represents a dead-key glyph (such as an accent). An application displays the dead-key glyph and does not advance the cursor. Typically, the next WM_CHAR message is a KC_COMPOSITE message, containing the glyph associated with the dead key.
KC_INVALIDCHAR	Indicates that the character is not valid for the current translation tables.
KC_INVALIDCOMP	Indicates that the character code is not valid in combination with the previous dead key.
KC_KEYUP	Indicates that the message was generated when the user released the key. If this flag is clear, the message was generated when the user pressed the key. An application can use this flag to determine key-down and key-up events.
KC_LONEKEY	In combination with the KC_KEYUP flag, this flag indicates that the user pressed no other key while this key was down.
KC_PREVDOWN	In combination with the KC_VIRTUALKEY flag, this flag indicates that the virtual key was pressed previously. If this flag is clear, the virtual key was not previously pressed.

Table 9-1 (Page 2 of 2). Keyboard Character Flags

Flag Name	Description
KC_SCANCODE	Indicates that the message contains a valid scan code generated by the keyboard when the user pressed the key. The system uses the scan code to identify the character code in the current code page; therefore, most applications do not need the scan code unless they cannot identify the key that the user pressed. WM_CHAR messages generated by user keyboard input generally have a valid scan code, but WM_CHAR messages posted to the queue by other applications might not contain a scan code.
KC_SHIFT	Indicates that the Shift key was down when the message was generated.
KC_TOGGLE	Toggles on and off every time the user presses a specified key. This is important for keys like NumLock, which have an on or off state.
KC_VIRTUALKEY	Indicates that the message contains a valid virtual-key code for a key. Virtual keys typically correspond to function keys. For those using hooks, when this bit is set, KC_SCANCODE should usually be set as well.

The *mp1* and *mp2* parameters of the WM_CHAR message contain information describing the nature of a keyboard input event, as follows:

- SHORT1FROMMP (*mp1*) contains the flag word.
- CHAR3FROMMP (*mp1*) contains the key-repeat count.
- CHAR4FROMMP (*mp1*) contains the scan code.
- SHORT1FROMMP (*mp2*) contains the character code.
- SHORT2FROMMP (*mp2*) contains the virtual key code.

An application window procedure should return TRUE if it processes a particular WM_CHAR message or FALSE if it does not. Typically, applications respond to key-down events and ignore key-up events.

The following sections describe the different types of WM_CHAR messages. Generally, an application decodes these messages by creating layers of conditional statements that discriminate among the different combinations of flag and code attributes that can occur in a keyboard message.

Key-Down or Key-Up Events

Typically, the first attribute that an application checks in a WM_CHAR message is the key-down or key-up event. If the KC_KEYUP bit of the flags word is set, the message is from a key-up event. If the flag is clear, the message is from a key-down event.

Repeat-Count Events

An application can check the key-repeat count of a WM_CHAR message to determine whether the message represents more than 1 keystroke. The count is greater than 1 if the keyboard is sending characters to the system queue faster than the application can retrieve them. If the system queue fills up, the system combines consecutive keyboard input events for each key into a single WM_CHAR message, with the key-repeat count set to the number of combined events.

Character Codes

The most typical use of WM_CHAR messages is to extract a character code from the message and display the character on the screen. When the KC_CHAR flag is set in the WM_CHAR message, the low word of *mp2* contains a character code based on the current code page. Generally, this value is a character code (typically, an ASCII code) for the key that was pressed.

Virtual-Key Codes

WM_CHAR messages often contain virtual-key codes that correspond to various function keys and direction keys on a typical keyboard. These keys do not correspond to any particular glyph code but are used to initiate operations. When the KC_VIRTUALKEY flag is set in the flag word of a WM_CHAR message, the high word of *mp2* contains a virtual-key code for the key.

Note: Some keys, such as the Enter key, have both a valid character code and a virtual-key code. WM_CHAR messages for these keys will contain character codes for both newline characters (ASCII 11) and virtual-key codes (VK_ENTER).

Scan Codes

A third possible value in a WM_CHAR message is the scan code of the key that was pressed. The scan code represents the value that the keyboard hardware generates when the user presses a key. An application can use the scan code to identify the physical key pressed, as opposed to the character code represented by the same key.

Accelerator-Table Entries

The system checks all incoming keyboard messages to see whether they match any existing accelerator-table entries (in either the system message queue or the application message queue). The system first checks the accelerator table associated with the active frame window; if it does not find a match, the system uses the accelerator table associated with the message queues. If the keyboard input event corresponds to an accelerator-table entry, the system changes the WM_CHAR message to a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, depending on the attributes of the accelerator table. If the keyboard input event does not correspond to an accelerator-table entry, the system passes the WM_CHAR message to the keyboard-focus window.

Applications should use accelerator tables to implement keyboard shortcuts rather than translate command keystrokes. For example, if an application uses the F2 key to save a document, the application should create a keyboard accelerator entry for the F2 virtual key so that, when pressed, the F2 key generates a WM_COMMAND message rather than a WM_CHAR message.

Mouse Messages

Mouse messages occur when a user presses or releases one of the mouse buttons (a click) and when the mouse moves. All mouse messages contain the *x* and *y* coordinates of the mouse-pointer *hot spot* (relative to the coordinates of the window receiving the message) at the time the event occurs. The mouse-pointer hot spot is the location in the mouse-pointer bit map that the system tracks and recognizes as the position of the mouse pointer.

If a window has the CS_HITTEST style, the system sends the window a WM_HITTEST message when the window is about to receive a mouse message. Most applications pass WM_HITTEST messages on to WinDefWindowProc by default, so disabled windows do not receive mouse messages. Windows that specifically respond to WM_HITTEST messages can change this default behavior. If the window is enabled and should receive the mouse message, the WinDefWindowProc function (using the default processing for WM_HITTEST) returns the value HT_NORMAL. If the window is disabled, WinDefWindowProc returns HT_ERROR, in which case the window does not receive the mouse message.

The default window procedure processes the WM_HITTEST message and the *usHit* parameter in the WM_MOUSEMOVE message. Therefore, unless an application needs to return special values for the WM_HITTEST message or the *usHit* parameter, it can ignore them. One possible reason for processing the WM_HITTEST message is for the application to react differently to a mouse click in a disabled window.

The contents of the mouse-message parameters (*mp1* and *mp2*) are as follows:

- SHORT1FROMMP (*mp1*) contains the *x* position.
- SHORT2FROMMP (*mp1*) contains the *y* position.
- SHORT1FROMMP (*mp2*) contains the hit-test parameter.

Capturing Mouse Input

The operating system generally posts mouse messages to the window that is under the mouse pointer at the time the system reads the mouse input events from the system message queue. An application can change this by using the WinSetCapture function to route all mouse messages to a specific window or to the message queue associated with the current thread. If mouse messages are routed to a specific window, that window receives all mouse input until either the window releases the mouse or the application specifies another capture window. If mouse messages are routed to the current message queue, the system posts each mouse message to the queue with the *hwnd* member of the QMSG structure for each message set to NULL. Because no window handle is specified, the WinDispatchMsg function in the application's main message loop cannot pass these messages to a window procedure for processing. Therefore, the application must process these messages in the main loop.

Capturing mouse input is useful if a window needs to receive all mouse input, even when the pointer moves outside the window. For example, applications commonly track the mouse-pointer position after a mouse "button down" event, following the pointer until a "button up" event is received from the system. If an application does not call WinSetCapture for a window and the user releases the mouse button, the application does not receive the button-up message. If the application sets a window to capture the mouse and tracks the mouse pointer, the application receives the button-up message even if the user moves the mouse pointer outside the window.

Some applications are designed to require a button-up message to match a button-down message. When processing a button-down message, these applications call WinSetCapture to set the capture to their own window; then, when processing a matching button-up message, they call WinSetCapture, with a NULL window handle, to release the mouse.

Button Clicks

An application window's response to a mouse click depends on whether the window is active. The first click in an inactive window should activate the window. Subsequent clicks in the active window produce an application-specific action.

A common problem for an application that processes WM_BUTTON1DOWN or similar messages is failing to activate the window or set the keyboard focus. If the window processes WM_CHAR messages, the window procedure should call WinSetFocus to make sure the window receives the keyboard focus and is activated. If the window does not process WM_CHAR messages, the application should call WinSetActiveWindow to activate the window.

Mouse Movement

The system sends WM_MOUSEMOVE messages to the window that is under the mouse pointer, or to the window that currently has captured the mouse, whenever the mouse pointer moves. This is useful for tracking the mouse pointer and changing its shape, based on its location in a window. For example, the mouse pointer changes shape when it passes over the size border of a standard frame window.

All standard control windows use WM_MOUSEMOVE messages to set the mouse-pointer shape. If an application handles WM_MOUSEMOVE messages in some situations but not others, unused messages should be passed to the WinDefWindowProc function to change the mouse-pointer shape.

Using the Mouse and Keyboard

This section explains how to perform the following tasks:

- Determine the active status of a frame window.
- Check for a key-up or key-down event.
- Respond to a character message.
- Handle virtual-key codes.
- Handle a scan code.

Determining the Active Status of a Frame Window

The activated state of a window is a frame-window characteristic. The system does not provide an easy way to determine whether a client window is part of the active frame window. That is, the window handle returned by the WinQueryActiveWindow function identifies the active frame window rather than the client window owned by the frame window.

Following are two methods for determining the activated state of a frame window that owns a particular client window:

- Call WinQueryActiveWindow and compare the window handle it returns with the handle of the frame window that contains the client window, as shown in the following code fragment:

```
HWND hwndClient;
BOOL fActivated;

fActivated = (WinQueryWindow(hwndClient, QW_PARENT) ==
              WinQueryActiveWindow(HWND_DESKTOP));
```

- Each time the frame window is activated, the client window receives a WM_ACTIVATE message with the low word of the *mp2* equal to TRUE. When the frame window is deactivated, the client window receives a WM_ACTIVATE message with a FALSE activation indicator.

Checking for a Key-Up or Key-Down Event

The following code fragment shows how to decode a WM_CHAR message to determine whether it indicates a key-up event or a key-down event:

```
USHORT fsKeyFlags;

case WM_CHAR:  {
USHORT fsKeyFlags = SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_KEYUP) {
     .
     . /* Perform key-up processing.   */
     .
} else {
     .
     . /* Perform key-down processing. */
     .
}

return;

}
```

Responding to a Character Message

The following code fragment shows how to respond to a character message:

```
USHORT fsKeyFlags;
UCHAR  uchChr1;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_CHAR) {

    /* Get the character code from mp2. */
    uchChr1 = (UCHAR) CHAR1FROMMP(mp2);
    .
    . /* Process the character.        */
    .

    return TRUE;
}
```

If the KC_CHAR flag is not set, the *mp2* parameter from CHAR1FROMMP still might contain useful information. If either the Alt key or the Ctrl key, or both, are down, the KC_CHAR bit is not set when the user presses another key. For example, if the user presses the **a** key when the Alt key is down, the low word of *mp2* contains the ASCII value for “a” (0x0061), the KC_ALT flag is set, and the KC_CHAR flag is clear. If the translation does not generate any valid characters, the *char* field is set to 0.

Handling Virtual-Key Codes

The following code fragment shows how to decode a WM_CHAR message containing a valid virtual-key code:

```
USHORT fsKeyFlags;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_VIRTUALKEY) {

    /* Get the virtual key from mp2.       */
    switch (SHORT2FROMMP(mp2)) {
     case VK_TAB:
         .
         . /* Process the TAB key.         */
         .
         return TRUE;
     case VK_LEFT:
         .
         . /* Process the LEFT key.        */
         .
         return TRUE;
     case VK_UP:
         .
         . /* Process the UP key.          */
         .
         return TRUE;
     case VK_RIGHT:
         .
         . /* Process the RIGHT key.       */
         .
         return TRUE;
     case VK_DOWN:
         .
         . /* Process the DOWN key.        */
         .
         return TRUE;
       .
       . /* Etc...                         */
       .
       default:
           return FALSE;
       }
   }
```

Handling a Scan Code

All WM_CHAR messages generated by keyboard input events have valid scan codes. WM_CHAR messages posted by other applications might or might not have valid scan codes. The following code fragment shows how to extract a scan code from a WM_CHAR message:

```
USHORT fsKeyFlags;
UCHAR  uchScanCode;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_SCANCODE) {

    /* Get the scan code from mp1.   */
    uchScanCode = CHAR4FROMMP(mp1);
    .
    . /* Process the scan code.      */
    .

    return (MRESULT) TRUE;
   }
```

Related Functions

This section covers the functions that are related to Mouse and Keyboard Input.

WinEnablePhysInput

This function enables or disables queuing of physical input (keyboard or mouse).

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinEnablePhysInput  (HWND  hwndDesktop, BOOL  fEnable)
```

Parameters

hwndDesktop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

fEnable (BOOL) – input

New state for the queuing of physical input.

TRUE	&Pdev. and keyboard input are queued
FALSE	&Pdev. and keyboard input are disabled.

Returns

rc (BOOL) – returns

Previous state for the queuing of physical input.

TRUE	&Pdev. and keyboard input were queued
FALSE	&Pdev. and keyboard input were disabled.

WinFocusChange

This function changes the focus window.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinFocusChange  (HWND hwndDeskTop, HWND hwndNewFocus,
                      ULONG flFocusChange)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

hwndNewFocus (HWND) – input
Window handle to receive the focus.

flFocusChange (ULONG) – input
Focus changing indicators.

FC_NOSETFOCUS	Do not send the WM_SETFOCUS message to the window receiving the focus.
FC_NOLOSEFOCUS	Do not send the WM_SETFOCUS message to the window losing the focus.
FC_NOSETACTIVE	Do not send the WM_ACTIVATE message to the window being activated.
FC_NOLOSEACTIVE	Do not send the WM_ACTIVATE message to the window being deactivated.
FC_NOSETSELECTION	Do not send the WM_SETSELECTION message to the window being selected.
FC_NOLOSESELECTION	Do not send the WM_SETSELECTION message to the window being deselected.
FC_NOBRINGTOTOP	Do not bring any window to the top.
FC_NOBRINGTOTOPFIRSTWINDOW	Do not bring the first frame window to the top.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinGetKeyState

This function returns the state of the key at the time that the last message obtained from the queue was posted.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
LONG WinGetKeyState  (HWND hwndDeskTop, LONG vkey)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

vkey (LONG) – input

Virtual key value.

Returns

lKeyState (LONG) – returns

Key state.

0x0001	The key has been pressed an odd number of times since the system has been started.
0x8000	The key is down.

WinGetPhysKeyState

This function returns the physical key state.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinGetPhysKeyState  (HWND hwndDeskTop, LONG sc)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

sc (LONG) – input
Hardware scan code.

Returns

lKeyState (LONG) – returns
Key state.

0x0001 The key has been pressed an odd number of times since the system has been started.

0x0002 The key has been pressed since the last time this function was issued, or since the system has been started if this is the first time the call has been issued.

0x8000 The key is down.

WinIsPhysInputEnabled

This function returns the status of hardware input (on/off).

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsPhysInputEnabled  (HWND  hwndDeskTop)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle

Returns

rc (BOOL) – returns

Return value.

TRUE	If input is enabled.
FALSE	If input is disabled.

WinQueryCapture

This function returns the handle of the window that has the pointer captured.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinQueryCapture  (HWND  hwndDesktop)
```

Parameters

hwndDesktop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hwnd (HWND) – returns

Handle of the window with the pointer captured.

NULLHANDLE	No window has the pointer captured, or an error occurred
Handle	Handle of the window with the pointer captured.

WinSetCapture

This function captures all pointing device messages.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetCapture  (HWND  hwndDesktop, HWND  hwnd)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle, or HWND_DESKTOP.

hwnd (HWND) – input
Handle of the window that is to receive all pointing device messages.

Returns

rc (BOOL) – returns
Success indicator.

TRUE Successful completion.

FALSE Error occurred. If the pointing device has already been captured by another thread or window, the call fails. This is to prevent applications from removing the capture from other windows or threads.

WinSetKeyboardStateTable

This function gets or sets the keyboard state.

Syntax

```
#define INCL_WININPUT /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetKeyboardStateTable  (HWND hwndDeskTop,
                                PBYTE abKeyStateTable, BOOL fSet)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

abKeyStateTable (PBYTE) – in/out
Key state table.

fSet (BOOL) – input
Set indicator.

TRUE	The keyboard state is set from *abKeyStateTable*
FALSE	The keyboard state is copied to *abKeyStateTable*.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Mouse and Keyboard Input.

WM_SETFOCUS

This message occurs when a window is to receive or lose the input focus.

Parameters

param1

hwnd (HWND)
Focus-window handle.

NULLHANDLE	No window is losing or receiving the focus.
Other	Window handle.

param2

usfocus (USHORT)
Focus flag.

TRUE The window is receiving the focus. *hwnd* is the window handle of the window losing the focus, or NULLHANDLE if no window previously had the focus.

FALSE The window is losing the focus. *hwnd* is the window handle of the window receiving the focus, or NULLHANDLE if no window is receiving the focus.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Summary

Following are the OS/2 functions and messages used with activation and keyboard/mouse input.

Table 9-2. Mouse/Keyboard Functions

Function Name	Description
WinEnablePhysInput	Enables or disables queuing of physical input.
WinFocusChange	Changes the focus window.
WinGetKeyState	Returns the state of the key at the time the last message from the message queue was posted.
WinGetPhysKeyState	Returns the physical key state.
WinIsPhysInputEnabled	Returns the status of the hardware (*on/off*)
WinQueryActiveWindow	Returns the active window for HWND_DESKTOP or other parent window.
WinQueryCapture	Returns the handle of the window the pointer has captured.
WinQueryFocus	Returns the focus window; NULL if there is not focus window.
WinSetActiveWindow	Makes the frame window the active window.
WinSetCapture	Captures all pointing device messages.
WinSetFocus	Sets the focus window.
WinSetKeyboardStateTable	Gets or sets the keyboard state.

Table 9-3. Focus-Change and Activation Messages

Message	Description
WM_ACTIVATE	Sent when a different window becomes the active window.
WM_FOCUSCHANGE	Occurs when the window having the focus is changed.
WM_QUERYFOCUSCHAIN	Requests the handle of a window in the focus chain.
WM_SETFOCUS	Occurs when a window is to lose or gain the input focus.
WM_SETSELECTION	Occurs when a window is selected or deselected.

Table 9-4. Mouse Messages

Message	Description
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON1UP	Occurs when the user releases pointer button 1.
WM_BUTTON2DBLCLK	Occurs when the user presses button 2 of the pointing device twice.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON2UP	Occurs when the user releases pointer button 2.
WM_BUTTON3DBLCLK	Occurs when the user presses button 3 on the pointing device twice.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON3UP	Occurs when the user releases pointer button 3.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.

Table 9-5. Keyboard Messages

Message	Description
WM_CHAR	Occurs when the user presses a key.
WM_COMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table into WM_COMMAND.

Chapter 10. Mouse Pointers and Icons

A *mouse pointer* is a special bit map the operating system uses to show a user the current location of the mouse on the screen. When the user moves the mouse, the mouse pointer moves on the screen. This chapter describes how to create and use mouse pointers and icons in PM applications.

About Mouse Pointers and Icons

Mouse pointers and icons are made up of bit maps that the operating system uses to paint images of the pointers or icons on the screen. A *monochrome bit map* is a series of bytes. Each bit corresponds to a single pel in the image. (The bit map representing the display typically has four bits for each pel.)

A mouse pointer or icon bit map always is twice as tall as it is wide. The top half of the bit map is an *AND* mask, in which the bits are combined, using the AND operator, with the screen bits where the pointer is being drawn. The lower half of the bit map is an *XOR* mask, in which the bits are combined, using the XOR operator, with the destination screen bits.

The combination of the AND and XOR masks results in four possible colors in the bit map. The pels of an icon or pointer can be black, white, transparent (the screen color beneath the pel), or inverted (inverting the screen color beneath the pel). Figure 10-1 shows the relationship of the bit values in the AND and XOR masks:

AND mask	0	0	1	1
XOR mask	0	1	0	1
Result	Black	White	Transparent	Inverted

Figure 10-1. Bit Values in the AND and XOR Masks

Mouse-Pointer Hot Spot

Each mouse pointer has its own *hot spot*, which is the point that represents the exact location of the mouse pointer. This location is defined as an *x* and *y* offset from the lower-left corner of the mouse-pointer bit map.

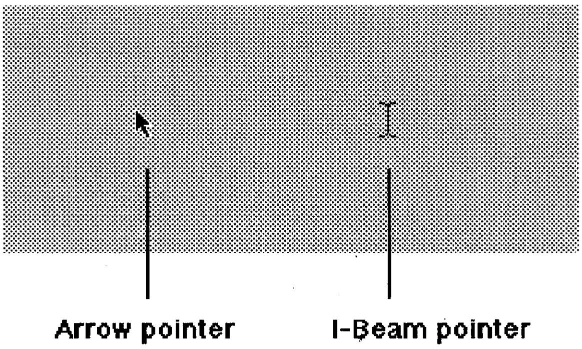


Figure 10-2. Mouse Pointers

For the arrow-shaped pointer, the hot spot is at the tip of the arrow. For the I-beam pointer, the hot spot is at the middle of the vertical line.

Predefined Mouse Pointers

Before an application can use a mouse pointer, it first must receive a handle to the pointer. Most applications load mouse pointers from the system or from their own resource file. The operating system maintains many predefined mouse pointers that an application can use by calling WinQuerySysPointer. System mouse pointers include all the standard mouse-pointer shapes and message-box icons. The following predefined mouse pointers are available:

Table 10-1 (Page 1 of 2). Predefined Mouse Pointers

Mouse Pointer	Description
SPTR_APPICON	Square icon; used to represent a minimized application window.
SPTR_ARROW	Arrow that points to the upper-left corner of the screen.
SPTR_ICONERROR	Icon containing an exclamation point; used in a warning message box.
SPTR_ICONINFORMATION	Octagon-shaped icon containing the image of a human hand; used in a warning message box.
SPTR_ICONQUESTION	Icon containing a question mark; used in a query message box.
SPTR_ICONWARNING	Icon containing an asterisk; used in a warning message box.
SPTR_MOVE	Four-headed arrow; used when dragging an object or window around the screen.
SPTR_SIZE	Small box within a box; used when resizing a window by dragging.

Table 10-1 (Page 2 of 2). Predefined Mouse Pointers

Mouse Pointer	Description
SPTR_SIZENS	Two-headed arrow that points up and down (north and south); used when sizing a window.
SPTR_SIZENESW	Two-headed diagonal arrow that points to the upper-right (northeast) and lower-left (southwest) window borders; used when sizing a window.
SPTR_SIZENWSE	Two-headed diagonal arrow that points to the upper-left (northwest) and lower-right (southeast) window borders; used when sizing a window.
SPTR_SIZEWE	Two-headed arrow that points left and right (west to east); used when sizing a window.
SPTR_TEXT	Text-insertion and selection pointer, often called the *I-beam pointer.*
SPTR_WAIT	Hourglass; used to indicate that a time-consuming operation is in progress.

The operating system contains a second set of predefined mouse pointers that are used as icons in PM applications. An application can use one of these icons by supplying one of the following constants in WinQuerySysPointer. If a copy of the system pointer is made using WinQuerySysPointer, the pointer copy must be destroyed using WinDestroyPointer before termination of the application.

Table 10-2. Presentation Manager Mouse Pointers

Icon	Description
SPTR_FILE	Represents a file (in the shape of a single sheet of paper).
SPTR_FOLDER	Represents a file folder.
SPTR_ILLEGAL	Circular icon containing a slash; represents an illegal operation.
SPTR_MULTFILE	Represents multiple files.
SPTR_PROGRAM	Represents an executable file.

Applications can use mouse-pointer resources to draw icons. WinDrawPointer draws a specified mouse pointer in a specified presentation space. Many of the predefined system mouse pointers are standard icons displayed in message boxes.

In addition to using the predefined pointer shapes, an application also can use pointers that have been defined in a resource file. Once the pointer or icon has been created (by Icon Editor or a similar application), the application includes it in the resource file, using the POINTER statement, a resource identifier, and a file name for the Icon Editor data. After including the mouse-pointer resource, the application can use the pointer or icon by calling WinLoadPointer, specifying the resource identifier and module handle. Typically, the resource is in the executable file of the application, so the application simply can specify NULL for the module handle to indicate the current application resource file.

An application can create mouse pointers at run time by constructing a bit map for the pointer and calling WinCreatePointer. This function, if successful, returns the new pointer handle, which the application then can use to set or draw the pointer. The bit map must be twice as tall as it is wide, with the first half defining the AND mask and the second half defining the XOR mask. The application also must specify the hot spot when creating the mouse pointer.

System Bit Maps

In addition to using the mouse pointers and icons defined by the system, applications can use standard system bit maps by calling WinGetSysBitmap. This function returns a bit map handle that is passed to WinDrawBitmap or to one of the GPI bit-map functions. The system uses standard bit maps to draw portions of control windows, such as the system menu, minimize/maximize box, and scroll-bar arrows. The following standard system bit maps are available:

Table 10-3 (Page 1 of 2). Standard System Bit Maps

Bit Map	Description
SBMP_BTNCORNERS	Specifies the bit map for push button corners.
SBMP_CHECKBOXES	Specifies the bit map for the check-box or radio-button check mark.
SBMP_CHILDSYSMENU	Specifies the bit map for the smaller version of the system-menu bit map; used in child windows.
SBMP_CHILDSYSMENUDEP	Same as SBMP_CHILDSYSMENU but indicates that the system menu is selected.
SBMP_COMBODOWN	Specifies the bit map for the downward pointing arrow in a drop-down combination box.
SBMP_MAXBUTTON	Specifies the bit map for the maximize button.
SBMP_MENUATTACHED	Specifies the bit map for the symbol used to indicate that a menu item has an attached, hierarchical menu.
SBMP_MENUCHECK	Specifies the bit map for the menu check mark.
SBMP_MINBUTTON	Specifies the bit map for the minimize button.
SBMP_OLD_CHILDSYSMENU	Same as SBM_CHILDSYSMENU. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_MAXBUTTON	Same as SBM_MAXBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_MINBUTTON	Same as SBM_MINBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_RESTOREBUTTON	Same as SBM_RESTOREBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBDNARROW	Same as SBM_SBDNARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBLFARROW	Same as SBM_SBLFARROW. (For compatibility with previous versions of the OS/2 operating system.)

Table 10-3 (Page 2 of 2). Standard System Bit Maps

Bit Map	Description
SBMP_OLD_SBRGARROW	Same as SBM_SBRGARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBUPARROW	Same as SBM_SBUPARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_PROGRAM	Specifies the bit map for the symbol that File Manager uses to indicate that a file is an executable program.
SBMP_RESTOREBUTTON	Specifies the bit map for the restore button.
SBMP_RESTOREBUTTONDEP	Same as SBMP_RESTOREBUTTON but indicates that the restore button is pressed.
SBMP_SBDNARROW	Specifies the bit map for the scroll-bar down arrow.
SBMP_SBDNARROWDEP	Same as SBMP_SBDNARROW but indicates that the scroll-bar down arrow is pressed.
SBMP_SBDNARROWDIS	Same as SBMP_SBDNARROW but indicates that the scroll-bar down arrow is disabled.
SBMP_SBLFARROW	Specifies the bit map for the scroll-bar left arrow.
SBMP_SBLFARROWDEP	Same as SBMP_SBLFARROW but indicates that the scroll-bar left arrow is pressed.
SBMP_SBMFARROWDIS	Same as SBMP_SBLFARROW but indicates that the scroll-bar left arrow is disabled.
SBMP_SBRGARROW	Specifies the bit map for the scroll-bar right arrow.
SBMP_SBRGARROWDEP	Same as SBMP_SBRGARROW but indicates that the scroll-bar right arrow is pressed.
SBMP_SBRGARROWDIS	Same as SBMP_SBRGARROW but indicates that the scroll-bar right arrow is disabled.
SBMP_SBUPARROW	Specifies the bit map for the scroll-bar up arrow.
SBMP_SBUPARROWDEP	Same as SBMP_SBUPARROW but indicates that the scroll-bar up arrow is pressed.
SBMP_SBUPARROWDIS	Same as SBMP_SBUPARROW but indicates that the scroll-bar up arrow is disabled.
SBMP_SIZEBOX	Specifies the bit map for the symbol that indicates an area of a window in which the user can click to resize the window.
SBMP_SYSMENU	Specifies the bit map for the system menu.
SBMP_TREEMINUS	Specifies the bit map for the symbol that File Manager uses to indicate an empty entry in the directory tree.
SBMP_TREEPLUS	Specifies the bit map for the symbol that File Manager uses to indicate that an entry in the directory tree contains more files.

Using Mouse Pointers and Icons

This section explains how to perform the following tasks:

- Save the current mouse pointer.
- Change the mouse pointer.
- Restore the original mouse pointer.

Changing the Mouse Pointer

Once you create or load a mouse pointer, you can change its shape by calling WinSetPointer. Following are three typical situations in which an application changes the shape of the mouse pointer:

- When an application receives a WM_MOUSEMOVE message, there is an opportunity to change the mouse pointer based on its location in the window. If you want the standard arrow pointer, pass this message on to WinDefWindowProc. If you want to change the mouse pointer on a standard dialog window, you need to capture the WM_CONTROLPOINTER message and return a pointing-device pointer handle.
- When an application is about to start a time-consuming process during which it will not accept user input, the application displays the *system-wait* mouse pointer (SPTR_WAIT). Upon finishing the process, the application resets the mouse pointer to its former shape.

  The following code fragment shows how to save the current mouse pointer, set the hourglass pointer, and restore the original mouse pointer. Notice that the hourglass pointer also is saved in a global variable so that the application can return it when responding to a WM_MOUSEMOVE message during a time-consuming process.

```
HPOINTER hptrOld, hptrWait, hptrCurrent;

/* Get the current pointer.                   */
hptrOld = WinQueryPointer(HWND_DESKTOP);

/* Get the wait mouse pointer.                */
hptrWait = WinQuerySysPointer(HWND_DESKTOP,
    SPTR_WAIT, FALSE);

/* Save the wait pointer to use in WM_MOUSEMOVE processing.*/
hptrCurrent = hptrWait;

/* Set the mouse pointer to the wait pointer. */
WinSetPointer(HWND_DESKTOP, hptrWait);

/*
 * Do a time-consuming operation, then restore the
 * original mouse pointer.
 */
WinSetPointer(HWND_DESKTOP, hptrOld);
```

- When an application needs to indicate its current operational mode, it changes the pointer shape. For example, a paint program with a palette of drawing tools should change the pointer shape to indicate which drawing tool is in use currently.

Related Functions

This section covers the functions that are related to Mouse Pointers and Icons.

WinCreatePointer

This function creates a pointer from a bit map.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

HPOINTER WinCreatePointer  (HWND hwndDesktop, HBITMAP hbmPointer,
                            BOOL fPointer, LONG xHotspot, LONG yHotspot)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle or HWND_DESKTOP.

hbmPointer (HBITMAP) – input
Bit-map handle from which the pointer image is created.

fPointer (BOOL) – input
Pointer-size indicator.

TRUE	The bit map should be stretched (if necessary) to the system pointer dimensions.
FALSE	The bit map should be stretched (if necessary) to the system icon dimensions.

xHotspot (LONG) – input
x-offset of hot spot within pointer from its lower left corner (in pels).

yHotspot (LONG) – input
y-offset of hot spot within pointer from its lower left corner (in pels).

Returns

hptr (HPOINTER) – returns
Pointer handle.

NULLHANDLE	Error
Other	Handle of the newly created pointer.

WinCreatePointerIndirect

This function creates a colored pointer or icon from a bit map.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HPOINTER WinCreatePointerIndirect  (HWND  hwndDesktop, PPOINTERINFO  pptri)
```

Parameters

hwndDesktop (HWND) – input
Desktop-window handle or HWND_DESKTOP.

pptri (PPOINTERINFO) – input
Pointer information structure.

Returns

hptr (HPOINTER) – returns
Pointer handle.

NULLHANDLE	Error
Other	Handle of the newly created pointer or icon.

WinDestroyPointer

This function destroys a pointer or icon.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinDestroyPointer  (HPOINTER hptrPointer)
```

Parameters

hptrPointer (HPOINTER) – input

Handle of pointer to be destroyed.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinDrawBitmap

This function draws a bit map using the current image colors and mixes.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinDrawBitmap  (HPS hpsDst, HBITMAP hbm, PRECTL pwrcSrc,
                     PPOINTL pptlDst, LONG clrFore, LONG clrBack,
                     ULONG fl)
```

Parameters

hpsDst (HPS) – input
Handle of presentation space in which the bit map is drawn.

hbm (HBITMAP) – input
Bit-map handle.

pwrcSrc (PRECTL) – input
Subrectangle of bit map to be drawn.

NULL	The whole of the bit map is drawn
Other	The whole of the bit map is not drawn.

pptlDst (PPOINTL) – input
Bit-map destination.

clrFore (LONG) – input
Foreground color.

clrBack (LONG) – input
Background color.

fl (ULONG) – input
Flags that determine how the bit map is drawn.

DBM_NORMAL	Draw the bit map normally using ROP_SRCCOPY, as defined in GpiBitBlt.
DBM_INVERT	Draw the bit map inverted using ROP_NOTSRCCOPY, as defined in GpiBitBlt.
DBM_STRETCH	*pptlDst* is used to point to a RECTL data structure representing a rectangle in the destination presentation space, into which the bit map will be stretched or compressed. If compression is required, some rows and columns of the bit map are eliminated.

DBM_HALFTONE Use the OR operator to combine the bit map with an alternating pattern of ones or zeros before drawing it. It can be used with either DBM_NORMAL or DBM_INVERT.

DBM_IMAGEATTRS If this is specified, color conversion of monochrome bit maps is done by using the image attributes.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinDrawPointer

This function draws a pointer in the passed *hps* at the passed coordinates [*lx*, *ly*].

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

BOOL WinDrawPointer (HPS hps, LONG lx, LONG ly, HPOINTER hptrPointer, ULONG ulHalftone)

Parameters

hps (HPS) – input
Presentation-space handle into which the pointer is drawn.

lx (LONG) – input
x-coordinate at which to draw the pointer, in device coordinates.

ly (LONG) – input
y-coordinate at which to draw the pointer, in device coordinates.

hptrPointer (HPOINTER) – input
Pointer handle.

ulHalftone (ULONG) – input
Shading control with which to draw the pointer.

DP_NORMAL	As it normally appears.
DP_HALFTONED	With a halftone pattern where black normally appears.
DP_INVERTED	Inverted, black for white and white for black.
DP_MINIICON	Bit map of a mini icon.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Function failed.

WinGetSysBitmap

This function returns a handle to one of the standard bit maps provided by the system.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

HBITMAP WinGetSysBitmap (HWND hwndDesktop, ULONG ibm)

Parameters

hwndDesktop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

ibm (ULONG) – input

System bit-map index value.

SBMP_SYSMENU	System menu
SBMP_SYSMENUDEP	System menu in depressed state
SBMP_SBUPARROW	Scroll bar up arrow
SBMP_SBUPARROWDEP	Scroll bar up arrow in depressed state
SBMP_SBUPARROWDIS	Scroll bar up arrow in disabled state
SBMP_SBDNARROW	Scroll bar down arrow
SBMP_SBDNARROWDEP	Scroll bar down arrow in depressed state
SBMP_SBDNARROWDIS	Scroll bar down arrow in disabled state
SBMP_SBRGARROW	Scroll bar right arrow
SBMP_SBRGARROWDEP	Scroll bar right arrow in depressed state
SBMP_SBRGARROWDIS	Scroll bar right arrow in disabled state
SBMP_SBLFARROW	Scroll bar left arrow
SBMP_SBLFARROWDEP	Scroll bar left arrow in depressed state
SBMP_SBLFARROWDIS	Scroll bar left arrow in disabled state
SBMP_MENUCHECK	Menu check mark
SBMP_MENUATTACHED	Cascading menu mark
SBMP_CHECKBOXES	Check box or radio button check marks
SBMP_COMBODOWN	Combobox down arrow
SBMP_BTNCORNERS	Push-button corners
SBMP_MINBUTTON	Minimize button
SBMP_MINBUTTONDEP	Minimize button in depressed state
SBMP_MAXBUTTON	Maximize button
SBMP_MAXBUTTONDEP	Maximize button in depressed state
SBMP_RESTOREBUTTON	Restore button
SBMP_RESTOREBUTTONDEP	Restore button in depressed state
SBMP_CHILDSYSMENU	System menu for child windows

SBMP_CHILDSYSMENUDEP	System menu for child windows in depressed state
SBMP_DRIVE	Drive
SBMP_FILE	File
SBMP_FOLDER	Folder
SBMP_TREEPLUS	Used by the file system to indicate that an entry in the directory can be expanded.
SBMP_TREEMINUS	Used by the file system to indicate that an entry in the directory can be collapsed.
SBMP_PROGRAM	Used by the file system to mark .EXE and .COM files.
SBMP_SIZEBOX	Used by some applications to display a sizebox in the bottom-right corner of a frame window.

Returns

hbm (HBITMAP) – returns

System bit-map handle.

NULLHANDLE	Error occurred
Other	System bit-map handle.

WinLoadPointer

This function loads a pointer from a resource file into the system.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HPOINTER WinLoadPointer  (HWND hwndDeskTop, HMODULE Resource,
                          ULONG idPointer)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop window
Other	Desktop-window handle returned by WinQueryDesktopWindow.

Resource (HMODULE) – input
Resource identity containing the pointer definition.

NULLHANDLE	Use the resources file for the application.
Other	Module handle returned by the DosLoadModule or DosQueryModuleHandle call referencing a dynamic-link library containing the resource.

idPointer (ULONG) – input
Identifier of the pointer to be loaded.

Returns

hptr (HPOINTER) – returns
Pointer handle.

NULLHANDLE	Error has occurred
Other	Handle of loaded pointer.

WinQueryPointer

This function returns the pointer handle for *hwndDeskTop*.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HPOINTER WinQueryPointer  (HWND  hwndDeskTop)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

Returns

hptrPointer (HPOINTER) – returns

Pointer handle.

NULLHANDLE	Error occurred.

WinQueryPointerInfo

This function returns pointer information.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryPointerInfo  (HPOINTER hptr, PPOINTERINFO pptriPointerInfo)
```

Parameters

hptr (HPOINTER) – input
Pointer handle.

pptriPointerInfo (PPOINTERINFO) – output
Pointer-information structure.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQueryPointerPos

This function returns the pointer position.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryPointerPos  (HWND  hwndDeskTop, PPOINTL  pptlPoint)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

pptlPoint (PPOINTL) – output

Pointer position in screen coordinates.

Returns

rc (BOOL) – returns

Pointer position returned indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQuerySysPointer

This function returns the system-pointer handle.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

**HPOINTER WinQuerySysPointer (HWND hwndDeskTop, LONG lIdentifier,
BOOL fCopy)**

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

lIdentifier (LONG) – input
System-pointer identifier.

SPTR_ARROW	Arrow pointer
SPTR_TEXT	Text I-beam pointer
SPTR_WAIT	Hourglass pointer
SPTR_SIZE	Size pointer
SPTR_MOVE	Move pointer
SPTR_SIZENWSE	Downward-sloping, double-headed arrow pointer
SPTR_SIZENESW	Upward-sloping, double-headed arrow pointer
SPTR_SIZEWE	Horizontal, double-headed arrow pointer
SPTR_SIZENS	Vertical, double-headed arrow pointer
SPTR_APPICON	Standard application icon pointer
SPTR_ICONINFORMATION	Information icon pointer
SPTR_ICONQUESICON	Question mark icon pointer
SPTR_ICONERROR	Exclamation mark icon pointer
SPTR_ICONWARNING	Warning icon pointer
SPTR_ILLEGAL	Illegal operation icon pointer
SPTR_FILE	Single file icon pointer
SPTR_MULTFILE	Multiple files icon pointer
SPTR_FOLDER	Folder icon pointer
SPTR_PROGRAM	Application program icon pointer

fCopy (BOOL) – input
Copy indicator.

TRUE	Create a copy of the default system pointer and return its handle. Specify this value if the system pointer is to be modified. The application should destroy the copy of the pointer created. This can be done by using the WinDestroyPointer function.
FALSE	Return the handle of the current system pointer.

Returns

hptrPointer (HPOINTER) – returns
Pointer handle.

WinSetPointer

This call sets the desktop-pointer handle.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetPointer  (HWND  hwndDeskTop, HPOINTER  hptrNewPointer)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

hptrNewPointer (HPOINTER) – input
New pointer handle.

NULL	Remove pointer from the screen.
Other	Pointer handle associated with *hwndDeskTop*. Handles for application-defined pointers are returned by the WinLoadPointer and WinCreatePointer calls.

Returns

rc (BOOL) – returns
Pointer-updated indicator.

TRUE	Pointer successfully updated
FALSE	Pointer not successfully updated.

WinSetPointerPos

This function sets the pointer position.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetPointerPos  (HWND  hwndDeskTop, LONG  lx, LONG  ly)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

lx (LONG) – input

x-position of pointer in screen coordinates.

ly (LONG) – input

y-position of pointer in screen coordinates.

Returns

rc (BOOL) – returns

Pointer position updated indicator.

TRUE	Pointer position successfully updated
FALSE	Pointer position not successfully updated.

WinShowPointer

This function adjusts the pointer display level to show or hide a pointer.

Syntax

```
#define INCL_WINPOINTERS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinShowPointer  (HWND  hwndDeskTop, BOOL  fShow)
```

Parameters

hwndDeskTop (HWND) – input

Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	The specified desktop-window handle.

fShow (BOOL) – input

Level-update indicator.

TRUE	Decrement pointer display level by one. (The pointer level is not decremented to a negative value.)
FALSE	Increment pointer display level by one.

Returns

rc (BOOL) – returns

Display-level-updated indicator.

TRUE	Pointer display level not successfully updated.
FALSE	Pointer display level successfully updated

Related Data Structures

This section covers the data structures that are related to Mouse Pointers and Icons.

POINTERINFO

Pointer-information structure.

Syntax

```
typedef struct _POINTERINFO {
ULONG          fPointer;
LONG           xHotSpot;
LONG           yHotSpot;
HBITMAP        hbmPointer;
HBITMAP        hbmColor;
HBITMAP        hbmMiniPointer;
HBITMAP        hbmMiniColor;
 } POINTERINFO;

typedef POINTERINFO *PPOINTERINFO;
```

Fields

fPointer (ULONG)
Bit-map size indicator.

TRUE	Pointer-sized bit map
FALSE	Icon-sized bit map.

xHotSpot (LONG)
X-coordinate of action point.

yHotSpot (LONG)
Y-coordinate of action point.

hbmPointer (HBITMAP)
Bit-map handle of pointer.

hbmColor (HBITMAP)
Bit-map handle of color bit map.

hbmMiniPointer (HBITMAP)
Bit-map handle of a pointer to a mini bit map.

hbmMiniColor (HBITMAP)
Bit-map handle of mini color bit map.

Summary

Following are the OS/2 functions and structure used with mouse pointers, icons, and bit maps.

Table 10-4. Pointer and Bit Map Functions

Function name	Description
WinCreatePointer	Creates a pointer from a bit map.
WinCreatePointerIndirect	Creates a colored pointer or icon from a bit map.
WinDestroyPointer	Destroys a pointer or an icon.
WinDrawBitmap	Draws a bit map using the current image colors and mixes.
WinDrawPointer	Draws a pointer.
WinGetSysBitmap	Returns a handle to one of the standard bit maps provided by the system.
WinLoadPointer	Loads a pointer from a resource file into the system.
WinQueryPointer	Returns the pointer handle for **DeskTop**.
WinQueryPointerInfo	Returns pointer information.
WinQueryPointerPos	Returns the pointer position.
WinQuerySysPointer	Returns the handle of the system pointer.
WinSetPointer	Sets the handle of the Desktop pointer.
WinSetPointerPos	Sets the pointer position.
WinShowPointer	Adjusts the pointer display level to show or hide a pointer.

Table 10-5. Pointer Structure

Structure	Description
POINTERINFO	Pointer information structure.

Chapter 11. Cursors

A *cursor* is a rectangle that can be shown at any location in a window, indicating where the user's next interaction with items on the screen will happen. This chapter describes how to create and use cursors in your PM applications.

About Cursors

Only one cursor appears on the screen at a time—either marking the text-insertion point (a *text cursor*) or indicating which items the user can interact with from the keyboard (a *selection cursor*). For example, when an entry field has the keyboard focus, it displays a blinking vertical bar to show the text-insertion point; however, when a button has the keyboard focus, the cursor appears as a halftone rectangle the size of the button. The operating system draws and blinks the cursor, freeing the application from handling these details. Notice that the cursor has no direct relationship with the mouse pointer.

Cursor Creation and Destruction

The system can use only one cursor at a time, so windows must create and destroy cursors as each windows gains and loses the keyboard focus. If an application attempts to use more than one cursor at a time, the results can be unpredictable and might affect other applications.

An application creates a cursor by calling WinCreateCursor. Generally, this is done when a window gains the keyboard focus. The application specifies the window in which to display the cursor, whether it be the desktop window, an application window, or a control window. An application destroys a cursor by calling WinDestroyCursor— when the specified window loses the keyboard focus, for example.

Position and Size

An application can set the position (in window coordinates) of an existing cursor by calling WinCreateCursor, specifying the CURSOR_SETPOS flag. The cursor width is usually 0 (nominal border width is used) for text-insertion cursors. This is preferable to a value of 1, since such a fine width is almost invisible on a high-resolution monitor. The cursor width also can be related to the window size—for example, when a button control uses a dotted-line cursor around the button text to indicate focus. To change the cursor size, the application must destroy the current cursor and create a new one of the desired size.

Other Cursor Characteristics

An application uses the WinCreateCursor function to specify information about the cursor rectangle and the clipping rectangle. WinCreateCursor specifies whether the cursor rectangle should be filled, framed, blinking, or halftone. In addition, the function specifies the clipping rectangle, in window coordinates, that controls the cursor clipping region. Probably the most efficient strategy is for the application to specify NULL, which causes the rectangle to clip the cursor to the window rectangle.

Cursor Visibility

An application can use the WinShowCursor function to show or hide a cursor. The operating system maintains a *show level* for the cursor: when the cursor is visible, the its show level is zero; each time the cursor is hidden, its show level is incremented; each time the cursor is shown, its show level is decremented. The show:hide relationship is 1:1, so the show level cannot drop below zero. When first creating a cursor, an application should show the cursor because the application creates the cursor with a show level of 1.

The operating system automatically hides the cursor when the application calls WinBeginPaint; it shows the cursor when the application calls WinEndPaint. Therefore, there is no conflict with the cursor during WM_PAINT processing.

Using Cursors

This section explains how to perform the following tasks:

- Create and destroy a cursor.
- Respond to a WM_SETFOCUS message.

Creating and Destroying a Cursor

The following code fragment shows how an application should respond to a WM_SETFOCUS message when using a cursor in a particular window:

```
LONG   curXPos,curYPos,curWidth,curHeight;

case WM_SETFOCUS:
    if (SHORT1FROMMP(mp2)) {

        /* Gain the focus. */
        WinCreateCursor(hwnd, curXPos, curYPos, curWidth, curHeight,
            CURSOR_SOLID | CURSOR_FLASH, (PRECTL) NULL);
        WinShowCursor(hwnd, TRUE);
    }
    else {

        /* Lose the focus. */
        WinDestroyCursor(hwnd);
    }

    return 0;
```

Figure 11-1. Response to a WM_SETFOCUS message

Related Functions

This section covers the functions that are related to Cursors.

WinCreateCursor

This function creates or changes a cursor for a specified window.

Syntax

```
#define INCL_WINCURSORS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinCreateCursor  (HWND hwnd, LONG lx, LONG ly, LONG lcx, LONG lcy,
                       ULONG ulrgf, PRECTL prclClip)
```

Parameters

hwnd (HWND) – input
Handle of window in which cursor is displayed.

lx (LONG) – input
x-position of cursor.

ly (LONG) – input
y-position of cursor.

lcx (LONG) – input
x-size of cursor.

lcy (LONG) – input
y-size of cursor.

ulrgf (ULONG) – input
Controls the appearance of the cursor.

CURSOR_SOLID	The cursor is solid.
CURSOR_HALFTONE	The cursor is halftone.
CURSOR_FRAME	The cursor is a rectangular frame.
CURSOR_FLASH	The cursor flashes.
CURSOR_SETPOS	Set a new cursor position. *lcx* and *lcy* are ignored. Used when a cursor has already been created. In this case, all other appearance flags are ignored.

prclClip (PRECTL) – input
Cursor rectangle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinDestroyCursor

This function destroys the current cursor, if it belongs to the specified window.

Syntax

```
#define INCL_WINCURSORS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinDestroyCursor  (HWND hwnd)
```

Parameters

hwnd (HWND) – input

Window handle to which the cursor belongs.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinQueryCursorInfo

This function obtains information about any current cursor.

Syntax

```
#define INCL_WINCURSORS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinQueryCursorInfo  (HWND hwndDeskTop,
                          PCURSORINFO pcsriCursorInfo)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop-window handle
Other	Specified desktop-window handle.

pcsriCursorInfo (PCURSORINFO) – output
Cursor information.

Returns

rc (BOOL) – returns
Current-cursor indicator.

TRUE	Cursor exists
FALSE	Cursor does not exist, *pcsriCursorInfo* is not updated by this call.

WinShowCursor

This function shows or hides the cursor that is associated with a specified window.

Syntax

```
#define INCL_WINCURSORS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinShowCursor  (HWND hwnd, BOOL fShow)
```

Parameters

hwnd (HWND) – input
Handle of window to which the cursor belongs.

fShow (BOOL) – input
Show indicator.

TRUE	Make cursor visible
FALSE	Make cursor invisible.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred, or an attempt was made to show the cursor when it was already visible.

Related Data Structure

This section covers the data structure related to Cursors.

CURSORINFO

Cursor-information structure.

Syntax

```
typedef struct _CURSORINFO {
HWND        hwnd;
LONG        x;
LONG        y;
LONG        cx;
LONG        cy;
ULONG       fs;
RECTL       rclClip;
 } CURSORINFO;

typedef CURSORINFO *PCURSORINFO;
```

Fields

hwnd (HWND)
Window handle.

x (LONG)
X-coordinate.

y (LONG)
Y-coordinate.

cx (LONG)
Cursor width.

cy (LONG)
Cursor height.

fs (ULONG)
Options.

rclClip (RECTL)
Cursor box.

Summary

Following are the OS/2 functions and structure used with cursors:

Table 11-1. Cursor Functions

Function name	Description
WinCreateCursor	Used to create, set the size of, and move the cursor around the screen.
WinDestroyCursor	Destroys the current cursor if it belongs to the specified window.
WinQueryCursorInfo	Obtains information about any current cursor.
WinShowCursor	Shows or hides the cursor associated with a specified window.

Table 11-2. Cursor Structure

Structure name	Description
CURSORINFO	Cursor information structure.

Chapter 12. Resource Files

Resource files enable you to specify the resource information used in creating an application's window. Some examples of resources that can be defined in resource files are:

- Menus
- Accelerator tables
- Dialog and window templates
- Icons
- Fonts
- Bit maps
- Strings

To add resource information to an application, use a text editor to create a *resource* script file, and then compile it using the *Resource Compiler*, RC.EXE. The advantage of using resource files is that resource information can be maintained and updated separately in the resource script file and then linked to your application program's .EXE file. This greatly simplifies customizing an application because you can modify resource information without having to recompile the entire application.

This chapter describes the use of resource files in Presentation Manager (PM) programming.

About Resource Files

A resource script file is a text file that contains one or more resource statements that define the type, identifier, and data for each resource. Because some resources might contain binary data that cannot be created using a text editor, there are resource statements that let you specify additional files to include when compiling the resource script file. For example, you can use the *Dialog Box Editor* to design dialog boxes, the *Font Editor* to edit font files, and the *Icon Editor* to create customized icons, pointers, and bit maps. The definitions for these resources can be included with other resource definitions in the resource file.

Resource Statements

This section provides overview information on resource statements and directives. Resource statements consist of one or more keywords, numbers, character strings, constants, or file names. You combine these to define the resource type, identifier, and data. Directives are special types of resource statements that perform functions such as including header files, defining constants, and conditionally compiling portions of the file. Resource statements have three basic forms:

- Single-line statements
- Multiple-line statements
- Directives

Single-line Statements

Single-line statements consist of a keyword identifying the resource type, a constant or number specifying the resource identifier, and a file name specifying the file containing the resource data. For example, this ICON statement defines an icon resource:

```
ICON 1 myicon.ico
```

The icon resource has the icon identifier 1, and the file MYICON.ICO contains the icon data.

Multiple-line Statements

Multiple-line statements consist of a keyword identifying the resource type, a constant or number specifying the resource identifier, and, between the BEGIN and END keywords, additional resource statements that define the resource data. For example, this MENU statement defines a menu resource:

```
MENU 1
BEGIN
    MENUITEM "Alpha", 101
    MENUITEM "Beta",  102
END
```

The menu identifier is 1. The menu contains two MENUITEM statements that define the contents of the menu.

In multiple-line statements such as DLGTEMPLATE and WINDOWTEMPLATE, any level of nested statements is allowed. For example, the DLGTEMPLATE and WINDOWTEMPLATE statements typically contain a single DIALOG or FRAME statement. These statements can contain any number of WINDOW and CONTROL statements; the WINDOW and CONTROL statements can contain additional WINDOW and CONTROL statements, and so forth. The nested statements let you define controls and other child windows for the dialog boxes and windows.

If a nested statement creates a child window or control, the parent and owner of the new window is the window created by the containing statement. (FRAME statements occasionally create frame controls whose parent and owner windows are not the same.)

Directives

Directives consist of the reserved character # in the first column of a line, followed by the directive keyword and any additional numbers, character strings, or file names.

Some examples of directives are:

- #define
- #if
- #ifdef
- #include

Descriptions of the individual directives follow the resource file statement descriptions.

Resource File Statement Descriptions

This section provides the syntax, description, and an example of each of the resource file statements.

The following table summarizes, at a general level, the most commonly used parameters on the statements.

Parameter	Description
id	Control identifier.
x	X coordinate of the lower-left corner of the control.
y	Y coordinate of the lower-left corner of the control.
height	Height of the control (in 1/8 character units).
width	Width of the control.
style	Predefined bit representation of a style or combination of styles.
load option	Definition of when the system should load the resource into memory (for example, PRELOAD or LOADONCALL).
mem option	Definition of how the system manages the resource when in memory (for example, FIXED, MOVABLE, or DISCARDABLE).
text	Text associated with a control.
class	Predefined class for a particular control.

ACCELTABLE Statement

The ACCELTABLE statement creates a table of accelerators for an application.

Syntax

```
ACCELTABLE acceltable-id [mem-option][load-option]
BEGIN
key-value, command[, accelerator-options] ...
    .
    .
    .
END
```

Description: An accelerator is a keystroke that gives the user a quick way to choose a command from a menu or carry out some other task. An accelerator table can be loaded when needed from the executable file by using the WinLoadAccelTable function.

Example: This example creates an accelerator table whose accelerator-table identifier is 1. The table contains two accelerators: Ctrl+S and Ctrl+G. These accelerators generate WM_COMMAND messages with values of 101 and 102, respectively, when the user presses the corresponding keys.

```
ACCELTABLE 1
BEGIN
    "S", 101, CONTROL
    "G", 102, CONTROL
END
```

ASSOCTABLE Statement

The ASSOCTABLE statement defines a file-association table for an application.

Syntax

```
ASSOCTABLE assoctable-id [load-option][mem-option]
BEGIN
association-name, file-match-string[, extended-attribute-flag]
  [, icon-filename]
  .
  .
  .
END
```

Description: This table associates the data files that an application creates with the executable file of the application. When the user selects one of these data files, the associated application begins executing.

A file-association table can also associate icons with the data files that an application creates. The icons are used to identify the data files graphically. Because a file-association table associates icons by file type, all data files having the same file type have the same icon.

You can provide any number of ASSOCTABLE statements in a resource script file, but each statement must specify a unique assoctable-id value. The file-association tables are written not only to the resources within your executable file, but also to the .ASSOC extended attribute. However, only the last file-association table specified in the resource script file is actually written to the extended attribute.

AUTOCHECKBOX Statement

The AUTOCHECKBOX statement creates an automatic-check-box control.

Syntax

```
AUTOCHECKBOX text, id, x, y, width [, style]
```

Description: The control is a small rectangle (check box) that contains an X when the user selects it. The specified text is displayed to the right of the check box. An X appears in the square when the user first selects the control and disappears the next time the user selects it. The AUTOCHECKBOX statement, which can only be used in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If the style is not specified, the default style is BS_AUTOCHECKBOX and WS_TABSTOP.

Example: This example creates an automatic-check-box control that is labeled "Italic."

```
AUTOCHECKBOX "Italic", 101, 10, 10, 100, 100
```

AUTORADIOBUTTON Statement

The AUTORADIOBUTTON statement creates an automatic-radio-button control.

Syntax

```
AUTORADIOBUTTON text, id, x, y, width, height [, style]
```

Description: This control is a small circle with the given text displayed to its right. The control highlights the circle and sends a message to its parent window when the user selects the button. The control also removes the selection from any other automatic-radio-button controls in the same group. When the user selects the button again, the control removes the highlight before sending a message. The AUTORADIOBUTTON statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_AUTORADIOBUTTON.

Example: This example creates an automatic-radio-button control that is labeled "Italic."

```
AUTORADIOBUTTON "Italic", 101, 10, 10, 24, 50
```

BITMAP Statement

The BITMAP statement defines a bit-map resource for an application.

Syntax

```
BITMAP bitmap-id [load-option] [mem-option] filename
```

Description: A bit-map resource, typically created using the Icon Editor, is a custom bit map that an application uses in its display or as an item in a menu.

The BITMAP statement copies the bit-map resource from the file specified in the *filename* field and adds it to the application's other resources. A bit-map resource can be loaded from the executable file when needed by using the GpiLoadBitmap function.

You can provide any number of BITMAP statements in a resource script file, but each statement must specify a unique bitmap-id value.

Example: This example defines a bit map whose bit-map identifier is 12. The bit-map resource is copied from the file CUSTOM.BMP.

```
BITMAP 12 custom.bmp
```

CHECKBOX Statement

The CHECKBOX statement creates a check-box control.

Syntax

```
CHECKBOX text, id, x, y, width, height [, style]
```

Description: The control is a small rectangle (check box) that has the specified text displayed to the right. The control highlights the rectangle and sends a message to its parent window when the user selects the control. The CHECKBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_CHECKBOX and WS_TABSTOP.

Example: This example creates a check-box control that is labeled “Italic.”

```
CHECKBOX "Italic", 101, 10, 10, 100, 100
```

CODEPAGE Statement

The CODEPAGE statement sets the code page for all subsequent resources.

Syntax

```
CODEPAGE codepage-id
```

Description: The code page specifies the character set used for characters in the resource.

If the CODEPAGE statement is not given in a resource script file, the resource compiler uses the code page set up for the individual system. If more than one CODEPAGE statement is given in the file, each CODEPAGE statement applies to the resource statements between it and the next CODEPAGE statement.

Example: In this example, the code page for the character-string resources is set to Portuguese (860).

```
CODEPAGE 860

STRINGTABLE
BEGIN
    1 "Filename not found"
    2 "Cannot open file for reading"
END
```

COMBOBOX Statement

The COMBOBOX statement creates a combination-box control.

Syntax

```
COMBOBOX text, id, x, y, width, height [, style]
```

Description: This control combines a list-box control with an entry-field control. It allows the user to place the selected item from a list box into an entry field.

The COMBOBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_COMBOBOX. If you do not specify a style, the default style is CBS_SIMPLE, WS_GROUP, WS_TABSTOP, and WS_VISIBLE.

Example: This example creates a combination-box control.

```
COMBOBOX "", 101, 10, 10, 24, 50
```

CONTAINER Statement

The CONTAINER statement creates a container control within a dialog window.

Syntax

```
CONTAINER  id, x, y, width, height [, style]
```

Description: The container control is a visual component that holds objects.

The CONTAINER statement defines the identifier, position, dimensions, and attributes of a container control. The predefined class for this control is WC_CONTAINER. If you do not specify a style, the default style is WS_TABSTOP, WS_VISIBLE, and CCS_SINGLESEL. A CONTAINER statement is only used in a DIALOG or WINDOW statement.

Example: This example creates a container control at position (30,30) within the dialog window. The container has a width of 70 character units and a height of 25 character units. Its resource ID is 301. The default style CCS_SINGLESEL has been overridden by the style specification CCS_MULTIPLESEL. The default styles WS_TABSTOP and WS_GROUP are both in effect, though only the latter is specified.

```
#define IDC_CONTAINER    301
#define IDD_CONTAINERDLG 504
DIALOG "Container", IDD_CONTAINERDLG, 23, 6, 120, 280, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
     CONTAINER   IDC_CONTAINER, 30, 30, 70, 200, CCS_MULTIPLESEL |
                           WS_GROUP
  END
```

CONTROL Statement

The CONTROL statement defines a control as belonging to the specified class.

Syntax

```
CONTROL text, id, x, y, width, height, class [, style]
[ data-definitions ] ...
[ BEGIN
control-definition
    .
    .
    .
END ]
```

Description: The statement defines the position and dimensions of the control within the parent window, as well as the control style. The CONTROL statement is most often used in a DIALOG or WINDOW statement.

Typically, several CONTROL statements are used in each DIALOG statement, and each CONTROL statement must have a unique *id* value. The optional BEGIN and END statements enclose any CONTROL statements that may be given with the control. CONTROL statements given in this manner represent child windows belonging to the control created by the CONTROL statement.

The CONTROL statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements, but it usually does not contain such statements.

Example: This example creates a push-button control with the WS_TABSTOP and WS_VISIBLE styles.

```
CONTROL "OK", 101, 10, 10, 20, 50, WC_BUTTON, BS_PUSHBUTTON |
                                              WS_TABSTOP    |
                                              WS_VISIBLE
```

CTEXT Statement

The CTEXT statement creates a centered-text control.

Syntax

```
CTEXT text, id, x, y, width, height [, style]
```

Description: The control is a simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The CTEXT statement defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_CENTER, and WS_GROUP.

Use the CTEXT statement only in a DIALOG or WINDOW statement.

Example: This example creates a centered-text control that is labeled “Filename.”

```
CTEXT "Filename", 101, 10, 10, 100, 100
```

CTLDATA Statement

The CTLDATA statement defines control data for a custom dialog box, window, or control.

Syntax

```
CTLDATA word-value [, word-value] ...

CTLDATA string

CTLDATA MENU
BEGIN
menuitem-definition
    .
    .
    .
END
```

Description: The statement has three basic forms to permit specifying a menu or specifying data in words or characters. The data can be in any format, because only your window procedure will use it. The window procedure of the dialog box, window, or control receives this data when the item is created. It is up to the window procedure to process the data.

CTLDATA is often used to supply data that controls the subsequent operation of the custom window. For example, the CTLDATA statement may contain extended style bits – that is, style bits designed specifically for your customized window.

You should reserve the CTLDATA statement for window classes that you create yourself.

Example: This example creates a menu for the window created with the WINDOW statement.

```
WINDOWTEMPLATE 1
BEGIN
    WINDOW "Sample", 1, 0, 0, 100, 100, "MYCLASS", 0, FCF_STANDARD
    CTLDATA MENU
    BEGIN
        MENUITEM "Exit", 101
    END
END
```

DEFAULTICON STATEMENT

This statement installs the named icon file definition under the ICON Extended Attribute of the program file.

Syntax

```
DEFAULTICON filename
```

Description: An icon with an icon-id of 1 is the default icon by default, unless you supply a different icon.

Example: DEFAULTICON filename.ico

DEFPUSHBUTTON Statement

The DEFPUSHBUTTON statement creates a default push-button control.

Syntax

```
DEFPUSHBUTTON text, id, x, y, width, height [, style]
```

Description: The control is a round-cornered rectangle containing the given text. The rectangle has a bold outline to represent that it is the default response for the user. The control sends a message to its parent window when the user chooses the control. The DEFPUSHBUTTON statement defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_PUSHBUTTON, BS_DEFAULT, and WS_TABSTOP.

Use the DEFPUSHBUTTON statement only in a DIALOG or WINDOW statement.

Example: This example creates a default push-button control that is labeled “Cancel.”

```
DEFPUSHBUTTON "Cancel", 101, 10, 10, 24, 50
```

DIALOG Statement

The DIALOG statement defines a window that an application can use to create dialog boxes.

Syntax

```
DIALOG text, id, x, y, width, height [, [style] [,framectl]] [data-definitions]
BEGIN
control-definition
    .
    .
    .
END
```

Description: The statement defines the position and dimensions of the dialog box on the screen, as well as the dialog-box style. The DIALOG statement is most often used in a DLGTEMPLATE statement.

Typically, you use only one DIALOG statement in each DLGTEMPLATE statement, and the DIALOG statement contains at least one control definition.

The exact meaning of the coordinates depends on the style defined by the *style* field. For dialog boxes with FS_SCREENALIGN style, the coordinates are relative to the origin of the display screen. For dialog boxes with the style FS_MOUSEALIGN, the coordinates are relative to the position of the mouse pointer at the time the dialog box is created. For all other dialog boxes, the coordinates are relative to the origin of the parent window.

The DIALOG statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a DIALOG statement contains one or more CONTROL statements.

Example: This example creates a dialog box that is labeled "Disk Error."

```
DLGTEMPLATE 1
BEGIN
    DIALOG  "Disk Error", 100, 10, 10, 300, 110
    BEGIN
        CTEXT "Select One:", 1, 10, 80, 280, 12
        RADIOBUTTON "Retry", 2, 75, 50, 60, 12
        RADIOBUTTON "Abort", 3, 75, 30, 60, 12
        RADIOBUTTON "Ignore", 4, 75, 10, 60, 12
    END
END
```

DLGINCLUDE Statement

The DLGINCLUDE statement adds the specified file name to the resource file.

Syntax

```
DLGINCLUDE id filename
```

Description: The DLGINCLUDE statement is typically used to let the application access the definitions file for the dialog box with the corresponding identifier. The file specified in the *filename* field must contain the define directives used by the dialog box.

You can provide any number of DLGINCLUDE statements in a resource script file, but each must have a unique identifier.

Example: This example includes the name of the definition file dlgdef.h. The dialog-box identifier is 5.

```
DLGINCLUDE 5 \\INCLUDE\\DLGDEF.H
```

DLGTEMPLATE Statement

The DLGTEMPLATE statement creates a dialog-box template.

Syntax

```
DLGTEMPLATE dialog-id [load-option] [mem-option]
BEGIN
dialog-definition
     .
     .
     .
END
```

Description: A dialog-box template consists of a series of statements that define the identifier, load and memory options, dialog-box dimensions, and controls in the dialog box. The dialog-box template can be loaded from the executable file by using the WinLoadDlg function.

You can provide any number of dialog-box templates in a resource script file, but each template must have a unique dialog-id value.

A DLGTEMPLATE statement can actually contain DIALOG, CONTROL, and WINDOW statements. Typically, you include only one DIALOG statement.

Example: This example uses a DLGTEMPLATE statement to create a dialog box.

```
DLGTEMPLATE ID_GETTIMER
BEGIN
    DIALOG "Timer", 1, 10, 10, 100, 40
    BEGIN
        LTEXT "Time (0 - 15):", 4, 8, 24, 72, 12
        ENTRYFIELD "0", ID_TIME, 80, 28, 16, 8, ES_MARGIN
        DEFPUSHBUTTON "Enter", ID_TIMEOK, 10, 6, 36, 12
        PUSHBUTTON "Cancel", ID_TIMECANCEL, 52, 6, 40, 12
    END
END
```

EDITTEXT Statement

The EDITTEXT statement creates an entry-field control.

Syntax

```
EDITTEXT text, id, x, y, width, height [, style]
```

Description: This control is a rectangle in which the user can type and edit text. The control displays a pointer when the user selects the control. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the characters to delete or select the place to insert new characters.

The EDITTEXT statement defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_ENTRYFIELD. If you do not specify a style, the default style is ES_AUTOSCROLL and WS_TABSTOP.

The EDITTEXT control statement is identical to the ENTRYFIELD control statement. Use the EDITTEXT statement only in a DIALOG or WINDOW statement.

Example: This example creates an entry-field control that is not labeled.

```
   EDITTEXT "", 101, 10, 10, 24, 50
```

ENTRYFIELD Statement

The ENTRYFIELD statement creates an entry-field control.

Syntax

```
ENTRYFIELD text, id, x, y, width, height [, style]
```

Description: This control is a rectangle in which the user can type and edit text. The control displays a pointer when the user selects the control. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the characters to delete or select the place to insert new characters. The ENTRYFIELD statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_ENTRYFIELD. If you do not specify a style, the default style is ES_AUTOSCROLL and WS_TABSTOP.

Example: This example creates an entry-field control that is not labeled.

```
ENTRYFIELD "", 101, 10, 10, 24, 50
```

FONT Statement

The FONT statement defines a font resource for an application.

Syntax

```
FONT font-id  [load-option] [mem-option] filename
```

Description: A font resource, typically created by using the OS/2 Font Editor, is a bit map defining the shape of the individual characters in a character set. The FONT statement copies the font resource from the file specified in the *filename* field and adds it to the other resources of the application. A font resource can be loaded from the executable file when needed by using the GpiLoadFonts function.

You can provide any number of FONT statements in a resource script file, but each statement must specify a unique font-id value.

Example: This example defines a font whose font identifier is 5. The font resource is copied from the file cmroman.fon.

```
FONT 5 cmroman.fon
```

FRAME Statement

The FRAME statement defines a frame window.

Syntax

```
FRAME text, id, x, y, width, height, style [, framectl]
  data-definitions
[ BEGIN
window-definition
     .
     .
     .
END ]
```

Description: The statement defines the title, identifier, position, and dimensions of the frame window, as well as the window style. The FRAME statement is most often used in a WINDOWTEMPLATE statement and, typically, only one FRAME statement is used. The FRAME statement, in turn, typically contains at least one WINDOW statement that defines the client window belonging to the frame window.

The frame window has no default style. You must use the *framectl* field to define additional frame controls, such as a title bar and system menu, to be created when the frame window is created. If the text field is not empty, the statement automatically adds a title-bar control to the frame window, whether or not you specify the FCF_TITLEBAR style. Frame controls are given default styles and control identifiers, depending on their class. For example, a title-bar control receives the identifier FID_TITLEBAR.

The FRAME statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a FRAME statement contains one WINDOW statement.

Example: This example creates a standard frame window with a title bar, a system menu, minimize and maximize boxes, and a vertical scroll bar. The FRAME statement contains a WINDOW statement defining the client window belonging to the frame window.

```
WINDOWTEMPLATE 1
BEGIN
    FRAME "My Window", 1, 10, 10, 320, 130, 0,
            FCF_STANDARD | FCF_VERTSCROLL
    BEGIN
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "MyClientClass"
    END
END
```

GROUPBOX Statement

The GROUPBOX statement creates a group-box control.

Syntax

```
GROUPBOX text, id, x, y, width, height [, style]
```

Description: The control is a rectangle that groups other controls together by drawing a border around them and displaying the given text in the upper-left corner.

The GROUPBOX statement defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_GROUPBOX and WS_TABSTOP.

Use the GROUPBOX statement only in a DIALOG or WINDOW statement.

Example: This example creates a group-box control that is labeled “Options.”

```
GROUPBOX "Options", 101, 10, 10, 100, 100
```

HELPITEM Statement

The HELPITEM statement defines the help items in a help table.

Syntax

```
HELPITEM application-window-id, help-subtable-id, extended-helppanel-id
```

Description: This statement specifies the resource identifier of an application window for which help is provided, along with the resource identifiers of the help subtable and extended help panel associated with the application window.

You can provide any number of HELPITEM statements in a HELPTABLE statement. You should provide one HELPITEM statement for each application window for which help is provided.

Use the HELPITEM statement only in a HELPTABLE statement.

Example: This example defines a help item that associates a help subtable called IDSUB_FILEMENU and an extended help panel called IDEXT_APPHLP with an application window called IDWIN_FILEMENU.

```
HELPITEM IDWIN_FILEMENU, IDSUB_FILEMENU, IDEXT_APPHLP
```

HELPSUBITEM Statement

The HELPSUBITEM statement defines the help subitems in a help subtable.

Syntax

```
HELPSUBITEM child-window-id, helppanel-id [, integer] ...
```

Description: This statement specifies the identifier of a child window for which help is provided, the identifier of the help panel associated with the child window, and one or more optional, application-defined integers.

You can provide any number of HELPSUBITEM statements in a HELPSUBTABLE statement. You should provide one HELPSUBITEM statement for each child window for which help is provided.

Use the HELPSUBITEM statement only in a HELPSUBTABLE statement.

Example: This example defines a help subitem that associates a child window called IDCLD_FILEMENU with a help panel called IDHP_FILEMENU.

```
HELPSUBITEM IDCLD_FILEMENU, IDHP_FILEMENU
```

HELPSUBTABLE Statement

The HELPSUBTABLE statement defines the contents of a help-subtable resource.

Syntax

```
HELPSUBTABLE helpsubtable-id
 [SUBITEMSIZE size]
BEGIN
helpsubitem-definition
    .
    .
    .
END
```

Description: A help-subtable resource contains a help-subitem entry for each item that can be selected in an application window. Each of these items should be a child window of the application window specified in the help-table resource. The help subtable should contain a help subitem for each control, child window, and menu item in the application window.

You can provide any number of HELPSUBTABLE statements in a resource script file, but each statement must specify a unique helpsubtable-id value. You can also provide any number of helpsubitem-definition statements in the help subtable. These specify the child window for which help is provided, the help panel containing the help text for the child window, and one or more application-defined integers.

If you include optional integers in the helpsubitem-definition statements, you must also include a SUBITEMSIZE statement to specify the size, in words, of each help subitem. All help subitems in a help subtable must be the same size. The default size is two words per help subitem.

Example: This example creates a help-subtable resource whose help-subtable identifier is IDSUB_FILEMENU. Each HELPSUBITEM statement specifies a child window and a help panel.

```
HELPSUBTABLE IDSUB_FILEMENU
BEGIN
    HELPSUBITEM IDCLD_OPEN, IDPNL_OPEN
    HELPSUBITEM IDCLD_SAVE, IDPNL_SAVE
END
```

HELPTABLE Statement

The HELPTABLE statement defines the contents of a help-table resource.

Syntax

```
HELPTABLE helptable-id
BEGIN
helpitem-definition
    .
    .
    .
END
```

Description: A help-table resource contains a help-item entry for each application window, dialog box, and message box for which help is provided.

You can provide any number of HELPTABLE statements in a resource script file, but each statement must specify a unique helptable-id value. You can also provide any number of helpitem-definition statements in the help table. These statements specify the application windows for which help is provided, the help subtables associated with each application window, and the extended help panels associated with each application window.

Example: This example creates a help-table resource whose help-table identifier is 1. Each HELPITEM statement specifies an application window, a help subtable, and an extended help panel.

```
HELPTABLE 1
BEGIN
    HELPITEM IDWIN_FILEMENU, IDSUB_FILEMENU, IDEXT_APPHLP
    HELPITEM IDWIN_EDITMENU, IDSUB_EDITMENU, IDEXT_APPHLP
END
```

ICON Statement (Resource)

This form of the ICON statement defines an icon resource for an application.

Syntax

```
ICON icon-id [load-option] [mem-option] filename
```

Description: An icon resource, typically created by using the Icon Editor, is a bit map defining the shape of the icon to be used for a given application. The ICON statement copies the icon resource from the file specified in the *filename* field and adds it to the application's other resources. An icon resource can be loaded when creating a window by using the WinCreateStdWindow function with the FS_ICON style.

You can provide any number of ICON statements in a resource script file, but each statement must specify a unique icon-id value.

An icon with an icon-id of 1 is the default icon. The RC program writes the icon not only to the resources in your executable file but also as the .ICON extended attribute.

Example: This example defines an icon whose icon identifier is 11. The icon resource is copied from the file custom.ico.

```
ICON 11 custom.ico
```

ICON Statement (Control)

This form of the ICON statement creates an icon control.

Syntax

```
ICON icon-id, id, x, y, width, height [, style]
```

Description: This control is an icon displayed in a dialog box. The ICON statement defines the icon-resource identifier, icon-control identifier, and position and attributes of a control window. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_ICON. For the ICON statement, the *width* and *height* fields are ignored; the icon automatically sizes itself.

Use the ICON statement only in a DIALOG or WINDOW statement.

Example: This example creates an icon control whose icon identifier is 99.

```
ICON 99, 101, 10, 10, 0, 0
```

LISTBOX Statement

The LISTBOX statement creates commonly-used controls for a dialog box or window.

Syntax

```
LISTBOX id, x, y, width, height [, style]
```

Description: The control is a rectangle containing a list of user-selectable strings, such as file names.

The LISTBOX statement defines the identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_LISTBOX. If you do not specify a style, the default style is WS_TABSTOP.

Use the LISTBOX statement only in a DIALOG or WINDOW statement.

Example: This example creates a list-box control whose identifier is 101.

```
LISTBOX 101, 10, 10, 100, 100
```

LTEXT Statement

The LTEXT statement creates a left-aligned text control.

Syntax

```
LTEXT text, id, x, y, width, height [, style]
```

Description: The control is a simple rectangle displaying the given text left-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The LTEXT statement defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_LEFT, and WS_GROUP.

Use the LTEXT statement only in a DIALOG or WINDOW statement.

Example: This example creates a left-aligned text control that is labeled "Filename."

```
LTEXT "Filename", 101, 10, 10, 100, 100
```

MENU Statement

The MENU statement defines the contents of a menu resource.

Syntax

```
MENU menu-id [load-option] [mem-option]
BEGIN
menuitem-definition
    .
    .
    .
END
```

Description: A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user choose commands from a list of command names. A menu resource can be loaded from the executable file when needed by using the WinLoadMenu function.

You can provide any number of MENU statements in a resource script file, but each statement must specify a unique menu-id value. You can provide any number of menuitem-definition statements in the menu. These define the submenus and menu items (commands) in the menu. The order of the statements defines the order of the menu items.

Example: This example creates a menu resource whose menu identifier is 1. The menu contains a menu item named Alpha and a submenu named Beta. The submenu contains two menu items: Item 1 and Item 2.

```
=
MENU 1
BEGIN
    MENUITEM "Alpha", 100
    SUBMENU "Beta", 101
    BEGIN
        MENUITEM "Item 1", 200
        MENUITEM "Item 2", 201, , MIA_CHECKED
    END
END
```

MENUITEM Statement

The MENUITEM statement creates a menu item for a menu.

Syntax

```
MENUITEM text, menu-id [, menuitem-style [, menuitem-attributerbrk.]
```

Description:

This statement defines the text, identifier, and attributes of a menu item. Use the MENUITEM statement only in a MENU or SUBMENU statement.

The system displays the text when it displays the corresponding menu. If the user chooses the menu item, the system generates a WM_COMMAND message that includes the specified menu-item identifier and sends it to the window owning the menu.

You can provide any number of MENUITEM statements, but each must have a unique menu-id value.

The alternative form of the MENUITEM statement, MENUITEM SEPARATOR, creates a menu separator. A menu separator is a horizontal dividing bar between two menu items in a submenu. The separator is not active – that is, the user cannot choose it, it has no text associated with it, and it has no identifier.

You can use the \t or \a character combination in any item name. The \t character inserts a tab when the name is displayed and is typically used to separate the menu-item name from the name of an accelerator key. The \a character aligns to the right all text that follows it. These characters are intended to be used for menu items in submenus only. The width of the displayed submenu is always adjusted so there is at least one space (and usually more) between any pieces of text separated by a \t or \a. (When compiling the menu resource, the compiler stores the \t and \a characters as control characters. For example, the \t is stored as 0x09.)

A tilde (˜) character in the item name indicates that the following character is used as a mnemonic character for the item. When the menu is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the menu item by pressing the key corresponding to the underlined mnemonic character.

Example: This example creates a menu item named Alpha. The item identifier is 101.

```
MENUITEM "Alpha", 101
```

This example creates a menu item named Beta. The item identifier is 102. The menu item has a text style and a checked attribute.

```
MENUITEM "Beta", 102, MIS_TEXT, MIA_CHECKED
```

This example creates a menu separator between menu items named Gamma and Delta.

```
MENUITEM "Gamma", 103
MENUITEM SEPARATOR
MENUITEM "Delta", 104
```

This example creates a menu item that has a bit map instead of a name. The bit-map identifier, 1, is first defined using a BITMAP statement. The identifier for the menu item is 301. Note that a # sign must be placed in front of the bit map identifier in the MENUITEM statement.

```
BITMAP 1 mybitmap.bmp

MENUITEM "#1", 301, MIS_BITMAP
```

MESSAGETABLE Statement

The MESSAGETABLE statement creates one or more string resources for an application.

Syntax

```
MESSAGETABLE [load-option] [mem-option]
BEGIN
string-id string-definition
    .
    .
    .
END
```

Description: A string resource is a null-terminated character string that has a unique string identifier. A string resource can be loaded from the executable file when needed by using the DosGetResource or DosGetResource2 function with the RT_MESSAGE resource type.

You can provide any number of MESSAGETABLE statements in a resource script file. The compiler treats all the strings from the various MESSAGETABLE statements as if they belonged to a single statement. This means that no two strings in a resource script file can have the same string identifier.

Although the MESSAGETABLE and STRINGTABLE statements are nearly identical, most applications use the STRINGTABLE statement instead of the MESSAGETABLE statement to create string resources.

You can continue a string on multiple lines by terminating the line with a backslash (\) or by terminating the line with a double quotation mark (") and then starting the next line with a double quotation mark.

Example: This example creates two string resources whose string identifiers are 1 and 2.

```
MESSAGETABLE
BEGIN
    1 "Filename not found"
    2 "Cannot open file for reading"
END
```

MLE Statement

The MLE statement creates a multiple-line entry-field control.

Syntax

```
MLE text, id, x, y, width, height [, style]
```

Description: The control is a rectangle in which the user can type and edit multiple lines of text. The control displays a pointer when the user selects it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the characters to delete or select the place to insert new characters. The MLE statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_MLE. If you do not specify a style, the default style is MLS_BORDER, WS_GROUP, and WS_TABSTOP.

Example: This example creates a multiple-line entry-field control that is not labeled.

```
MLE "", 101, 10, 10, 50, 100
```

NOTEBOOK Statement

The NOTEBOOK statement creates a notebook control within the dialog window.

Syntax

```
NOTEBOOK   id, x, y, width, height [, style]
```

Description: This control is used to organize information on individual pages so that it can be located and displayed easily. The NOTEBOOK statement defines the identifier, position, dimensions, and attributes of a notebook control. The predefined class for this control is

WC_NOTEBOOK. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

Use the NOTEBOOK statement only in a DIALOG or WINDOW statement.

Example: This example creates a notebook control at position (20, 20) within the dialog window. The notebook has a width of 200 character units and a height of 50 character units. Its resource ID is 201. The tabs style BKS_ROUNDEDTABS specification overrides the notebook default style of square tabs. The default styles WS_TABSTOP and WS_GROUP are both in effect, although only the latter is specified.

```
#define    IDC_NOTEBOOK     201
#define    IDD_NOTEBOOKDLG  503
DIALOG "Notebook", IDD_NOTEBOOKDLG, 11, 11, 420, 420, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    NOTEBOOK   IDC_NOTEBOOK, 20, 20, 200, 400, BKS_ROUNDEDTABS | WS_GROUP
  END
```

POINTER Statement

The POINTER statement defines a pointer resource for an application.

Syntax

```
POINTER pointer-id [load-option] [mem-option] filename
```

Description: A pointer resource, typically created by using the OS/2 Icon Editor, is a bit map defining the shape of the mouse pointer on the screen. The POINTER statement copies the pointer resource from the file specified in the *filename* field and adds it to the application's other resources. A pointer resource can be loaded from the executable file when needed by using the WinLoadPointer function.

You can provide any number of POINTER statements in a resource script file, but each statement must specify a unique pointer-id value.

Example: This example defines a pointer whose pointer identifier is 10. The pointer resource is copied from the file custom.cur.

```
POINTER 10 custom.cur
```

PRESPARAMS Statement

The PRESPARAMS statement defines presentation fields that customize a dialog box, menu, window, or control.

Syntax

```
PRESPARAMS presparam, value [, value]  ...
```

Description: PRESPARAMS data is a series of types and values. The window procedure of the dialog box, menu, window, or control receives and processes this data when the item is created. The data for custom controls can be in any format.

PRESPARAMS is often used to supply data to control the appearance of the customized window when it is first created. For example, the PRESPARAMS statement may specify the colors to be used in the window.

Example: This example creates a menu resource with a menu identifier of 1. The PRESPARAMS statement specifies that the following three menu items be displayed in the 12-point Helvetica** font.

```
MENU 1
BEGIN
    PRESPARAMS PP_FONTNAMESIZE, "12.Helv"
    MENUITEM "New", 100
    MENUITEM "Open", 101
    MENUITEM "Save", 102
END
```

PUSHBUTTON Statement

The PUSHBUTTON statement creates a push-button control.

Syntax

```
PUSHBUTTON text, id, x, y, width, height [, style ]
```

Description: The control is a round-cornered rectangle containing the given text. The control sends a message to its parent whenever the user chooses the control. The PUSHBUTTON statement defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_PUSHBUTTON and WS_TABSTOP.

Use the PUSHBUTTON statement only in a DIALOG or WINDOW statement.

Example: This example creates a push-button control that is labeled "OK."

```
PUSHBUTTON "OK", 101, 10, 10, 100, 100
```

RADIOBUTTON Statement

The RADIOBUTTON statement creates a radio-button control, which is a small circle that has the given text displayed to its right.

Syntax

```
RADIOBUTTON text, id, x, y, width, height [, style]
```

Description: The control highlights the circle and sends a message to its parent window when the user selects the button. The control removes the highlight and sends a message when the button is next selected. The RADIOBUTTON statement defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_RADIOBUTTON.

Use the RADIOBUTTON statement only in a DIALOG or WINDOW statement.

Example: This example creates a radio-button control that is labeled "Italic."

```
RADIOBUTTON "Italic", 101, 10, 10, 24, 50
```

RCDATA Statement

The RCDATA statement defines a custom-data resource for an application.

Syntax

```
RCDATA resource-id
BEGIN
data-definition [, data-definition]   ...
    .
    .
    .
END
```

Description: The custom data can be in whatever format the application requires. You can provide any number of RCDATA statements in a resource script file, but each statement must specify a unique resource-id value. A custom-data resource can be loaded from the executable file when needed by using the DosGetResource or DosGetResource2 functions with the RT_RCDATA resource type.

Example: This example defines custom data that has a resource identifier of 5.

```
RCDATA 5
BEGIN
    "E. A. Poe", 1849, -32, 3L, 0x80000001, 3+4+5
END
```

RCINCLUDE Statement

The RCINCLUDE statement causes RC to process the resource script file specified in the *filename* field along with the current resource script file.

Syntax

```
RCINCLUDE filename
```

Description: The contents of both script files are compiled by RC and the results are placed in one binary resource file and/or executable file.

RCINCLUDE statements are processed before any other processing is done, including preprocessing by RCPP.EXE, which removes comments, replaces values in the define directives, and so forth.

When specifying a high performance file system (HPFS) file name on an RCINCLUDE statement, enclose the path and file name in double quotes; for example:

```
   RCINCLUDE "d:\project\long dialog.dlg"
```

Double quotes enable the resource compiler to recognize a name containing embedded blank characters.

Example: This example includes the file DIALOGS.RC as part of the current resource script file.

```
RCINCLUDE dialogs.rc
```

RESOURCE Statement

The RESOURCE statement defines a custom resource for an application.

Syntax

```
RESOURCE type-id resource-id [load-option] [mem-option] filename
```

Description: A custom resource can be any data in any format. The RESOURCE statement copies the custom resource from the specified file and adds it to the application's other resources. A custom resource can be loaded from the executable file when needed by using the DosGetResource or DosGetResource2 function and specifying the resource's type and resource identifier.

You can provide any number of RESOURCE statements in a resource script file, but each statement must specify a unique combination of type-id and resource-id values. That is, RESOURCE statements having the same type-id value are permitted as long as the resource-id value for each is unique.

Example: This example defines a custom resource whose type identifier is 300 and whose resource identifier is 14. The custom resource is copied from the file CUSTOM.RES.

```
RESOURCE 300 14 custom.res
```

RTEXT Statement

The RTEXT statement creates a right-aligned text control.

Syntax

```
RTEXT text, id, x, y, width, height [, style]
```

Description: The control is a simple rectangle displaying the given text right-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The RTEXT statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of the control. The predefined class for the control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_RIGHT, and WS_GROUP.

Example: This example creates a right-aligned text control that is labeled “Filename.”

```
RTEXT "Filename", 101, 10, 10, 100, 100
```

SLIDER Statement

The SLIDER statement creates a slider control within the dialog window.

Syntax

```
SLIDER   id, x, y, width, height [, style]
```

Description: This control lets the user set, display, or modify a value by moving a slider arm along a slider shaft. The SLIDER statement defines the identifier, position, dimensions, and attributes of a slider control. The predefined class for this control is WC_SLIDER. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

Use the SLIDER statement only in a DIALOG or WINDOW statement.

Example: This example creates a slider control at position (40, 30) within the dialog window. The slider has a width of 120 character units and a height of 2 character units. Its resource ID is 101. The style specification SLS_BUTTONSLEFT adds buttons to the left of the slider shaft. The default styles WS_TABSTOP and WS_VISIBLE are both in effect, although only the latter is specified.

```
#define    IDC_SLIDER       101
#define    IDD_SLIDERDLG    502
DIALOG "Slider", IDD_SLIDERDLG, 11, 11, 200, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    SLIDER   IDC_SLIDER, 40, 30, 120, 16, SLS_BUTTONSLEFT | WS_VISIBLE
  END
```

SPINBUTTON Statement

The SPINBUTTON statement creates a spin-button control within the dialog window.

Syntax

```
SPINBUTTON   id, x, y, width, height [, style]
```

Description: This control gives the user quick access to a finite set of data. The SPINBUTTON statement defines the identifier, position, dimensions, and attributes of a spin-button control. The predefined class for this control is WC_SPINBUTTON. If you do not specify a style, the default style is WS_TABSTOP, WS_VISIBLE, and SPBS_MASTER.

Use the SPINBUTTON statement only in a DIALOG or WINDOW statement.

Example: This example creates a spin-button control at position (80, 20) within the dialog window. The spin button has a width of 60 character units and a height of 3 character units. Its resource ID is 302. The style specification SPBS_NUMERICONLY creates a control which accepts only the digits 0-9 and virtual keys. The default styles SPBS_MASTER, WS_TABSTOP, and WS_VISIBLE are all in effect, although only WS_TABSTOP is specified.

```
#define    IDC_SPINBUTTON   302
#define    IDD_SPINDLG    502
DIALOG "Spin button", IDD_SPINDLG, 11, 11, 200, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    SPINBUTTON  IDC_SPINBUTTON, 80, 20, 60, 24, SPBS_NUMERICONLY | WS_TABSTOP
  END
```

STRINGTABLE Statement

The STRINGTABLE statement creates one or more string resources for an application.

Syntax

```
STRINGTABLE [load-option] [mem-option]
BEGIN
string-id string-definition
    .
    .
    .
END
```

Description: A string resource is a null-terminated character string that has a unique string identifier. A string resource can be loaded from the executable file when needed by using the WinLoadString function.

You can provide any number of STRINGTABLE statements in a resource script file. The compiler treats all the strings from the various STRINGTABLE statements as if they belonged to a single statement. This means that no two strings in a resource script file can have the same string identifier.

You can continue a string on multiple lines by terminating the line with a backslash (\) or by terminating the line with a double quotation mark (“) and then starting the next line with a double quotation mark.

Example: This example creates two string resources whose string identifiers are 1 and 2.

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
    IDS_HELLO   "Hello"
    IDS_GOODBYE "Goodbye"
END
```

SUBITEMSIZE Statement

The SUBITEMSIZE statement specifies the size, in words, of each help subitem in a help subtable.

Syntax

```
SUBITEMSIZE  size
```

Description: The minimum size is two words, and each help subitem in a help subtable must be the same size. When used, the SUBITEMSIZE statement must appear after the HELPSUBTABLE statement and before the BEGIN keyword.

You do not need to use the SUBITEMSIZE statement if the help subitems are the default size (2).

Example: The SUBITEMSIZE statement in this example specifies that each HELPSUBITEM statement contains three words.

```
HELPSUBTABLE 1
SUBITEMSIZE 3
BEGIN
    HELPSUBITEM IDCLD_FILEMENU, IDHP_FILEMENU, 5
    HELPSUBITEM IDCLD_HELPMENU, IDHP_HELPMENU, 6
END
```

SUBMENU Statement

The SUBMENU statement creates a submenu for a given menu.

Syntax

```
SUBMENU text, submenu-id [, menuitem-style[, menuitem-attributerbrk.]
BEGIN
menuitem-definition
    .
    .
    .
END
```

Description: A submenu is a vertical list of menu items from which the user can choose a command.

You can provide any number of SUBMENU statements in a MENU statement, but each SUBMENU statement must specify a unique submenu-id value. You can provide any number of menuitem-definition statements in the SUBMENU statement. These define the menu items (commands) in the menu. The order of the statements determines the order of the menu items.

Example: This example creates a submenu named Elements. Its identifier is 2. The submenu contains three menu items, which are created by using MENUITEM statements.

```
SUBMENU "Elements", 2
BEGIN
    MENUITEM "Oxygen", 200
    MENUITEM "Carbon", 201, , MIA_CHECKED
    MENUITEM "Hydrogen", 202
END
```

VALUESET Statement

The VALUESET statement creates a value set control within the dialog window.

Syntax

```
VALUESET   id, x, y, width, height [, style]
```

Description: This control lets a user select one choice from a group of mutually exclusive choices. The VALUESET statement defines the identifier, position, dimensions, and attributes of a value set control. The predefined class for this control is WC_VALUESET. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

Use the VALUESET statement only in a DIALOG or WINDOW statement.

Example: This example creates a value set control at position (40, 40) within the dialog window. The value set control has a width of 220 character units and a height of 20 character units. Its resource ID is 302. The style specification VS_ICON creates a control to show items in icon form. The default styles WS_TABSTOP and WS_VISIBLE are both in effect, although only WS_TABSTOP is specified.

```
#define    IDC_VALUESET     302
#define    IDD_VALUESETDLG  501
DIALOG "Value set", IDD_VALUESETDLG, 11, 11, 260, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    VALUESET  IDC_VALUESET, 40, 40, 220, 160, VS_ICON | WS_TABSTOP
  END
```

WINDOW Statement

The WINDOW statement creates a window of the specified class.

Syntax

```
WINDOW text, id, x, y, width, height, class [, style [, framect]]
  data-definitions
[ BEGIN
control-definition
    .
    .
    .
END ]
```

Description: The statement defines the position and dimensions of the window relative to its parent window, as well as the window-box style. The WINDOW statement is typically used in a WINDOWTEMPLATE or FRAME statement.

Usually, only one WINDOW statement is used in a FRAME statement. It defines the client window belonging to the corresponding frame window. The optional BEGIN and END keywords enclose any CONTROL statements that are given with the window. CONTROL statements given in this manner represent child windows belonging to the window created by the WINDOW statement.

The WINDOW statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a WINDOW statement contains one or no such statements.

Example: This example creates a client window belonging to the frame window. The client window belongs to the "MyClientClass" window class and has the standard window identifier FID_CLIENT.

```
WINDOWTEMPLATE 1
BEGIN
    FRAME "My Window", 1, 10, 10, 320, 130,
             0, FCF_STANDARD | FCF_VERTSCROLL
    BEGIN
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "MyClientClass"
    END
END
```

WINDOWTEMPLATE Statement

The WINDOWTEMPLATE statement creates a window template.

Syntax

```
WINDOWTEMPLATE window-id [load-option] [mem-option]
BEGIN
window-definition
    .
    .
    .
END
```

Description: A window template consists of a series of statements that define the window identifier, load and memory options, window dimensions, and controls in the window. The window template can be loaded from the executable file by using the WinLoadDlg function.

You can provide any number of window templates in a resource script file, but each template must have a unique window-id value.

A WINDOWTEMPLATE statement can contain DIALOG, CONTROL, and WINDOW statements. Typically, only one WINDOW statement is used in the WINDOWTEMPLATE statement.

Directive Descriptions

This section provides the syntax, a description, and an example of each of the directives.

#define Directive

The #define directive assigns the given value to the specified name. All subsequent occurrences of the name are replaced by the value.

Syntax

```
#define name value
```

Example

This example assigns values to the names "NONZERO" and "USERCLASS".

```
#define    NONZERO     1
#define    USERCLASS   "MyControlClass"
```

#elif Directive

The #elif directive marks an optional clause of a conditional-compilation block defined by a #ifdef, #ifndef, or #if directive. The directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, #elif directs the compiler to continue processing statements up to the next #endif, #else, or #elif directive and then skip to the statement after #endif. If the constant expression is zero, #elif directs the compiler to skip to the next #endif, #else, or #elif directive. You can use any number of #elif directives in a conditional block.

Syntax

```
#elif constant-expression
```

Example

In this example, #elif directs the compiler to process the second BITMAP statement only if the value assigned to the name "Version" is less than 7. The #elif directive itself is processed only if Version is greater than or equal to 3.

```
#if Version < 3
BITMAP 1 errbox.bmp
#elif Version < 7
BITMAP 1 userbox.bmp
#endif
```

#else Directive

The #else directive marks an optional clause of a conditional-compilation block defined by a #ifdef, #ifndef, or #if directive. The #else directive must be the last directive before the #endif directive.

This directive has no arguments.

Syntax

```
#else
```

Example

This example compiles the second BITMAP statement only if the name "DEBUG" is not defined.

```
#ifdef DEBUG
    BITMAP 1 errbox.bmp
#else
    BITMAP 1 userbox.bmp
#endif
```

#endif directive

The #endif directive marks the end of a conditional-compilation block defined by a #ifdef directive. One #endif is required for each #if, #ifdef, or #ifndef directive.

This directive has no arguments.

Syntax

```
#endif
```

#if Directive

The #if directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, #if directs the compiler to continue processing statements up to the next #endif, #else, or #elif directive and then skip to the statement after the #endif directive. If the constant expression is zero, it directs the compiler to skip to the next #endif, #else, or #elif directive.

Syntax

```
#if constant-expression
```

Example

This example compiles the BITMAP statement only if the value assigned to the name "Version" is less than 3.

```
#if Version < 3
BITMAP 1 errbox.bmp
#endif
```

#ifdef Directive

The #ifdef directive controls conditional compilation of the resource file by checking the specified name. If the name has been defined by using a define directive or by using the -d command-line option of rc, #ifdef directs the compiler to continue with the statement immediately after the #ifdef directive. If the name has not been defined, #ifdef directs the compiler to skip all statements up to the next #endif directive.

Syntax

```
#ifdef name
```

Example

This example compiles the BITMAP statement only if the name "Debug" is defined.

```
#ifdef Debug
BITMAP 1 errbox.bmp
#endif
```

#ifndef Directive

The #ifndef directive controls conditional compilation of the resource file by checking the specified name. If the name has not been defined or if its definition has been removed by using the #undef directive, #ifndef directs the compiler to continue processing statements up to the next #endif, #else, or #elif directive and then skip to the statement after the #endif directive. If the name is defined, #ifndef directs the compiler to skip to the next #endif, #else, or #elif directive.

Syntax

```
#ifndef name
```

Example

This example compiles the BITMAP statement only if the name "Optimize" is not defined.

```
#ifndef Optimize
BITMAP 1 errbox.bmp
#endif
```

#include Directive

The #include directive causes RC to process the file specified in the *filename* field. This file should be a header file that defines the constants used in the resource script file. Only the define directives in the specified file are processed. All other statements are ignored.

The *filename* field is handled as a C string. Therefore, you must include two backslashes (\\) wherever one is required in the path. (As an alternative, you can use a single forward slash (/) instead of two backslashes.)

Syntax

```
#include filename
```

Example

This example processes the header files OS2.H and HEADERS\MYDEFS.H\I while compiling the resource script file.

```
#include <os2.h>
#include "headers\\\\mydefs.h"
```

#undef Directive

The undef directive removes the current definition of the specified name. All subsequent occurrences of the name are processed without replacement.

Syntax

```
#undef name
```

Example

This example removes the definitions for the names “nonzero” and “USERCLASS”.

```
#undef     nonzero
#undef     USERCLASS
```

Using Resource Files

This section explains how to create a resource script file, compile it using the Resource Compiler (RC.EXE), and optionally add the resources to your executable file. Resource script files have a default file-name extension of .RC.

For resource information on the individual controls, see the chapter on the specific control. For example, an example of a resource script file for frame windows is in Chapter 6, "Frame Windows" on page 6-1.

Creating and Compiling a Resource File

The resource compiler (RC) compiles a resource script file to create a new file, called a binary resource file, which has a .RES file-name extension. The binary resource file can be added to the executable file of the application, thereby replacing any existing resources in that file.

The RC command line has the following three basic forms:

```
rc resource-script-file [executable-file]

rc binary-resource-file [executable-file]

rc -r  resource-script-file [binary-resource-file]
```

Note: The third option does not add to the executable file.

The *resource-script-file* parameter is the file name of the resource script file to be compiled.

The *executable-file* parameter must be the name of the executable file to receive the compiled resources. This is a file having a file-name extension of either .EXE or .DLL. If you omit the executable-file field, RC adds the compiled resources to the executable file that has the same name as the resource script file but which has the .EXE file-name extension.

The *binary-resource-file* parameter is the name of the binary resource file to be added to the executable file.

The *-r* option directs RC to compile the resource script file without adding it to an executable file.

Compiling and Adding Resources to the .EXE File

To compile the resource script file EXAMPLE.RC and add the result to the executable file EXAMPLE.EXE, use the following command:

```
rc example
```

You do not need to specify the .RC extension. RC creates the binary resource file EXAMPLE.RES and adds the compiled resource to the executable file EXAMPLE.EXE.

Compiling without Adding Resources to the .EXE File

To compile the resource script file EXAMPLE.RC into a binary resource file without adding the resources to an executable file, use the following command:

```
rc -r example
```

The compiler creates the binary resource file EXAMPLE.RES. To create a binary resource file that has a name different from the resource script file, use the following command:

```
rc -r example newfile.res
```

Adding the Compiled Resources to the .EXE File

To add the compiled resources in the binary resource file EXAMPLE.RES to an executable file, use the following command:

```
rc example.res
```

To specify the name of the executable file, if the name is different from the resource file, use the following command:

```
rc example.res newfile.exe
```

Adding the Compiled Resources to a DLL

To add the compiled resources to a dynamic-link-library (DLL) file, use the following command:

```
rc example.res dynalink.dll
```

Summary

The following tables summarize the resource statements and directives associated with resource files.

Table 12-1 (Page 1 of 4). Resource File Statements

Statement Name	Description
ACCELTABLE	Creates a table of accelerators for an application. An accelerator is a keystroke that gives the user a quick way to choose a command from a menu or carry out some other task.
ASSOCTABLE	Defines a file-association table for an application. This table associates the data files that an application creates with the executable file of the application.
AUTOCHECKBOX	Creates an automatic-check-box control, which is a small rectangle (check box) that contains an X when the user selects it.
AUTORADIOBUTTON	Creates an automatic-radio-button control, which is a small circle with the given text displayed to its right.
BITMAP	Defines a bit-map resource for an application. A bit-map resource, typically created using the Icon Editor, is a custom bit map that an application uses in its display or as an item in a menu.
CHECKBOX	Creates a check-box control, which is a small rectangle (check box) that has the specified text displayed to the right.
CODEPAGE	Sets the code page for all subsequent resources. The code page specifies the character set used for characters in the resource.
COMBOBOX	Creates a combination-box control, which combines a list-box control with an entry-field control. It allows the user to place the selected item from a list box into an entry field.
CONTAINER	Creates a container control within a dialog window. The container control is a visual component that holds objects.
CONTROL	Defines a control as belonging to the specified class. The statement defines the position and dimensions of the control within the parent window, as well as the control style.
CTEXT	Creates a centered-text control, which is a simple rectangle displaying the given text centered in the rectangle.
CTLDATA	Defines control data for a custom dialog box, window, or control. The CTLDATA statement is reserved for window classes that you create yourself.
DEFAULTICON	Installs an icon definition under the ICON EA of the program file.
DEFPUSHBUTTON	Creates a default push-button control, which is a round-cornered rectangle containing the given text. The rectangle has a bold outline to represent that it is the default response for the user.

Table 12-1 (Page 2 of 4). Resource File Statements

Statement Name	Description
DIALOG	Defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen, as well as the dialog-box style.
DLGINCLUDE	Adds the specified file name to the resource file. The DLGINCLUDE statement is typically used to let the application access the definitions file for the dialog box with the corresponding identifier.
DLGTEMPLATE	Creates a dialog-box template, which consists of a series of statements that define the identifier, load and memory options, dialog-box dimensions, and controls in the dialog box.
EDITTEXT	Creates an entry-field control, which is a rectangle in which the user can type and edit text. The control displays a pointer when the user selects the control.
ENTRYFIELD	Creates an entry-field control, which is a rectangle in which the user can type and edit text.
FONT	Defines a font resource for an application. A font resource, typically created by using the OS/2 Font Editor, is a bit map defining the shape of the individual characters in a character set.
FRAME	Defines a frame window, specifying title, identifier, position, and dimensions of the frame window, as well as the window style.
GROUPBOX	Creates a group-box control, which is a rectangle that groups other controls together by drawing a border around them and displaying the given text in the upper-left corner.
HELPITEM	Defines the help items in a help table, specifying the resource identifier of an application window for which help is provided, along with the resource identifiers of the help subtable and extended help panel associated with the application window.
HELPSUBITEM	Defines the help subitems in a help subtable, specifying the identifier of a child window for which help is provided, the identifier of the help panel associated with the child window, and one or more optional, application-defined integers.
HELPSUBTABLE	Defines the contents of a help-subtable resource, which contains a help-subitem entry for each item that can be selected in an application window.
HELPTABLE	Defines the contents of a help-table resource, which contains a help-item entry for each application window, dialog box, and message box for which help is provided.
ICON (resource)	This form of the ICON statement defines an icon resource for an application. An icon resource, typically created by using the Icon Editor, is a bit map defining the shape of the icon to be used for a given application.

Table 12-1 (Page 3 of 4). Resource File Statements

Statement Name	Description
ICON (control)	This form of the ICON statement creates an icon control, which is an icon displayed in a dialog box. The ICON statement defines the icon-resource identifier, icon-control identifier, position, and attributes of a control window.
LISTBOX	Creates commonly-used controls for a dialog box or window. The control is a rectangle containing a list of user-selectable strings such as file names.
LTEXT	Creates a left-aligned text control, which is a simple rectangle displaying the given text left-aligned in the rectangle.
MENU	Defines the contents of a menu resource, which is a collection of information that defines the appearance and function of an application menu.
MENUITEM	Creates a menu item for a menu, specifying the text, identifier, and attributes of a menu item.
MESSAGETABLE	Creates one or more string resources for an application. A string resource is a null-terminated character string that has a unique string identifier.
MLE	Creates a multiple-line entry-field control, which is a rectangle in which the user can type and edit multiple lines of text.
NOTEBOOK	Creates a notebook control within the dialog window. This control is used to organize information on individual pages so it can be located and displayed easily.
POINTER	Defines a pointer resource for an application. A pointer resource, typically created by using the OS/2 Icon Editor, is a bit map defining the shape of the mouse pointer on the screen.
PRESPARAMS	Defines presentation fields that customize a dialog box, menu, window, or control.
PUSHBUTTON	Creates a push-button control, which is a round-cornered rectangle containing the given text.
RADIOBUTTON	Creates a radio-button control, which is a small circle that has the given text displayed to its right.
RCDATA	Defines a custom-data resource for an application. The custom data can be in whatever format the application requires.
RCINCLUDE	Causes RC to process the resource script file specified in the *filename* field along with the current resource script file.
RESOURCE	Defines a custom resource for an application. A custom resource can be any data in any format.
RTEXT	Creates a right-aligned text control, which is a simple rectangle displaying the given text right-aligned in the rectangle.
SLIDER	Creates a slider control within the dialog window. This control lets the user set, display, or modify a value by moving a slider arm.

Table 12-1 (Page 4 of 4). Resource File Statements

Statement Name	Description
SPINBUTTON	Creates a spin-button control within the dialog window. This control gives the user quick access to a finite set of data.+
STRINGTABLE	Creates one or more string resources for an application. A string resource is a null-terminated character string that has a unique string identifier.
SUBITEMSIZE	Specifies the size, in words, of each help subitem in a help subtable.
SUBMENU	Creates a submenu for a given menu. A submenu is a vertical list of menu items from which the user can choose a command.
VALUESET	Creates a value set control within the dialog window. This control lets a user select one choice from a group of mutually exclusive choices.
WINDOW	Creates a window of the specified class, defining the position and dimensions of the window relative to its parent window, as well as the window-box style.
WINDOWTEMPLATE	Creates a window template, which consists of a series of statements that define the window identifier, load and memory options, window dimensions, and controls in the window.

Table 12-2. Directives

Directive Name	Description
#define	Assigns the given value to the specified name. All subsequent occurrences of the name are replaced by the value.
#elif	Marks an optional clause of a conditional-compilation block defined by a #ifdef, #ifndef, or #if directive.
#else	Marks an optional clause of a conditional-compilation block defined by a #ifdef, #ifndef, or #if directive.
#endif	Marks the end of a conditional-compilation block defined by a #ifdef directive.
#if	Controls conditional compilation of the resource file by checking the specified constant expression.
#ifdef	Controls conditional compilation of the resource file by checking the specified name.
#ifndef	Controls conditional compilation of the resource file by checking the specified name.
#include	Causes RC to process the file specified in the *filename* field.
#undef	Removes the current definition of the specified name.

Chapter 13. Menus

A *menu* is a window that contains a list of items— text strings, bit maps, or images drawn by the application—that enables the user, by mouse or keyboard, to choose from these predetermined choices. This chapter describes how to use menus in your PM applications.

About Menus

A menu always is owned by another window, usually a frame window. When a user makes a choice from a menu, the menu posts a message containing the unique identifier for the menu item to its owner by way of the owner window's window procedure.

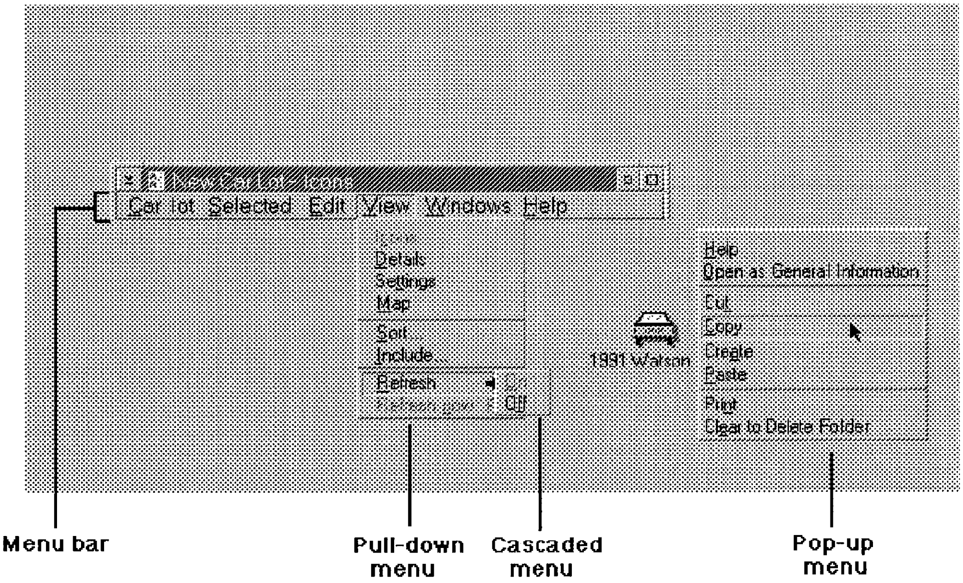


Figure 13-1. Menus

An application typically defines its menus using Resource Compiler, and then associates the menus with a frame window when the frame window is created. Applications also can create menus by filling in menu-template data structures and creating windows with the WC_MENU class. Either way, applications can add, delete, or change menu items dynamically by issuing messages to menu windows.

Menu Bar and Pull-Down Menus

A typical application uses a menu bar and several pull-down submenus. The pull-down submenus ordinarily are hidden, but become visible when the user makes selections in the menu bar. Pull-down submenus always are attached to the menu bar.

The menu bar is a child of the frame window; the menu bar window handle is the key to communicating with the menu bar and its submenus. You can retrieve this handle by calling WinWindowFromID, with the handle of the parent window and the FID_MENU frame-control identifier. Most messages for the menu bar and its submenus can be issued to the menu-bar window. Flags in the messages tell the window whether to search submenus for requested menu items.

Pop-Up Menus

A pop-up menu is like a pull-down submenu, except that it is not attached to the menu bar; it can appear anywhere in its parent window. A pop-up menu usually is associated with a *portion* of a window, such as the client window (see Figure 13-2), or it is associated with a specific object, such as an icon.

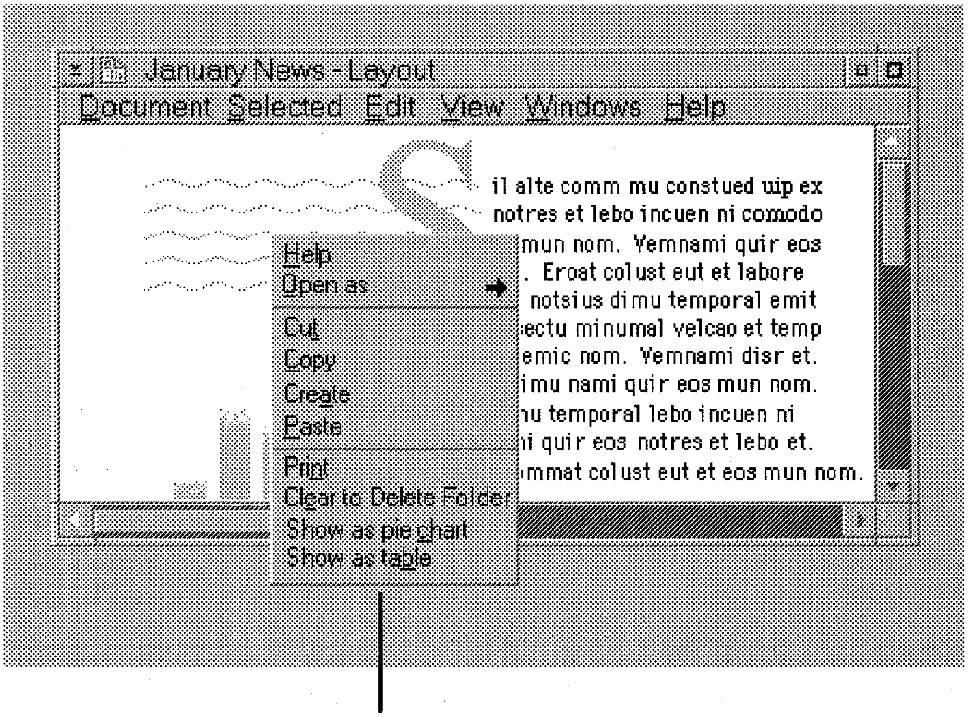

January News - Layout
Document Selected Edit View Windows Help
Help
Open as
Cut
Copy
Create
Paste
Print
Clear to Delete Folder
Show as pie chart
Show as table
il alte comm mu constued uip ex
notres et lebo incuen ni comodo
mun nom. Yemnami quir eos
. Eroat colust eut et labore
notsius dimu temporal emit
ectu minumal velcao et temp
emic nom. Yemnami disr et.
imu nami quir eos mun nom.
u temporal lebo incuen ni
i quir eos notres et lebo et.
mmat colust eut et eos mun nom.
Pop-up menu

Figure 13-2. Pop-Up Menu

A pop-up menu remains hidden until the user selects it (either by moving the cursor to the appropriate location and pressing Enter or clicking on the location with the mouse). Typically, pop-up menus are displayed at the position of the cursor or mouse pointer; they provide a quick mechanism for selecting often-used menu items.

To include a pop-up menu in an application, you first must define a menu resource in a resource-definition file, then load the resource using the WinLoadMenu or WinCreateMenu functions. You must call WinPopupMenu to create the pop-up menu and display it in the parent window. Applications typically call WinPopupMenu in a window procedure in response to a user-generated message, such as WM_BUTTON2DBLCLK or WM_CHAR.

WinPopupMenu requires that you specify the pop-up menu's handle and also the handles of the parent and owner windows of the pop-up menu. WinLoadMenu and WinCreateMenu return the handle of the pop-up menu window, but you must obtain the handles of the parent and owner by using WinQueryWindow.

You determine the position of the pop-up menu in relation to its parent by specifying coordinates and style flags in WinPopupMenu. The *x* and *y* coordinates determine the position of the lower-left corner of the menu relative to the lower-left corner of the parent. The system may adjust this position, however, if you include the PU_HCONSTRAIN or PU_VCONSTRAIN style flags in the call to WinPopupMenu. If necessary, PU_HCONSTRAIN adjusts the horizontal position of the menu so that its left and right edges are within the borders of the desktop window. PU_VCONSTRAIN makes the same adjustments vertically. Without these flags, a desktop-level pop-up menu can lie partially off the screen, with some items not visible nor selectable.

The PU_POSITIONONITEM flag also can affect the position of the pop-up menu. This flag positions the pop-up menu so that, when the pop-up menu appears, the specified item lies directly under the mouse pointer. Also, PU_POSITIONONITEM automatically selects the item. PU_POSITIONONITEM is useful for placing the current menu selection under the pointer so that, if the user releases the mouse button without selecting a new item, the current selection remains unchanged.

The PU_SELECTITEM flag is similar to PU_POSITIONONITEM except that it just selects the specified item; it does not affect the position of the menu.

You can enable the user to choose an item from a pop-up menu by using the same mouse button that was used to display the menu. To do this, specify the PU_MOUSEBUTTON*n* flag, where *n* corresponds to the mouse button used to display the menu. This flag specifies the mouse buttons for the user to interact with a pop-up menu once it is displayed.

By using the PU_MOUSEBUTTON*n* flag, you can enable the user to display the pop-up menu, select an item, and dismiss the menu, all in one operation. For example, if your window procedure displays the pop-up window when the user double-clicks mouse button 2, specify the PU_MOUSEBUTTON2DOWN flag in the WinPopupMenu function. Then, the user can display the menu with mouse button 2; and, while holding the button down, select an item. When the user releases the button, the item is chosen and the menu dismissed.

System Menu

The system menu in the upper-left corner of a standard frame window is different from the menus defined by the application. The system menu is controlled and defined almost exclusively by the system; your only decision about it is whether to include it when creating a frame window. (It is unusual for a frame window *not* to include a system menu.) The system menu generates WM_SYSCOMMAND messages instead of WM_COMMAND messages. Most applications simply use the default behavior for WM_SYSCOMMAND messages, although applications can add, delete, and change system-menu entries.

Menu Items

All menus can contain two main types of menu items: command items and submenu items. When the user chooses a command item, the menu immediately posts a message to the parent window. When the user selects a submenu item, the menu displays a submenu from which the user may choose another item. Since a submenu window also can contain a submenu item, submenus can originate from other submenus.

When the user chooses a command item from a menu, the menu system posts a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message to the owner window, depending on the style bits of the menu item.

Applications can change the attributes, style, and contents of menu items, and insert and delete items at run time, to reflect changes in the command environment. An application also can add items to or delete items from the menu bar, a pop-up menu, or a submenu. For example, an application might maintain a menu of the fonts currently available in the system. This application would use graphics programming interface (GPI) calls to determine which fonts were available and, then, insert a menu item for each font into a submenu. Furthermore, the application might set the check-mark attribute of the menu item for the currently chosen font. When the user chose a new font, the application would remove the check-mark attribute from the previous choice and add it to the new choice.

The Help Item

To present a standard interface to the novice user, all applications must have a Help item in their menu bars. The Help item is defined with a particular style, attributes, and position in the menu. When the user chooses the Help item, the menu posts a WM_HELP message to the owner window, enabling the application to respond appropriately.

The item should read `Help`, have an identifier of 0, and have the MIS_BUTTONSEPARATOR or MIS_HELP item styles. The Help menu item should be the last item in the menu template, so that it is displayed as the rightmost item in the menu bar.

If an application uses the system default accelerator table, the user can select the Help item using either a mouse or the F1 key.

Menu-Item Styles

All menu items have a combination of style bits that determine what kind of data the item contains and what kind of message it generates when the user selects it. For example, a menu item can have the MIS_TEXT, MIS_BITMAP, or other styles that specify the visual representation of the menu item on the screen. Other styles determine what kinds of messages the item sends to its owner and whether the owner draws the item. Menu-item styles typically do not change during program execution, but you can query and set them dynamically by sending MM_QUERYITEM and MM_SETITEM messages with the menu-item identifier to the menu-bar window. For text menu items (MIS_TEXT), an MM_SETITEMTEXT message sets the text. The MM_QUERYITEMTEXT message queries the text of the item.

For non-text menu items, the *hItem* field of the MENUITEM structure typically contains the handle of a display object, such as a bit-map handle for MIS_BITMAP menu items.

An application can draw a menu item by setting the style MIS_OWNERDRAW for the menu item. This usually is done by specifying the MIS_OWNERDRAW style for the menu item in the resource-definition file; but it also can be done at run time. When the application draws a menu item, it must respond to messages from the menu each time the item must be drawn.

Menu-Item Attributes

Menu items have attributes that determine how the items are displayed and whether or not the user can choose them. An application can set and query menu-item attributes by sending MM_SETITEMATTR and MM_QUERYITEMATTR messages, with the menu-item identifier, to the menu-bar window. If the specified item is in a submenu, there are two methods of determining its attributes. The first is to send MM_SETITEMATTR and MM_QUERYITEMATTR messages to the top-level menu, specifying the identifier of the item and setting a flag so that the message searches all submenus for the item. Then, you can retrieve the handle of the menu-bar by calling WinWindowFromID, with the handle of the frame window and the FID_MENU frame-control identifier.

The second method, which is more efficient if you want to either work with more than one submenu item or set the same item several times, involves two steps:

1. Send an MM_QUERYITEM message to the menu, with the identifier of the submenu. The updated MENUITEM structure contains the window handle of the submenu.
2. Send an MM_QUERYITEMATTR (or MM_SETITEMATTR) message to the submenu window, specifying the identifier of the item in the submenu.

Menu-Item Structure

A single menu item is defined by the MENUITEM data structure. This structure is used with the MM_INSERTITEM message to insert items in a menu or to query and set item characteristics with the MM_QUERYITEM and MM_SETITEM messages. The MENUITEM structure has the following form:

```
typedef struct _MENUITEM { /* mi */
    SHORT  iPosition;
    USHORT afStyle;
    USHORT afAttribute;
    USHORT id;
    HWND   hwndSubMenu;
    ULONG  hItem;
} MENUITEM;
```

You can derive the values of most of the fields in this structure directly from the resource-definition file. However, the last field in the structure, *hItem*, depends on the style of the menu item.

The *iPosition* field specifies the ordinal position of the item within its menu window. If the item is part of the menu bar, *iPosition* specifies its relative left-to-right position, with 0 being the leftmost item. If the item is part of a submenu, *iPosition* specifies its relative top-to-bottom and left-to-right positions, with 0 being the upper-left item. An item with the MIS_BREAKSEPARATOR style in a pull-down menu causes a new column to begin.

The *afStyle* field contains the style bits of the item. The *afAttribute* field contains the attribute bits.

The *id* field contains the menu-item identifier. The identifier *should* be unique but does not *have* to be. Just remember that, when multiple items have the same identifier, they post the same command number in the WM_COMMAND, WM_SYSCOMMAND, and WM_HELP messages. Also, any message that specifies a menu item with a non-unique identifier will find the first item that has that identifier.

The *hwndSubMenu* field contains the window handle of a submenu window (if the item is a submenu item). The *hwndSubMenu* field is NULL for command items.

The *hItem* field contains a handle to the display object for the item, unless the item has the MIS_TEXT style, in which case, *hItem* is 0. For example, a menu item with the MIS_BITMAP style has an *hItem* field that is equal to its bit-map handle.

Menu Access

The OS/2 operating system is designed to work with or without a mouse or other pointing device. The system provides default behavior that enables a user to interact with menus without a mouse. Following are the keystrokes that produce this default behavior:

Table 13-1 (Page 1 of 2). Keystroke Menu Access

Keystroke	Action
Alt	Toggles in and out of menu-bar mode.
Alt+Spacebar	Shows the system menu.
F10	Backs up one level. If a submenu is displayed, it is canceled. If no submenu is displayed, this keystroke exits the menu.
Shift+Esc	Shows the system menu.
Right Arrow	Cycles to the next top-level menu item. If the selected item is at the far-left side of the menu, the menu code sends a WM_NEXTMENU message to the frame window. The default processing by the frame window is to cycle between the application and system menus. (An application can modify this behavior by subclassing the frame window.) If the selected item is in a submenu, the next column in the submenu is selected, or the next top-level menu item is selected; this keystroke also can send or process a WM_NEXTMENU message.
Left Arrow	Works like the Right Arrow key, except in the opposite direction. In submenus, this keystroke backs up one column, except when the currently selected item is in the far-left column, in which case the previous submenu is selected.
Up Arrow or **Down Arrow**	When pressed in a top-level menu, activates a submenu. When pressed in a submenu, this keystroke selects the previous or next or item, respectively.

Table 13-1 (Page 2 of 2). Keystroke Menu Access

Keystroke	**Action**
Enter	Activates a submenu, and highlights the first item if an item has a submenu associated with it; otherwise, this keystroke chooses the item as though the user released the mouse button while the item was selected.
Alphabetic character	Selects the first menu item with the specified character as its mnemonic key. A mnemonic is defined for a menu item by placing a tilde (~) before the character in the menu text. If the selected item has a submenu associated with it, the menu is displayed, and the first item is highlighted; otherwise, the item is chosen.

An application does not support the default keyboard behavior with any unusual code; instead, the application receives a message when a menu item is chosen by the keyboard just as though it had been chosen by a mouse.

Mnemonics

Adding mnemonics to menu items is one way of providing the user with keyboard access to menus. You can indicate a mnemonic keystroke for a menu item by preceding a character in the item text with a tilde, as in *~nFile*. Then, the user can choose that item by pressing the mnemonic key when the menu is active. Figure 13-3 shows the result on screen.

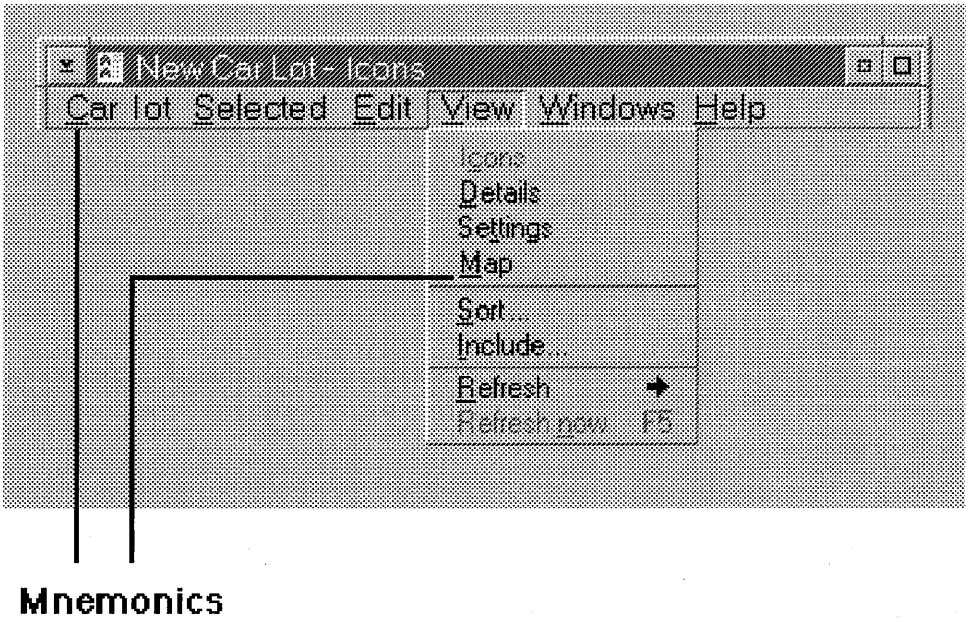


Figure 13-3. Examples of Mnemonics

The menu bar is *active* when the user presses and releases the Alt key, and the first item in the menu bar is highlighted. A pop-up or pull-down menu is active when it is *open*.

Accelerators

In addition to mnemonics, a menu item can have an associated *keyboard accelerator*. Accelerators are different from mnemonics in that the menu need not be active for the accelerator key to work. If you have associated a menu item with a keyboard accelerator, display the accelerator to the right of the menu item. Do this in the resource-definition file by placing a tab character (\t) in the menu text before the characters that will be displayed on the right. For example, if the Close item had the F3 function key as its keyboard accelerator, the text for the item would be `Close\tF3`.

Using Menus

This section explains how to perform the following tasks:

- Define menu items in a resource file.
- Include a menu bar in a standard window.
- Create a pop-up menu.
- Add a menu to a dialog window.
- Access the system menu.
- Respond to a the menu choice of a user.
- Set and query menu-item attributes.
- Add and delete menu items.
- Create a custom menu item.

Defining Menu Items in a Resource File

Typically, a menu resource represents the menu bar or pop-up menu and all the related submenus. A menu-item definition is organized as shown in the following code:

```
MENUITEM item text, item identifier, item style, item attributes
```

The menu resource-definition file specifies the text of each item in the menu, its unique identifier, its style and attributes, and whether it is a command item or a submenu item. A menu item that has no specification for style or attributes has the default style of MIS_TEXT and all attribute bits off, indicating that the item is enabled. The MIS_SEPARATOR style identifies nonselectable lines between menu items. Figure 13-4 on page 13-9 is sample Resource Compiler source code that defines a menu resource. The code defines a menu with three submenu items in the menu bar (*File, Edit*, and *Font*) and a command item (*Help*). Each submenu has several command items, and the *Font* submenu has two other submenus within it.

```
MENU ID_MENU_RESOURCE
BEGIN
    SUBMENU "~File", IDM_FILE
        BEGIN
            MENUITEM "~Open...",       IDM_FI_OPEN
            MENUITEM "~Close\tF3",     IDM_FI_CLOSE, 0, MIA_DISABLED
            MENUITEM "~Quit",          IDM_FI_QUIT
            MENUITEM "",               IDM_FI_SEP1, MIS_SEPARATOR
            MENUITEM "~About Sample",  IDM_FI_ABOUT
        END
    SUBMENU "~Edit", IDM_EDIT
        BEGIN
            MENUITEM "~Undo",          IDM_ED_UNDO, 0, MIA_DISABLED
            MENUITEM "",               IDM_ED_SEP1, MIS_SEPARATOR
            MENUITEM "~Cut",           IDM_ED_CUT
            MENUITEM "C~opy",          IDM_ED_COPY
            MENUITEM "~Paste",         IDM_ED_PASTE
            MENUITEM "C~lear",         IDM_ED_CLEAR
        END
    SUBMENU "Font", IDM_FONT
        BEGIN
            SUBMENU "Style",           IDM_FONT_STYLE
                BEGIN
                    MENUITEM "Plain",  IDM_FONT_STYLE_PLAIN
                    MENUITEM "Bold",   IDM_FONT_STYLE_BOLD
                    MENUITEM "Italic", IDM_FONT_STYLE_ITALIC
                END
            SUBMENU "Size",            IDM_FONT_SIZE
                BEGIN
                    MENUITEM "10",     IDM_FONT_SIZE_10
                    MENUITEM "12",     IDM_FONT_SIZE_12
                    MENUITEM "14",     IDM_FONT_SIZE_14
                END
        END
    MENUITEM "F1=Help", 0x00, MIS_TEXT | MIS_BUTTONSEPARATOR | MIS_HELP
END
```

Figure 13-4. Resource Compiler Code Defining a Menu Resource

To define a menu item with the MIS_BITMAP style, an application must use a tool such as Icon Editor to create a bit map, include the bit map in its resource-definition file, and define a menu in the file (as shown in Figure 13-5 on page 13-10). The text for the bit map menu items is an ASCII representation of the resource identifier of the bit map resource to be displayed for that item.

```
/* Bring externally created bit maps into the resource file. */
BITMAP 101 button.bmp
BITMAP 102 hirest.bmp
BITMAP 103 hizoom.bmp
BITMAP 104 hired.bmp

/* Connect a menu item with a bit map.                      */
SUBMENU "~Bitmaps", IDM_BITMAP
    BEGIN
        MENUITEM "#101", IDM_BM_01, MIS_BITMAP
        MENUITEM "#102", IDM_BM_02, MIS_BITMAP
        MENUITEM "#103", IDM_BM_03, MIS_BITMAP
        MENUITEM "#104", IDM_BM_04, MIS_BITMAP
    END
```

Figure 13-5. Defining a Menu with the MIS_BITMAP Style

Including a Menu Bar in a Standard Window

If you have defined a menu resource in a resource-definition file, you can use the menu resource to create a menu bar in a standard window. You include the menu bar by using the FCF_MENU attribute flag and specifying the menu-resource identifier in a call to WinCreateStdWindow, as shown in the following code fragment:

```
#define ID_MENU_RESOURCE 100

HWND hwndFrame;
CHAR szClassName[]="MyClass";
CHAR szTitle[]="My Title";

ULONG flControlStyle = FCF_MENU     | FCF_SIZEBORDER |
                       FCF_TITLEBAR | FCF_ACCELTABLE;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &flControlStyle,
    szClassName,
    szTitle,
    0, (HMODULE) NULL,
    ID_MENU_RESOURCE,
    NULL);
```

After you make this call, the operating system automatically includes the menu in the window, drawing the menu bar across the top of the window. When the user chooses an item from the menu, the menu posts the message to the frame window. The frame window passes any WM_COMMAND messages to the client window. (The frame window does not pass WM_SYSCOMMAND messages to the client window.) WM_HELP messages are posted to the focus window. The WinDefWindowProc function passes WM_HELP messages to the parent window. If a WM_HELP message is passed to a frame window, the frame window calls the HK_HELP hook. Your client window procedure must process these messages to respond to the user's actions.

Creating a Pop-up Menu

The following code fragment shows how to make a pop-up menu appear when the user double-clicks mouse button 2 anywhere in the parent window. The menu is positioned with the mouse pointer located on the item having the IDM_OPEN identifier and is constrained horizontally and vertically. Then, the user can select an item from the pop-up menu using mouse button 2.

```
#define ID_MENU_RESOURCE  110
#define IDM_OPEN          120

HWND hwndFrame;

MRESULT ClientWndProc(
HWND hwnd,
ULONG msg,
MPARAM mp1,
MPARAM mp2)
{
    HWND hwndMenu;
    BOOL fSuccess;

    switch (msg) {
        .
        .   /* Process other messages. */
        .
        case WM_BUTTON2DBLCLK:
            hwndMenu = WinLoadMenu(hwnd, (HMODULE) NULL, ID_MENU_RESOURCE);
            fSuccess = WinPopupMenu(hwnd,
                                    hwndFrame,
                                    hwndMenu,
                                    20,
                                    50,
                                    IDM_OPEN,
                                    PU_POSITIONONITEM   |
                                    PU_HCONSTRAIN       |
                                    PU_VCONSTRAIN       |
                                    PU_MOUSEBUTTON2DOWN |
                                    PU_MOUSEBUTTON2);

                   .
                   .
                   .
```

Adding a Menu to a Dialog Window

You might want to use menus in windows that were not created using the WinCreateStdWindow function. For these windows, you can load a menu resource by using the WinLoadMenu function and specifying the parent window for the menu. WinLoadMenu assigns the specified menu resource to the parent. To see the menu in the window, you must send a WM_UPDATEFRAME message to the parent after loading the menu resource. This strategy is especially useful for adding menus to a window created as a dialog window, but it can be used no matter what type of window is specified as the parent.

Accessing the System Menu

Although most applications do not alter the system menu, you can obtain the handle of the system menu by calling WinWindowFromID with a frame-window handle (or dialog-window handle) and the identifier FID_SYSMENU. Once you have the handle of the system menu, you can access the individual menu items by using predefined constants. For example, the following code fragment shows how to disable the *Close* menu item in the system menu of a window:

```
HWND hwndSysMenu;
HWND hwndFrame;

hwndSysMenu = WinWindowFromID(hwndFrame, FID_SYSMENU);

WinSendMsg(hwndSysMenu, MM_SETITEMATTR,
    MPFROM2SHORT(SC_CLOSE, TRUE),
    MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
```

Responding to a User's Menu Choice

When a user chooses a menu item, the client window procedure receives a WM_COMMAND message with SHORT1FROMMP(mp1) equal to the menu identifier of the chosen item. Your application must use the menu identifier to guide its response to the choice. Typically, the code in the client window procedure resembles the following code fragment:

```
case WM_COMMAND:
    DoMenuCommand(hwnd, SHORT1FROMMP(mp1));
    return 0;
```

The function that translates the menu identifier into an action typically resembles the following code fragment:

```
VOID DoMenuCommand(
HWND hwnd,
USHORT usItemID)
{

    /* Test the menu item. */
    switch (usItemID) {
        case IDM_FI_NEW:
            DoNew(hwnd);
            break;

      .
      . /* etc. */
      .

    }
}
```

The menu window sends a WM_MENUSELECT message every time the menu selection changes. SHORT1FROMMP(mp1) contains the identifier of the item that is changing state, and SHORT2FROMMP(mp2) is a 16-bit Boolean value that describes whether or not the item is chosen; the *mp2* parameter contains the handle of the menu.

If the Boolean value is FALSE, the item is selected but not chosen; for example, the user may have moved the cursor or mouse pointer over the item while the button was down. An application can use this message to display Help information at the bottom of the application window. The return value is ignored.

If the Boolean value is TRUE, the item is chosen—that is, the user pressed Enter or released the mouse button while an item was selected. If the application returns FALSE, the menu does not generate a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, and the menu is not dismissed.

Setting and Querying Menu-Item Attributes

Menu-item attributes are represented in the *fAttribute* field of the MENUITEM data structure. Typically, attributes are set in the resource-definition file of the menu and are changed at run time as required. Applications can use the MM_SETITEMATTR and MM_QUERYITEMATTR messages to set and query attributes for a particular menu item. One of the most common uses of these messages is to check and uncheck menu items to let the user know what option is selected currently. For example, if you have a menu item that should toggle between checked and unchecked each time the user selects it, you can use Figure 13-6 to change the checked attribute. In this example, you send an MM_QUERYITEMATTR message to the menu item to obtain its current checked attribute; then, you use the exclusive OR operator to toggle the state; and finally, you send the new attribute state back to the item using an MM_SETITEMATTR message.

```
usAttrib = SHORT1FROMMR(
    WinSendMsg(hwndMenu,            /* Submenu window       */
    MM_QUERYITEMATTR,               /* Message              */
    (MPARAM)itemID,                 /* Item identifier      */
    (MPARAM)MIA_CHECKED             /* Attribute mask       */
    ));

usAttrib = MIA_CHECKED;         /* XOR to toggle checked attribute */

WinSendMsg(hwndMenu,                           /* Submenu window       */
    MM_SETITEMATTR,                            /* Message              */
    (MPARAM)itemID,                            /* Item identifier      */
    MPFROM2SHORT(MIA_CHECKED, usAttrib)); /* Attribute mask, value */
```

Figure 13-6. Changing a Menu Item to Toggle Between Checked and Unchecked

Adding and Deleting Menu Items

An application can add and delete items from its menus dynamically by sending MM_INSERTITEM and MM_DELETEITEM messages to the menu window. Any item, including those in submenus, can be deleted by sending a message to the menu window.

Messages to insert items in submenus must be sent to the submenu's window (rather than to the window of the top-level menu). You can retrieve the handle of a submenu of the menu bar by sending an MM_QUERYITEM message to the menu-bar and specifying the identifier of the submenu item for the submenu, as shown in the following code fragment:

```
/* IDM_MYMENUID is the identifier of the submenu containing the item. */

MENUITEM mi;
HWND hwndMenu, hwndSubMenu, hwndPullDown,hwndFrame;

hwndMenu = WinWindowFromID(hwndFrame, FID_MENU);
WinSendMsg(hwndMenu,                           /* Handle of menu bar  */
    MM_QUERYITEM,                              /* Message             */
    MPFROM2SHORT(IDM_MYMENUID, TRUE),          /* Submenu identifier  */
    (MPARAM) &mi);                             /* Pointer to MENUITEM */

hwndPullDown = mi.hwndSubMenu;                 /* Handle to submenu   */
```

Once the application has the handle of the submenu, it can insert an item by filling in a MENUITEM structure and sending an MM_INSERTITEM message to the submenu. For text-menu items, the application must send a pointer to the text string as well as to the MENUITEM structure, as shown in Figure 13-7.

```
PSZ  pszNewItemString;

mi.iPosition = MIT_END;
mi.afStyle = MIS_TEXT;
mi.afAttribute = 0;
mi.id = IDM_MYMENU_FIRST;
mi.hwndSubMenu = NULL;
mi.hItem = 0;

WinSendMsg(hwndPullDown, MM_INSERTITEM, (MPARAM) &mi,
    (MPARAM) pszNewItemString);
```

Figure 13-7. Inserting a Menu Item

To delete an item, the application sends an MM_DELETEITEM message to the menu bar, specifying the identifier of the item to delete. For example, to clear all the items following IDM_MYMENU_FIRST in a submenu in which the items are numbered sequentially, use the following code:

```
USHORT usItemNum;

/* Clear all the items in MYMENU.                 */
hwndMenu = WinWindowFromID(hwndFrame, FID_MENU);
usItemNum = IDM_MYMENU_FIRST;
while (WinSendMsg(hwndMenu, MM_DELETEITEM,
    MPFROM2SHORT(usItemNum++, TRUE), NULL) != 0);
```

Adding a complete submenu to the menu bar is a more complicated procedure than that shown in the previous examples. There are two strategies. The recommended technique is to define all possible submenus in your resource-definition file; and then, as your application runs, selectively remove and insert the submenus as needed.

For example, assume that your application has a submenu that you want to be displayed only when a particular application tool is in use. You must first define the submenu as part of the main menu resource in your resource-definition file, so that the system reads in the resource menu template and creates the submenu window along with the rest of the menu. You then can remove the submenu from the menu bar, saving the title of the submenu and the MENUITEM structure that defines the submenu, as shown in Figure 13-8:

```
HWND hwndMenu, hwndClient;
MENUITEM mi;
CHAR szMenuTitle[MAX_STRINGSIZE];

/* Remove a submenu so that you can replace it later.          */

/* Obtain the handle of a menu.                                 */
hwndMenu = WinWindowFromID(WinQueryWindow(hwndClient, QW_PARENT),
                           FID_MENU);

/* Obtain information on the item to remove.                    */
WinSendMsg(hwndMenu, MM_QUERYITEM,
    MPFROM2SHORT(IDM_MENUID, TRUE),  /* TRUE to search submenus */
    (MPARAM)&mi);

/* Save the text for the submenu item.                          */
WinSendMsg(hwndMenu, MM_QUERYITEMTEXT,
    MPFROM2SHORT(IDM_FONT, MAX_STRINGSIZE),
    (MPARAM)szMenuTitle);

/* Remove the item, but retain mi and szMenuTitle.              */
WinSendMsg(hwndMenu, MM_REMOVEITEM,
    MPFROM2SHORT(IDM_FONT, TRUE), NULL);
```

Figure 13-8. Removing a Submenu from the Menu Bar

It is important to use the MM_REMOVEITEM message, rather than MM_DELETEITEM, to remove the item; deleting the item destroys the submenu window—removing it does not. The submenu should remain intact so that you can insert it later.

To reinsert the submenu, send an MM_INSERTITEM message to the menu bar, passing the MENUITEM structure and menu title that you saved when you removed the item. The following code fragment shows how to insert a submenu that was removed by using the previous code example.

```
/* Put the submenu back in and obtain the handle of the menu bar. */
hwndMenu = WinWindowFromID(
               WinQueryWindow(hwndClient, QW_PARENT), FID_MENU);

/* Use the information that you saved when you removed the menu.  */
WinSendMsg(hwndMenu, MM_INSERTITEM, (MPARAM)&mi,
           (MPARAM)szMenuTitle);
```

The other technique that you can use to insert a submenu in the menu bar is to build up, in memory, a data structure as a menu template and use that template and WinCreateWindow to create a submenu. The resultant submenu window handle then is placed in the *hwndSubMenu* field of a MENUITEM structure, and the menu item is sent to the menu bar with an MM_INSERTITEM message.

You also can create an empty submenu window by using WinCreateWindow. Pass NULL for the *pCtlData* and *pPresParams* parameters, instead of building the menu template in memory. Then insert a new menu item in the menu bar by using the MM_INSERTITEM message, setting the MIS_SUBMENU style, and putting the window handle of the created menu into the *hwndSubMenu* field. Then use the MM_INSERTITEM message to insert the items in the new pull-down menu.

Creating a Custom Menu Item

Applications can customize the appearance of an individual menu item by setting the MIS_OWNERDRAW style bit for the item. The operating system sends two different messages to an application that include owner-drawn menu items: WM_MEASUREITEM and WM_DRAWITEM. Both messages include a pointer to an OWNERITEM data structure.

WM_MEASUREITEM is sent only once for each owner-drawn item when the menu is initialized. The message is sent to the owner of the menu (typically, a frame window), which forwards the message to its client window. Typically, the client window procedure processes WM_MEASUREITEM by filling in the *yTop* and *Right* fields of the RECTL structure, specified by the *rclItem* field of this OWNERITEM structure; this specifies the size of the rectangle needed to enclose the item when it is drawn. The following code fragment responds to a WM_MEASUREITEM message.

```
case WM_MEASUREITEM:
    ((POWNERITEM) mp2)->rclItem.xRight = 26;
    ((POWNERITEM) mp2)->rclItem.yTop = 10;
    return 0;
```

If a menu item has the MIS_OWNERDRAW style, the owner window receives a WM_DRAWITEM message every time the menu item needs to be drawn. You process this message by using the *hps* and *rclItem* fields of the OWNERITEM structure to draw the item. There are two situations in which the owner window receives a WM_DRAWITEM message:

- When the item must be redrawn completely
- When the item must be highlighted or have its highlight removed.

You can choose to handle one or both of these situations. Typically, you handle the drawing of the item. You may not want to handle the second situation, however, since the system-default behavior (inverting the bits in the item rectangle) often is acceptable. The two situations in which a WM_DRAWITEM message is received are detected by comparing the values of the *fsState* and *fsStateOld* fields of the OWNERITEM structure that is sent as part of the message. If the two fields are the same, draw the item. Before drawing the item, however, check its attributes to see whether it has the attributes MIA_CHECKED, MIA_FRAMED, or MIA_DISABLED. Then draw the item according to the attributes.

For example, when the checked attribute of an owner-drawn menu item changes, the system sends a WM_DRAWITEM message to the item so that it can redraw itself and either draw or remove the check mark. If you want the system-default check mark, simply draw the item and leave the *fsAttribute* and *fsAttributeOld* fields unchanged; the system draws the check mark if necessary. If you draw the check mark yourself, clear the MIA_CHECKED bit in both *fsAttribute* and *fsAttributeOld* so that the system does not attempt to draw a check mark.

In the same example, if *fsAttribute* and *fsAttributeOld* are not equal, the highlight showing that an item is selected needs to change. The MIA_HILITED bit of the *fsAttribute* field is set if the item needs to be highlighted and is not set if the highlight needs to be removed. If you do not want to provide your own highlighting, you should ignore any WM_DRAWITEM message in which *fsAttribute* and *fsAttributeOld* are not equal. If you do not alter these two fields, the system performs its default highlighting operation. If you want to provide your own visual cue that an item is selected, respond to a WM_DRAWITEM message in which the *fsAttribute* and *fsAttributeOld* fields are not equal by providing the cue and clearing the MIA_HILITED bit of both fields before returning from the message.

Likewise, the MIA_CHECKED and MIA_FRAMED bits of *fsAttribute* and *fsAttributeOld* either can be used to perform the corresponding action or passed on, unchanged, so that the system performs the action. The following code fragment shows how to respond to a WM_DRAWITEM message when you want to draw the item and also be responsible for its highlighted state.

```
case WM_DRAWITEM:
    {
    POWNERITEM poi;
    RECTL      rcl;
    MPARAM     mp2;

    poi = (POWNERITEM) mp2;

    /*
     * If the new attribute equals the old attribute,
     * redraw the entire item.
     */

    if (poi->fsAttribute == poi->fsAttributeOld) {

        /*
         * Draw the item in poi->hps and poi->rclItem, and check the
         * attributes for check marks. If you produce your own check marks,
         * use this line of code:
         *
         *      poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_CHECKED;
         */

    }

    /* Else highlight the item or remove its highlight. */

    else if ((poi->fsAttribute & MIA_HILITED) !=
            (poi->fsAttributeOld & MIA_HILITED)) {

        /*
         * Set bits the same so that the menu window does not highlight
         * the item or remove its highlight.
         */

        poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_HILITED);
    }
    return TRUE; /* TRUE means the item is drawn. */
    } /* endcase */
```

Figure 13-9. Responding to WM_DRAWITEM Message

Related Functions

This section covers the functions that are related to Menus.

WinCheckMenuItem

This macro sets the check state of the specified menu item to the flag.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinCheckMenuItem  (HWND hwndMenu, USHORT usId, BOOL fCheck)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Item identifier.

fCheck (BOOL) – input
Check flag.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinCreateMenu

This function creates a menu window from the menu template.

Syntax

```
#define INCL_WINMENUS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinCreateMenu  (HWND  hwndParent, PVOID  lpmt)
```

Parameters

hwndParent (HWND) – input

Owner- and parent-window handle of the created menu window.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

lpmt (PVOID) – input

Menu template in binary format.

Returns

hwndMenu (HWND) – returns

Menu-window handle.

WinEnableMenuItem

This macro sets the state of the specified menu item to the enable flag.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinEnableMenuItem  (HWND hwndMenu, USHORT usId, BOOL fEnable)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Item identifier.

fEnable (BOOL) – input
Enable flag.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinIsMenuItemChecked

This macro returns the state (checked/not checked) of the identified menu item.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinIsMenuItemChecked  (HWND  hwndMenu, USHORT  usId)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Identity of the menu item.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinIsMenuItemEnabled

This macro returns the state (enable/disable) of the menu item specified.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinIsMenuItemEnabled  (HWND hwndMenu, USHORT usId)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Identity of the menu item.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinIsMenuItemValid

This macro returns TRUE if the specified item is a valid choice.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinIsMenuItemValid  (HWND hwndMenu, USHORT usId)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Identity of the menu item.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinLoadMenu

This function creates a menu window from the menu template *idMenu* from *hmod*, and returns in *hwndMenu* the window handle for the created window.

Syntax

```
#define INCL_WINMENUS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinLoadMenu  (HWND hwndFrame, HMODULE hmod, ULONG idMenu)
```

Parameters

hwndFrame (HWND) – input

Owner- and parent-window handle.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

hmod (HMODULE) – input

Resource identifier.

NULLHANDLE	The resource is in the .EXE file of the application.
Other	The module handle returned by the DosLoadModule or DosQueryModuleHandle call.

idMenu (ULONG) – input

Menu identifier within the resource file.

Returns

hwndMenu (HWND) – returns

Menu-window handle.

WinPopupMenu

This function causes a pop-up menu to be presented.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

**BOOL WinPopupMenu (HWND hwndParent, HWND hwndOwner,
HWND hwndMenu, LONG x, LONG y, LONG idItem,
ULONG fs)**

Parameters

hwndParent (HWND) – input
Parent-window handle.

hwndOwner (HWND) – input
Owner-window handle.

hwndMenu (HWND) – input
Pop-up menu-window handle.

x (LONG) – input
x-coordinate of the pop-up menu position.

y (LONG) – input
y-coordinate of the pop-up menu position.

idItem (LONG) – input
Item identity.

fs (ULONG) – input
Options.

Position
Pop-up menu position.

PU_POSITIONONITEM Position the pop-up menu so that the item identified by the *idItem* parameter of the top-level menu specified by the *hwndMenu* parameter lies directly under the *x* \ *y* coordinates.

The position of the pop-up menu can be affected, if either the PU_HCONSTRAIN or or PU_VCONSTRAIN values of the *fs* parameter is also set.

This value also causes the pop-up menu item identified by the *idItem* to be selected.

Restrain

Pop-up menu position constraints.

These options allow the application to ensure that the pop-up menu is visible on the desktop.

PU_HCONSTRAIN Constrain the pop-up menu so that its width is wholly visible on the desktop.

If necessary the position of the pop-up menu will be adjusted so that its left edge is coincident with the left edge of the desktop or that its right edge is coincident with the right edge of the desktop.

PU_VCONSTRAIN Constrain the pop-up menu so that its height is wholly visible on the desktop.

If necessary the position of the pop-up menu will be adjusted so that its top edge is coincident with the top edge of the desktop or that its bottom edge is coincident with the bottom edge of the desktop.

InitialState

Initial input state of the pop-up menu.

This allows the user interaction which caused the application to summon the pop-up menu to be carried through as the initial user interaction with the pop-up menu.

For example, this permits the application to support the user interface in which mouse button 1 can be depressed to cause the pop-up menu to be presented and held down while moving the mouse over the menu in order to select another menu item and then released to dismiss the menu.

Only one of the following values can be selected:

PU_MOUSEBUTTON1DOWN The pop-up menu is initialized with mouse button 1 depressed.

PU_MOUSEBUTTON2DOWN The pop-up menu is initialized with mouse button 2 depressed.

PU_MOUSEBUTTON3DOWN The pop-up menu is initialized with mouse button 3 depressed.

PU_NONE The pop-up menu is to be presented uninfluenced by the user interaction which caused it to be summoned.

This is the default value.

Select

Item selection.

PU_SELECTITEM The item identified by *idItem* is to be selected. This is only valid if PU_NONE is set in the *InitialState* parameter.

If the identified item is in a submenu of the pop-up menu, then all the previous submenus in the menu hierarchy are presented with the correct path to the identified item.

Usage

Input device usage.

The window procedure controlling the pop-up menu must be informed of which input devices are available for interaction with the pop-up menu.

These options are independent to those of the *InitialState* parameter. Therefore, if an application indicates in the *InitialState* parameter that the pop-up menu is to be initialized with a particular user interaction, then the mechanism which permits that user interaction would usually be specified in this parameter. In this way the user's expectation, that once a device has been employed for the manipulation of the pop-up menu then that device can continue to be used for that purpose, is fulfilled.

It is valid to specify a user interaction as an initialization of the pop-up menu by an input mechanism which is not identified as available for interaction with the pop-up menu. This implies that the user cannot necessarily complete the interaction with the pop-up menu with that input mechanism.

For example, if a pop-up menu is initialized with a mouse button depressed but that mouse button is not identified as available for manipulating the pop-up menu, then that mouse button can manipulate the pop-up menu until it is released. Assuming that the pop-up menu is not dismissed when that mouse button is released, then the mouse button cannot be used for further interaction with the pop-up menu, since it is not identified as available for that use.

The following list shows the input device valid for interaction with the pop-up menu with each option:

PU_KEYBOARD	The keyboard.
PU_MOUSEBUTTON1	Mouse button 1.
PU_MOUSEBUTTON2	Mouse button 2.
PU_MOUSEBUTTON3	Mouse button 3.

Returns

rc (BOOL) – returns

Pop-up menu invoked indicator.

TRUE	Pop-up menu successfully invoked
FALSE	Pop-up menu not successfully invoked.

WinSetMenuItemText

This macro sets the text for Menu indexed item to buffer.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetMenuItemText  (HWND hwndMenu, USHORT usId, PSZ pText)
```

Parameters

hwndMenu (HWND) – input
Menu window handle.

usId (USHORT) – input
Identity of the menu item.

pText (PSZ) – input
Text for the menu item.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Menus.

MM_DELETEITEM

This message deletes a menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and delete it.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemsLeft (SHORT)
Number remaining.

MM_ENDMENUMODE

This message is sent to a menu control to terminate menu selection.

Parameters

param1

usdismiss (USHORT)
Dismiss menu indicator.

TRUE	Dismiss the submenu or subdialog window
FALSE	Do not dismiss the submenu or subdialog window.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

MM_INSERTITEM

This message inserts a menu item into a menu.

Parameters

param1

pmenuitem (PMENUITEM)
Menu-item data structure.

This points to a MENUITEM structure.

param2

pszItemText (PSZ)
Item text.

This points to a string containing the text to be inserted.

Returns

sIndexInserted (SHORT)
Index of inserted item.

MIT_MEMERROR	The menu control cannot allocate space to insert the menu item in the menu.
MIT_ERROR	An error other than MIT_MEMERROR occurred.
Other	The zero-based index of the offset of the item within the menu.

MM_ISITEMVALID

This message returns the selectable status of a specified menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier.

FALSE If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Selectable indication.

TRUE The user can select and enter the specified item.
FALSE The user cannot select and enter the specified item.

MM_ITEMIDFROMPOSITION

This message returns the identity of a menu item of a specified index.

Parameters

param1

sItemIndex (SHORT)
Item index.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sIdentity (SHORT)
Item identity.

MIT_ERROR	Error occurred; for example, because *sItemIndex* is not valid.
Other	Item identity.

MM_ITEMPOSITIONFROMID

This message returns the index of a menu item of a particular identity.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sIndex (SHORT)
Item index.

MIT_NONE	Item does not exist
Other	Item index.

MM_QUERYDEFAULTITEMID

This message returns the default item id for a conditional cascade menu. For any other type of menu or submenu, this message returns zero.

Parameters

param1

ulReserved (ULONG)
Reserved value, must be 0.

param2

ulReserved (ULONG)
Reserved value, must be 0.

Returns

ulDefItemID (ULONG)
Menu id of the default menu item.

MM_QUERYITEM

This message returns the definition of the specified menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus flag.

TRUE If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and copy its definition.

FALSE If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

pmenuitem (PMENUITEM)
Menu-item data structure.

This points to a MENUITEM structure.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

MM_QUERYITEMATTR

This message returns the attributes of a menu item.

Parameters

param1

usitem (USHORT)
Item identity.

usIncludeSubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and return its state.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

usattributemask (USHORT)
Attribute mask.

Returns

usState (USHORT)
State.

MM_QUERYITEMCOUNT

This message returns the number of items in the menu.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sresult (SHORT)
Item count.

MM_QUERYITEMRECT

This message returns the bounding rectangle of a menu item.

Parameters

param1

usitem (USHORT)
Item identity.

fIncludeSubmenus (BOOL)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and return its state.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

prect (PRECTL)
Bounding rectangle of the menu item in device coordinates relative to the menu window.

Returns

rc (BOOL)
Success indicator.

TRUE	Specified item was found.
FALSE	Specified item was not found.

MM_QUERYITEMTEXT

This message returns the text of the specified menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

smaxcount (SHORT)
Maximum count.

Copy the item text as a null-terminated string, but limit the number of characters copied, including the null termination character, to this value, which must be greater than 0.

param2

pszItemText (PSZ)
Buffer into which the item text is to be copied.

This points to a string (character) buffer.

Returns

sTextLength (SHORT)
Length of item text.

0	Error occurred. For example, no item of the specified identity exists or the item has no text. No text is copied.
Other	Length of item text.

MM_QUERYITEMTEXTLENGTH

This message returns the text length of the specified menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sLength (SHORT)
Length of item text.

0	Error occurred. For example, no item of the specified identity exists or the item has no text. No text is copied.
Other	Length of item text.

MM_QUERYSELITEMID

This message returns the identity of the selected menu item.

Parameters

param1

usReserve (USHORT)
Reserved value, should be 0.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for a selected item with the specified identifier.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for a selected item with the specified identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sresult (SHORT)
Selected item identifier.

MID_ERROR	Error occurred
MIT_NONE	No item selected
Other	Selected item identifier.

MM_REMOVEITEM

This message removes a menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and delete it.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemsLeft (SHORT)
Count of remaining items.

MM_SELECTITEM

This message selects or deselects a menu item.

Parameters

param1

sitem (SHORT)
Item identifier.

MIT_NONE	Deselect all the items in the menu.
Other	Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and select or deselect it.

FALSE If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

usReserve (USHORT)
Reserved value, should be 0.

usdismissed (USHORT)
Dismissed flag.

TRUE	Dismiss the menu
FALSE	Do not dismiss the menu.

Returns

rc (BOOL)
Success indicator.

TRUE A selection has been made, or *sitem* is MIT_NONE.
FALSE A selection has not been made, or a deselection has been made, or *sitem* is not MIT_NONE.

MM_SETDEFAULTITEMID

This message is used to set the default item in a conditional cascade menu.

Parameters

param1

ulDefItemID (ULONG)
The menu id of the item to become the new default.

param2

ulReserved (ULONG)
Reserved value, must be 0.

Returns

rc (BOOL)
Success of failure indicator.

TRUE	The conditional cascade default was set.
FALSE	The conditional cascade default was not set.

MM_SETITEM

This message sets the definition of a menu item.

Parameters

param1

usReserve (USHORT)
Reserved value, should be 0.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE — If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and set its definition.

FALSE — If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

pmenuitem (PMENUITEM)
Menu-item data structure.

This points to a MENUITEM structure.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

MM_SETITEMATTR

This message sets the attributes of a menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

usincludesubmenus (USHORT)
Include submenus indicator.

TRUE	If the menu does not have an item with the specified identifier, search the submenus and subdialogs of the menu for an item with the specified identifier and set its attributes.
FALSE	If the menu does not have an item with the specified identifier, do not search the submenus and subdialogs of the menu for an item with the specified identifier.

param2

usattributemask (USHORT)
Attribute mask.

usattributedata (USHORT)
Attribute data.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

MM_SETITEMHANDLE

This message sets the handle of a menu item.

Parameters

param1

usitem (USHORT)
Item index.

param2

ulitemhandle (ULONG)
Item handle.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

MM_SETITEMTEXT

This message sets the text of a menu item.

Parameters

param1

usitem (USHORT)
Item identifier.

param2

pszItemText (PSZ)
Item text.

This points to a string containing the text to set the menu item to.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

MM_STARTMENUMODE

This message is used to begin menu selection.

Parameters

param1

usshowsubmenu (USHORT)
Show submenu flag.

TRUE Show the submenu (pull-down menu) of the selected action bar item when the menu enters selection mode. If the action bar is not visible, the submenu is shown, otherwise it is not shown. If the item selected does not have a submenu, this parameter is ignored.

FALSE Do not show the submenu (pull-down menu) of the selected action bar item when the menu enters selection mode.

usresumemenu (USHORT)
Resume menu mode flag.

TRUE Resume the user interaction with the menu from where it left off. The menu is assumed to have been used previously and left without dismissing one of the submenus, and therefore is resumed in that submenu.

FALSE Begin user interaction with the menu from the action bar, subject to the value of the *usshowsubmenu* parameter.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_CONTEXTMENU

This message occurs when the operator requests a pop-up menu.

Parameters

param1

ptspointerpos (POINTS)
Pointer position.

The pointer position is in window coordinates relative to the bottom-left corner of the window. This value is ignored if *fPointer* is not set to TRUE.

param2

usReserved (USHORT)
Reserved value, 0.

fPointer (USHORT)
Input device flag.

TRUE	Message resulted from keyboard event.
FALSE	Message resulted from mouse pointer event.

Returns

rc (BOOL)
Processed indicator.

TRUE	Message processed
FALSE	Message ignored.

WM_INITMENU

This message occurs when a menu control is about to become active.

Parameters

param1

smenuid (SHORT)
Menu-control identifier.

param2

hwnd (HWND)
Menu-window handle.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_MENUEND

This message occurs when a menu control is about to terminate.

Parameters

param1

usmenuid (USHORT)
Menu-control identifier.

param2

hwnd (HWND)
Menu-control window handle.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Menus.

MENUITEM

Menu item.

Syntax

```
typedef struct _MENUITEM {
SHORT        iPosition;
USHORT       afStyle;
USHORT       afAttribute;
USHORT       id;
HWND         hwndSubMenu;
ULONG        hItem;
 } MENUITEM;

typedef MENUITEM *PMENUITEM;
```

Fields

iPosition (SHORT)
Position.

afStyle (USHORT)
Style.

afAttribute (USHORT)
Attribute.

id (USHORT)
Identity.

hwndSubMenu (HWND)
Submenu.

hItem (ULONG)
Item.

OWNERITEM

Owner item.

Syntax

```
typedef struct _OWNERITEM {
HWND        hwnd;
HPS         hps;
ULONG       fsState;
ULONG       fsAttribute;
ULONG       fsStateOld;
ULONG       fsAttributeOld;
RECTL       rclItem;
LONG        idItem;
ULONG       hItem;
 } OWNERITEM;

typedef OWNERITEM *POWNERITEM;
```

Fields

hwnd (HWND)
Window handle.

hps (HPS)
Presentation-space handle.

fsState (ULONG)
State.

fsAttribute (ULONG)
Attribute.

fsStateOld (ULONG)
Old state.

fsAttributeOld (ULONG)
Old attribute.

rclItem (RECTL)
Item rectangle.

idItem (LONG)
Item identity.

hItem (ULONG)
Item.

Summary

This section lists the OS/2 functions, messages, and structures used with menus.

Table 13-2. Menu Functions

Function Name	Description
WinCheckMenuItem	Sets the check state of the specified menu item to the flag.
WinCreateMenu	Creates a menu window from the menu template.
WinEnableMenuItem	Sets the state of the specified menu item to the enable flag.
WinIsMenuItemChecked	Returns the state (checked/not checked) of the identified menu item.
WinIsMenuItemEnabled	Returns the state (enable/disable) of the specified menu item.
WinIsMenuItemValid	Returns TRUE if the specified item is a valid choice.
WinLoadMenu	Creates a menu window from the menu template *Menuid* from *Resource*, and returns in *Menu* the window handle for the created window.
WinPopupMenu	Displays a pop-up menu.
WinSetMenuItemText	Sets the text for menu indexed item to buffer.

Table 13-3 (Page 1 of 2). Messages Received by a Menu

Message	Description
MM_DELETEITEM	Deletes a menu item.
MM_ENDMENUMODE	Sent to a menu control to terminate menu selection.
MM_INSERTITEM	Inserts a menu item in a menu.
MM_ISITEMVALID	Returns the selectable status of a specified menu item.
MM_ITEMIDFROMPOSITION	Returns the identity of a menu item of a specified index.
MM_ITEMPOSITIONFROMID	Returns the index of a menu item of a particular identify.
MM_QUERYDEFAULTITEMID	Returns the default item id for a conditional cascade menu.
MM_QUERYITEM	Returns the definition of the specified menu item.
MM_QUERYITEMATTR	Returns the attributes of a menu item.
MM_QUERYITEMCOUNT	Returns the number of items in the menu.
MM_QUERYITEMRECT	Returns the bounding rectangle of a menu item.
MM_QUERYITEMTEXT	Returns the text of the specified menu item.
MM_QUERYITEMTEXTLENGTH	Returns the text length of the specified menu item.
MM_QUERYSELITEMID	Returns the identity of the selected menu item.
MM_REMOVEITEM	Removes a menu item.

Table 13-3 (Page 2 of 2). Messages Received by a Menu

Message	Description
MM_SELECTITEM	Selects or deselects a menu item.
MM_SETDEFAULTITEMID	Used to set the default item in a conditional cascade menu.
MM_SETITEM	Sets the definition of a menu item.
MM_SETITEMATTR	Sets the attributes of a menu item.
MM_SETITEMHANDLE	Sets the handle of a menu item.
MM_SETITEMTEXT	Sets the text of a menu item.
MM_STARTMENUMODE	Used to begin menu selection.

Table 13-4 (Page 1 of 2). Messages Generated by a Menu

Message	Description
WM_ADJUSTWINDOWPOS	Sent by WinSetWindowPos to enable the window to adjust its new position or size whenever it is about to be moved.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_COMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table.
WM_CONTEXTMENU	Occurs when the operator requests a pop-up menu.
WM_CONTROLPOINTER	Sent to the owner window of a control when the pointing device pointer moves over the control window, enabling the owner to set the pointer.
WM_CREATE	Occurs when an application requests the creation of a window.
WM_DESTROY	Occurs when an application requests the destruction of a window.
WM_DRAWITEM	Sent to the owner of a menu control each time an item is to be drawn.
WM_ENABLE	Sets the enable state of a window.
WM_FOCUSCHANGE	Occurs when the window possessing the focus is changed.
WM_HELP	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_HELP.
WM_INITMENU	Occurs when a menu control is about to become active.
WM_MEASUREITEM	Sent to the owner of a meu control to establish the height for an item in that control.

Table 13-4 (Page 2 of 2). Messages Generated by a Menu

Message	Description
WM_MENUEND	Occurs when a menu control is about to terminate.
WM_MENUSELECT	Occurs when a menu item has been selected.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_NEXTMENU	Occurs when either the beginning or the end of the menu is reached using the cursor control keys.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_SETFOCUS	Occurs when a window is to receive or lose the input focus.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the menu parameters.
WM_SYSCOMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_SYSCOMMAND.

Table 13-5. Menu Structures

Structure Name	Description
MENUITEM	Menu item.
OWNERITEM	Owner item.

Chapter 14. Keyboard Accelerators

A *keyboard accelerator* (*shortcut key* to the user) is a keystroke that generates a command message for an application. This chapter describes how to use keyboard accelerators in your PM applications.

About Keyboard Accelerators

Using a keyboard accelerator has the same effect as choosing a menu item. While menus provide an easy way to learn an application's command set, accelerators provide quick access to those commands.

Without accelerators, a user might generate commands by pressing the Alt key to access the menu bar, using the Arrow keys to select an item, then pressing the Enter key to choose the item. In contrast, accelerators allow the user to generate commands *with a single keystroke*. Figure 14-1 shows examples of accelerators.

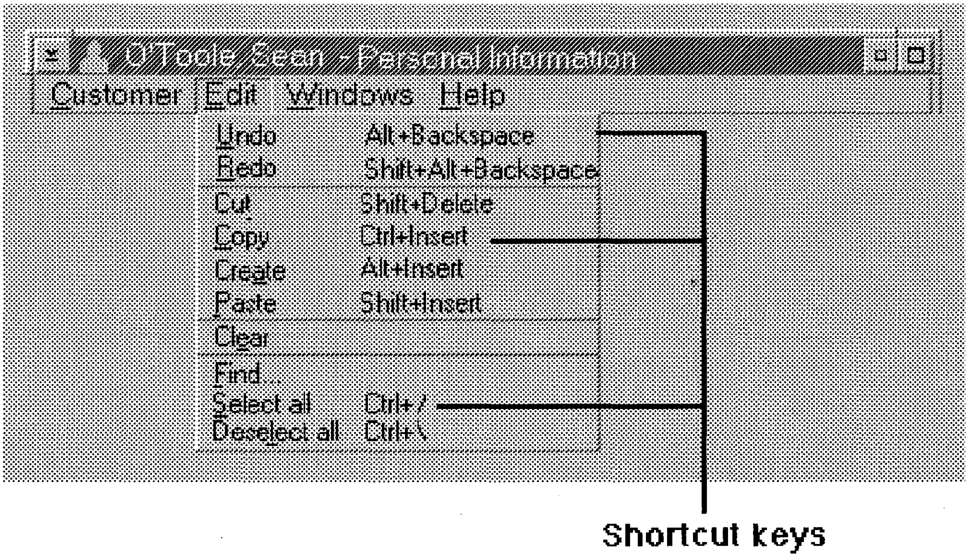


Figure 14-1. Accelerators

Like menu items, accelerators can generate WM_COMMAND, WM_HELP, and WM_SYSCOMMAND messages. Although, normally, accelerators are used to generate existing commands as menu items, they also can send commands that have no menu-item equivalent.

Accelerator Tables

An accelerator table contains an array of accelerators. Accelerator tables exist at two levels within the operating system: a single accelerator table for the system queue and individual accelerator tables for application windows. Accelerators in the system queue apply to all applications–for example, the F1 key always generates a WM_HELP message. Having accelerators for individual application windows ensures that an application can define its own accelerators without interfering with other applications. An accelerator for an application window can override the accelerator in the system queue. An application can modify both its own accelerator table and the system's accelerator table.

The application can set and query the accelerator table for a specific window or for the entire system. For example, an application can query the system accelerator table, copy it, modify the copied data structures; and then, use the modified copy to set the system accelerator table. An application also can modify its window's accelerator table at run time to respond more appropriately to the current environment.

Note: An application that modifies any accelerator table other than its own should maintain the original accelerator table; and, before terminating, restore that table.

Accelerator-Table Resources

You can use accelerators in an application by creating an accelerator-table resource in a resource-definition file. Then, when the application creates a standard frame window, the application can associate that window with the resource.

As specified in a resource-definition file, an accelerator table consists of a list of accelerator items, each defining the keystroke that triggers the accelerator, the command the accelerator generates, and the accelerator's style. The style specifies whether the keystroke is a virtual key, a character, or a scan code, and whether the generated message is WM_COMMAND, WM_SYSCOMMAND, or WM_HELP; WM_COMMAND is the default.

Accelerator-Table Handles

Applications that use accelerator tables refer to them with a 32-bit handle. An application using this handle, by default, can make most API function calls for accelerators without having to account for the internal structures that define the accelerator table. When an application needs to dynamically create or change an accelerator table, it must use the ACCEL and ACCELTABLE data structures.

Accelerator-Table Data Structures

An accelerator table consists of individual accelerator items. Each item in the table is represented by an ACCEL structure that defines the accelerator's style, keystroke, and command identifier. Typically, an application defines these aspects of an accelerator in a resource-definition file, but the ACCEL structure also can be built in memory at run time.

An accelerator table is represented by an ACCELTABLE structure that specifies the number of accelerator items in the table, the code page used for the keystrokes in the accelerator items, and an array of ACCEL structures (one for each item in the table). Applications that use ACCELTABLE structures directly must allocate sufficient memory to hold all the items in the table.

Accelerator-Item Styles

An accelerator item has a style that determines what combination of keys produces the accelerator and what command message is generated by the accelerator. An application can specify the following accelerator-item styles in the *fs* field of the ACCEL structure:

Table 14-1. Accelerator-Item Styles

Style	Description
AF_ALT	Specifies that the user must hold down the Alt key while pressing the accelerator key.
AF_CHAR	Specifies that the keystroke is a character that is translated using the code page for the accelerator table. (This is the default style.)
AF_CONTROL	Specifies that the user must hold down the Ctrl key while pressing the accelerator key.
AF_HELP	Specifies that the accelerator generates a WM_HELP message instead of a WM_COMMAND message.
AF_LONEKEY	Specifies that the user need not press another key while the accelerator key is down. Typically, this style is used with the Alt key to specify that simply pressing and releasing that key triggers the accelerator.
AF_SCANCODE	Specifies that the keystroke is an untranslated scan code from the keyboard.
AF_SHIFT	Specifies that the user must hold down the Shift key when pressing the accelerator key.
AF_SYSCOMMAND	Specifies that the accelerator generates a WM_SYSCOMMAND message instead of a WM_COMMAND message.
AF_VIRTUALKEY	Specifies that the keystroke is a virtual key–for example, the F1 function key.

Using Keyboard Accelerators

This section explains how to perform the following tasks:

- Create an accelerator-table resource.
- Include an accelerator table in a frame window.
- Modify an accelerator table.

Creating an Accelerator-Table Resource

The following code fragment shows a typical accelerator-table resource:

```
ACCELTABLE ID_ACCEL_RESOURCE
BEGIN
    VK_ESC,    IDM_ED_UNDO,  AF_VIRTUALKEY | AF_SHIFT
    VK_DELETE, IDM_ED_CUT,   AF_VIRTUALKEY
    VK_F2,     IDM_ED_COPY,  AF_VIRTUALKEY
    VK_INSERT, IDM_ED_PASTE, AF_VIRTUALKEY
END
```

This accelerator table has four accelerator items. The first one is triggered when the user presses Shift+Esc, which sends a WM_COMMAND message (the default).

An accelerator table in a resource-definition file has an identifier (ID_ACCEL_RESOURCE in the previous example). You can associate an accelerator-table resource with a standard frame window by specifying the table's resource identifier as the *idResources* parameter of the WinCreateStdWindow function.

An application can load an accelerator table resource-definition file automatically when creating a standard frame window, or it can load the resource independently and associate it with a window or with the entire system.

Including an Accelerator Table in a Frame Window

You can add an accelerator table to a frame window either by using the WinSetAccelTable function or by defining an accelerator-table resource (as shown in the previous section) and creating a frame window with the FCF_ACCELTABLE frame style. The second method is shown in the following code fragment:

```
HWND  hwndFrame,hwndClient;
CHAR  szClassName[]="MyClass";
CHAR  szTitle[]="MyWindow";

ULONG flControlStyle = FCF_SIZEBORDER | FCF_ACCELTABLE |
                       FCF_TITLEBAR   | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &flControlStyle,
    szClassName,
    szTitle,
    0,
    (HMODULE)NULL,
    ID_MENU_RESOURCE,
    &hwndClient);
```

Notice that if you set the *flControlStyle* parameter to the FCF_STANDARD flag, you must define an accelerator-table resource, because FCF_STANDARD includes the FCF_ACCELTABLE flag.

If the window being created also has a menu, the menu resource and accelerator resource must have the same resource identifier; this is because the WinCreateStdWindow function has only one input parameter to specify the resource identifiers for menus, accelerator tables, and icons. If an application creates an accelerator table resource-definition file; then, opens a standard frame window (as shown in the preceding example), the accelerator table is installed automatically in the window's message queue, and keyboard events are translated during the normal processing of events. The application simply responds to WM_COMMAND, WM_SYSCOMMAND, and WM_HELP messages; it does not matter whether these messages come from a menu or an accelerator.

An application also can add an accelerator table to a window by calling the WinSetAccelTable function with an accelerator-table handle and a frame-window handle. The application can call either the WinLoadAccelTable function to retrieve an accelerator table from a resource file or the WinCreateAccelTable function to create an accelerator table from an accelerator-table data structure in memory.

Modifying an Accelerator Table

You can modify an accelerator table, for either your application windows or the system, by doing the following:

1. Retrieve the handle of the accelerator table.
2. Use that handle to copy the accelerator-table data to an application-supplied buffer.
3. Change the data in the buffer.
4. Use the changed data to create a new accelerator table.

Then you can use the new accelerator-table handle to set the accelerator table, as outlined in the following list:

1. Call WinQueryAccelTable to retrieve an accelerator-table handle.
2. Call WinCopyAccelTable with a NULL buffer handle to determine how many bytes are in the table.
3. Allocate sufficient memory for the accelerator-table data.
4. Call WinCopyAccelTable, with a pointer to the allocated memory.
5. Modify the data in the buffer (assuming it has the form of an ACCELTABLE structure).
6. Call WinCreateAccelTable, passing a pointer to the buffer with the modified accelerator-table data.
7. Call WinSetAccelTable with the handle returned by WinCreateAccelTable.

Related Functions

This section covers the functions that are related to Keyboard Accelerators.

WinCopyAccelTable

This function is used to get the accelerator-table data corresponding to an accelerator-table handle, or to determine the size of the accelerator-table data.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG WinCopyAccelTable  (HACCEL hAccel, PACCELTABLE pacctAccelTable,
                          ULONG ulCopyMax)
```

Parameters

hAccel (HACCEL) – input
Accelerator-table handle.

pacctAccelTable (PACCELTABLE) – in/out
Accelerator-table data area.

NULL Return the size, in bytes, of the complete accelerator table, and ignore the *ulCopyMax* parameter.

Other Copy up to *ulCopyMax* bytes of the accelerator table into this data area.

ulCopyMax (ULONG) – input
Maximum data area size.

Returns

ulCopied (ULONG) – returns
Amount copied or size required.

Other Amount of data copied into the data area, or the size of data area required for the complete accelerator table.

0 Error occurred.

WinCreateAccelTable

This function creates an accelerator table from the accelerator definitions in memory.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HACCEL WinCreateAccelTable  (HAB hab, PACCELTABLE pacctAccelTable)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pacctAccelTable (PACCELTABLE) – input
Accelerator table.

Returns

haccelhAccel (HACCEL) – returns
Accelerator-table handle.

WinDestroyAccelTable

This function destroys an accelerator table.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinDestroyAccelTable  (HACCEL haccelAccel)
```

Parameters

haccelAccel (HACCEL) – input
Accelerator-table handle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinLoadAccelTable

This function loads an accelerator table.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HACCEL WinLoadAccelTable  (HAB hab, HMODULE Resource,
                           ULONG idAccelTable)
```

Parameters

hab (HAB) – input
Anchor-block handle.

Resource (HMODULE) – input
Resource identity containing the accelerator table.

idAccelTable (ULONG) – input
Accelerator-table identifier, within the resource file.

Returns

haccelAccel (HACCEL) – returns
Accelerator-table handle.

WinQueryAccelTable

This function queries the window or queue accelerator table.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HACCEL WinQueryAccelTable  (HAB hab, HWND hwndFrame)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwndFrame (HWND) – input
Frame-window handle.

NULLHANDLE	Return queue accelerator.
Other	Return the window accelerator table, by sending the WM_QUERYACCELTABLE message to *hwndFrame*.

Returns

haccelAccel (HACCEL) – returns
Accelerator-table handle.

NULLHANDLE	Error occurred
Other	Accelerator-table handle.

WinSetAccelTable

This function sets the window-accelerator, or queue-accelerator table.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
```

BOOL WinSetAccelTable (HAB hab, HACCEL haccelAccel, HWND hwndFrame)

Parameters

hab (HAB) – input
Anchor-block handle.

haccelAccel (HACCEL) – input
Accelerator-table handle.

NULLHANDLE	Remove any accelerator table in effect for the window or the queue
Other	Accelerator-table handle.

hwndFrame (HWND) – input
Frame-window handle.

NULLHANDLE	Set the queue-accelerator table
Other	Set the window-accelerator table.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinTranslateAccel

This function translates a WM_CHAR message.

Syntax

```
#define INCL_WINACCELERATORS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinTranslateAccel  (HAB hab, HWND hwnd, HACCEL haccelAccel,
                         PQMSG pQmsg)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwnd (HWND) – input
Destination window.

haccelAccel (HACCEL) – input
Accelerator-table handle.

pQmsg (PQMSG) – in/out
Message to be translated.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Keyboard Accelerators.

WM_QUERYACCELTABLE

This message returns the handle to the accelerator table of a window.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

haccel (HACCEL)
Accelerator table handle.

NULLHANDLE	No accelerator table is associated with the window.
Other	The handle of the accelerator table associated with the window.

WM_SETACCELTABLE

This message establishes the window accelerator table to be used for translation, when the window is active.

Parameters

param1

haccelNew (HACCEL)
New accelerator table.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Data Structures

This section covers the data structures that are related to Keyboard Accelerators.

ACCEL

Accelerator structure.

Syntax

```
typedef struct _ACCEL {
USHORT      fs;
USHORT      key;
USHORT      cmd;
 } ACCEL;

typedef ACCEL *PACCEL;
```

Fields

fs (USHORT)
Options.

key (USHORT)
Key.

cmd (USHORT)
Command code.

The value to be placed in the *uscmd* parameter of a WM_HELP, a WM_COMMAND, or a WM_SYSCOMMAND.

ACCELTABLE

Accelerator-table structure.

Syntax

```
typedef struct _ACCELTABLE {
USHORT      cAccel;
USHORT      codepage;
ACCEL       aaccel[1];
 } ACCELTABLE;

typedef ACCELTABLE *PACCELTABLE;
```

Fields

cAccel (USHORT)

Number of accelerator entries.

codepage (USHORT)

Code page for accelerator entries.

aaccel[1] (ACCEL)

Accelerator entries.

The default accelerator table has the following 16 entries:

Options		Key	Command
HELP	VIRTUALKEY	VK_F1	0
SYSCOMMAND ALT	VIRTUALKEY	VK_F4	SC_CLOSE
SYSCOMMAND ALT	VIRTUALKEY	VK_ENTER	SC_RESTORE
SYSCOMMAND ALT	VIRTUALKEY	VK_NEWLINE	SC_RESTORE
SYSCOMMAND ALT	VIRTUALKEY	VK_F5	SC_RESTORE
SYSCOMMAND ALT	VIRTUALKEY	VK_F6	SC_NEXTFRAME
SYSCOMMAND ALT	VIRTUALKEY	VK_F7	SC_MOVE
SYSCOMMAND ALT	VIRTUALKEY	VK_F8	SC_SIZE
SYSCOMMAND ALT	VIRTUALKEY	VK_F9	SC_MINIMIZE
SYSCOMMAND ALT	VIRTUALKEY	VK_F10	SC_MAXIMIZE
SYSCOMMAND	VIRTUALKEY	VK_F10	SC_APPMENU
SYSCOMMAND LONEKEY	VIRTUALKEY	VK_ALT	SC_APPMENU
SYSCOMMAND LONEKEY	VIRTUALKEY	VK_ALTGRAF	SC_APPMENU
SYSCOMMAND ALT	VIRTUALKEY	VK_SPACE	SC_SYSMENU
SYSCOMMAND SHIFT	VIRTUALKEY	VK_ESC	SC_SYSMENU
SYSCOMMAND CONTROL	VIRTUALKEY	VK_ESC	SC_TASKMANAGER

Summary

Following are the OS/2 functions, messages, and structures used with accelerator tables:

Table 14-2. Accelerator-Table Functions

Function name	Description
WinCopyAccelTable	Used to get the accelerator table corresponding to an accelerator-table handle, or to determine the size of the accelerator-table data.
WinCreateAccelTable	Creates an accelerator table from the accelerator definitions in memory.
WinDestroyAccelTable	Destroys an accelerator table.
WinLoadAccelTable	Loads an accelerator table.
WinQueryAccelTable	Queries the window or queue accelerator table.
WinSetAccelTable	Sets the window-accelerator or queue-accelerator table.
WinTranslateAccel	Translates a WM_CHAR message.

Table 14-3. Accelerator-Table Messages

Message	Description
WM_QUERYACCELTABLE	Returns the handle to a window's accelerator table.
WM_SETACCELTABLE	Establishes the window accelerator table to be used for translation when the window is active.
WM_TRANSLATEACCEL	Sent to the focus window when a WM_CHAR message occurs.

Table 14-4. Accelerator-Table Structures

Structure name	Description
ACCEL	Accelerator structure.
ACCELTABLE	Accelerator-table structure.

Chapter 15. Dialog Windows

Dialog windows (also called *dialog boxes*) provide a high-level method for applications to display and gather information. This chapter describes the creation and use of dialog windows and message boxes in your PM applications.

Note: *Dialog windows, dialog boxes*, and *message boxes* all are *secondary windows* to the user.

About Dialog Windows

A dialog window is a temporary window that contains one or more control windows and, typically, is used to display messages to and gather input from the user. An application usually destroys a dialog window immediately after using it.

The OS/2 operating system contains many functions and messages that help manage the control windows that make up a dialog window, thereby easing the burden of maintaining complex input and output systems.

Modal and Modeless Dialog Windows

Dialog windows can be modal or modeless. A *modal* dialog window requires that the dialog window be dismissed before the user can activate other windows in the same application. Generally, an application uses a modal dialog window to get essential information from the user before proceeding with an operation. A *modeless* dialog window allows the user to activate other windows in the same application without dismissing the dialog window. Both modal and modeless dialog windows allow the user to activate windows in another application before responding to the dialog window.

Modal dialog windows are easier for an application to manage because they are created, perform their task, and are closed, all with a single function call.

Modeless dialog windows require more attention from the application because they exist until explicitly dismissed. Modeless dialog windows provide a more flexible interface, however, by allowing the user to move to other windows in the application before responding to the dialog window.

Dialog Items

A *dialog item* is a child window of the dialog window, which usually is a window of class WC_FRAME. The operating system provides many predefined window classes, called *control windows*, that you can use as dialog items. Figure 15-1 on page 15-2 is an example.

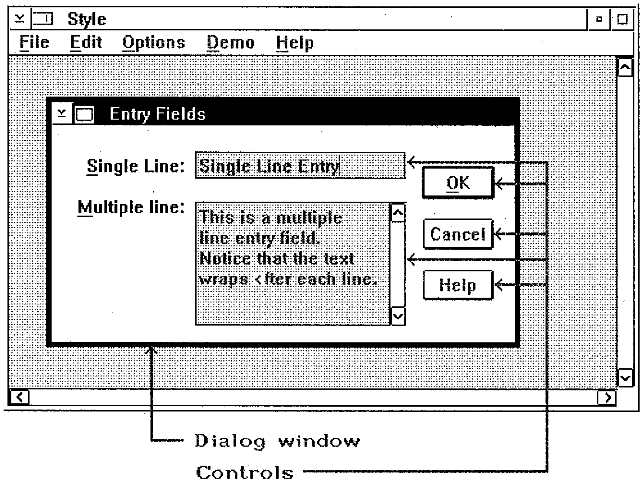

Style
File Edit Options Demo Help
Entry Fields
Single Line:
Single Line Entry
Multiple line:
This is a multiple line entry field. Notice that the text wraps <fter each line.
OK
Cancel
Help
Dialog window
Controls

Figure 15-1. Dialog Window with Control Windows

Predefined control windows include static display boxes, text-entry fields, buttons, and list boxes. You also can use customized window classes as dialog items.

Dialog items are windows and, thus, can be manipulated by all window-management functions relating to size, position, and visibility. Dialog items always are owned by the dialog frame window. Most predefined control-window classes send notification messages to their owners when the user interacts with their control windows. The dialog frame window receives these notification messages and passes them to the application through the application-defined dialog procedure.

Dialog-Item Groups

Items within a dialog window can be organized into *dialog-item groups*. When items are arranged in a group, the user can move from one item to another in the same group by using the direction keys. When the user presses a direction key, the focus will not shift to items in other groups within the dialog window.

Arranging items in groups is useful for radio buttons and check boxes. Although some control types also can be displayed this way, entry-field controls cannot; they process direction keys themselves, as do MLE, value-set, container, slider, and notebook controls.

The first item in a dialog-item group has the WS_GROUP window style. All subsequent items in the dialog template are considered part of that group until another item is given the WS_GROUP style, which begins a new group.

The WS_TABSTOP style often is used along with the WS_GROUP style. WS_TABSTOP marks the items that can receive the focus when the user presses the Tab key. Each time the user presses the Tab key, the focus moves to the next item that has the WS_TABSTOP style. Generally, the WS_GROUP and WS_TABSTOP styles are defined together for the first item of each group in the dialog template. This makes it possible for a user to press the

Tab key to move among groups of items and to use the direction keys to move among items in a group.

The WS_TABSTOP style should not be used for radio buttons because the system automatically maintains a tab stop on any selected item in a radio-button group; therefore, when the Tab key is pressed in a group of radio buttons, the focus remains on the currently selected item.

The WS_GROUP and WS_TABSTOP styles are also useful for preventing the user from moving to a particular button when using the keyboard. For example, if the dialog window has **OK** and **Cancel** push buttons, they should be in the same group, with the **OK** push button as the first item in the group. The user can press Tab to select the **OK** push button but not the **Cancel** push button. To move to the **Cancel** button using the keyboard, the user first must press the Tab key to move to the **OK** push button, and then press a direction key to move the focus to the **Cancel** push button.

Message Boxes

Message boxes are dialog windows predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. Message boxes are best for short notification messages that require a simple acknowledgment or choice by the user. Applications do not specify a dialog procedure for message boxes, so they cannot readily change the action of a message box. However, there is no need to do so, since there are many predefined message-box styles. There are two types of message boxes available to applications: standard and enhanced.

Standard Message Boxes

To generate a standard message box, an application calls WinMessageBox and specifies the type of message box and message text. The system displays the message and waits for the user to dismiss the message box by selecting a button in the message box. The system then returns a result code to the application, indicating which button the user selected.

Standard message boxes are always modal—either application-modal or system-modal. *Application-modal* (the default style) means that the user cannot activate another window in the current application before responding to the message box but can switch to another application. *System-modal* means that the user cannot activate another window in any application before responding to the message box. A system-modal message box should be used only to display urgent error messages (running out of memory, for example). Figure 15-2 on page 15-4 shows a sample standard message box.

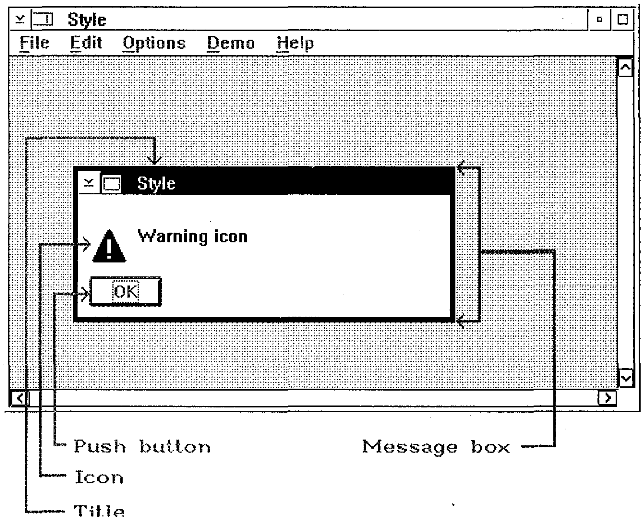


Figure 15-2. Example of a Standard Message Box

Enhanced Message Boxes

To generate an enhanced message box, an application calls WinMessageBox2. An enhanced message box has all the functionality of the standard message box, with the addition of the following:

- It can be modeless.
- Its buttons can be customized with text and icons.
- It can support customized icons in the window icon field.

Figure 15-3 shows a sample enhanced message box.

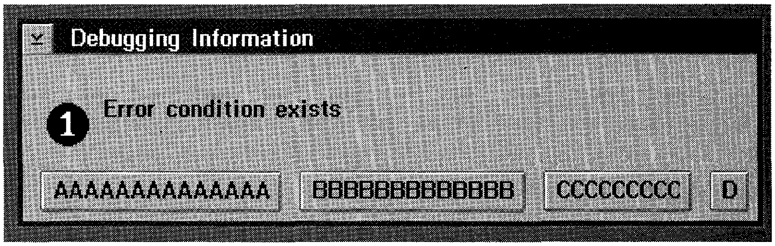


Figure 15-3. Example of an Enhanced Message Box

In creating enhanced message boxes, the MB2INFO button information block structure is used to specify button style flags as shown in the following table:

Table 15-1. MB2INFO Button Style Flags

Style Name	Description
MB_APPLMODAL	Message box is application modal. This is the default case.
MB_CUSTOMICON	A user-specified value.
MB_ERROR	Message box contains a STOP sign (white background).
MB_ICONASTERISK	Message box contains an asterisk icon.
MB_ICONEXCLAMATION	Message box contains an exclamation point icon.
MB_ICONHAND	Message box contains a hand icon.
MB_ICONQUESTION	Message box contains a question mark icon.
MB_INFORMATION	Message box contains a black "i" in a box.
MB_MOVEABLE	Message box is moveable. A title bar and system menu with Move, Close and Task Manager choices are displayed.
MB_NOICON	Message box does not contain an icon.
MB_NONMODAL	Message box is modeless (the program continues after displaying the message box).
MB_QUERY	Message box contains a question mark in a box.
MB_SYSTEMMODAL	Message box is system modal.
MB_WARNING	Message box contains a black "!" in a box.

Minimizing Dialog Windows

Whenever the dialog window is to be minimized to the desktop (as opposed to the Minimized Window Viewer), the program should hide all its child control windows, then restore them when the dialog needs to be restored.

Dialog Data Structures

Each item in a dialog window is described by a DLGTITEM data structure. This structure is rarely accessed directly by an application, since system functions handle most of the manipulation of dialog items. Applications that create dialog items that are not defined as part of a dialog-template resource must create dialog-window-item structures in memory.

A dialog window can have many items, so applications can use another structure, DLGTEMPLATE, to define the items. This structure consists of header information, followed by an array of dialog-window items. Applications that create dialog windows without using dialog resources must create a dialog template in memory, and, then, call the WinCreateDlg function.

Dialog Resources

Most applications define dialog templates in resource files rather than constructing template data structures in memory at run time. The dialog resource file defines the size and style of the dialog-window frame and specifies each dialog item.

The dimensions and position of each dialog item are specified in dialog coordinates, which are based on the size of the system font. A horizontal unit is one-fourth the average width of the characters in the system font; a vertical unit is one-eighth the average height of the characters in the system font. The origin of the dialog template is the lower-left corner of the dialog window. The operating system provides the WinMapDlgPoints function for converting dialog coordinates into window coordinates.

Using Message Boxes and Dialog Windows

The simplest dialog window is the message box. Most message boxes present simple messages and offer the user one, two, or three responses (represented by buttons). A message box is easy to use and is appropriate when an application requires a clearly defined response to a static message. However, standard message boxes lack flexibility in size and placement on the screen and are limited in the choices they offer the user. Applications that require more control over the size, position, and content should use enhanced message boxes or regular Dialog Windows instead of standard message boxes.

Creating a Standard Message Box

There are three parts to a message box: the icon, the message, and buttons. Applications specify the icons and buttons by using message-box style constants. Message text is specified by a null-terminated string.

To create a message box, the application calls WinMessageBox, which displays the message box and processes user input until the user selects a button in the message box. The WinMessageBox return value indicates which button the user selected.

The following code fragment illustrates how to create a message box with a default **Yes** button, a **No** button, and a question-mark (?) icon. This example assumes that you have defined a string resource with the MY_MESSAGESTR_ID identifier in the resource file.

```
UCHAR  szMessageString[255];
ULONG  ulResult;

WinLoadString(hab, (HMODULE) NULL, MY_MESSAGESTR_ID,
    sizeof(szMessageString), szMessageString);

ulResult = WinMessageBox(hwndFrame,   /* Parent    */
    hwndFrame,                        /* Owner     */
    szMessageString,                  /* Text      */
    (PSZ) NULL,                       /* caption   */
    MY_MESSAGEWIN,                    /* Window ID */
    MB_YESNO |
    MB_ICONQUESTION |
    MB_DEFBUTTON1);                   /* Style     */

 if (ulResult == MBID_YES) {

      /* Do yes case. */

 } else {

      /* Do no case.  */
 }
```

The WinMessageBox function returns predefined values indicating which button has been selected.

Notice that strings for message boxes should be defined as string resources to facilitate program translation for other countries. However, there is danger in using string resources in message boxes that are called in low-memory situations; loading a string resource in such situations could result in severe memory problems and cause an application to fail. One way to prevent this problem is to preload the string resource and make it nondiscardable so it will be available when the message box must be displayed.

Creating a System-Modal Standard Message Box

There are two levels of modality for system-modal message boxes—*soft* modal and *hard* modal. A soft-modal message box does not allow keystrokes or mouse input to reach any other window but does allow other messages, such as deactivation and timer messages, to reach other windows. A hard-modal message box does not allow any messages to reach other windows. A hard-model message box is appropriate for serious system warnings.

To create a hard-modal message box, combine the MB_ICONHAND style with the MB_SYSTEMMODAL style. To create a soft-modal message box, use the MB_SYSTEMMODAL style with any style other than MB_ICONHAND. The MB_SYSTEMMODAL icon always is in memory and is available even in low-memory situations.

Creating an Enhanced Message Box

WinMessageBox2 creates a message window that can be used to display error messages and ask questions. It is a more powerful version of WinMessageBox, including options for non-modality and customization of buttons with text and icons or mini-icons. Buttons included in the enhanced message box are specified in the button definition array MB2D, where custom text can be added.

To support the use of the MB_NONMODAL style, two notification messages are used:

WM_MSGBOXINIT This message notifies the owner of the message when a non-modal message box is being displayed. It is the responsibility of the owner window to store the window handle returned by the function for later use when the message box is to be destroyed.

WM_MSGBOXDISMISS This message notifies the owner of the message when a non-modal message box has been dismissed. It is the parent window's responsibility to destroy the message box.

The following example uses WinMessageBox2 to create a message box containing a customized icon:

```
#define  INCL_WINDIALOGS          /* Window Dialog Manager Functions   */
#define  INCL_WINPOINTERS         /* Window Pointer Functions          */

#include <os2.h>
#include <stdio.h>
#include <string.h>

CHAR      szMsg[100];         /* Message
HWND      hwndClient;             /* Client-window handle                */
MB2INFO   mb2info;                /* Message Box input structure         */

MB2D mb2d[4] = {                  /* Array of button definitions*/
    { "AAAA", ID_BUTTON1, BS_DEFAULT},
    { "BBBB", ID_BUTTON2, 0},
    { "CCCC",  ID_BUTTON3, 0},
    { "DDDD", ID_BUTTON4, 0}
};

 mb2info.hIcon = WinLoadPointer(HWND_DESKTOP, 0, ID_ICON1);
 mb2info.cButtons = 4;            /* Number of buttons   */
 mb2info.flStyle = MB_CUSTOMICON | MB_MOVEABLE;
                                  /* Icon style flags    */
 mb2info.hwndNotify = NULLHANDLE; /* Reserved            */
 mb2info.cb = sizeof(MB2INFO) + ((mb2info.cButons >1) ?
              (mb2info.cButtons -1) * sizeof (MB2D) : 0);

 memcpy (&mb2info.mb2d, &mb2d, mb2info.cb);

 mb2info.pmb2d = mb2d;            /* Array of button definitions*/

 sprintf (&szMsg, %s, "Error condition exists");

 WinMessageBox2(HWND_DESKTOP,
                hwndClient,       /* Client-window handle     */
               &szMsg,            /* Body of the message
               "Debugging Information",
                                  /* Title of the message     */
               0,                 /* Message Box id           */
               &mb2info);         /* Message Box input structure
```

Using a Dialog Window

When using a dialog window, an application must load the dialog window, process user input, and destroy the dialog window when the user finishes the task. The process for handling a dialog window varies, depending on whether the dialog window is modal or modeless.

Creating a Dialog Template

The following source-code fragment creates a dialog template. Notice that the WS_GROUP and WS_TABSTOP style designations are given for the first item in each group.

```
DLGTEMPLATE IDD_ABOUT
BEGIN
  DIALOG "", IDD_ABOUT2,
  10, 10, 150, 110, FS_DLGBORDER, 0
  BEGIN
    CONTROL "Attributes:",100,
      10, 30, 100, 70,
      WC_STATIC,
      SS_GROUPBOX | WS_VISIBLE
    CONTROL "Highlighted",101,
      20, 80, 58, 12,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Enabled",102,
      20, 60, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Checked",103,
      20, 40, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Okay", DID_OK,
      10, 10, 50, 14,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_PUSHBUTTON | BS_DEFAULT | WS_VISIBLE
    CONTROL "Cancel", DID_CANCEL,
      80, 10, 50, 14,
      WC_BUTTON,
      BS_PUSHBUTTON | WS_VISIBLE
  END
END
```

Creating a Modal Dialog Window

The easiest way to use a modal dialog window is to define a dialog template in the resource file (as in the preceding section), and then, call the WinDlgBox function, specifying the dialog-window resource identifier and a pointer to the dialog procedure. WinDlgBox loads the dialog-window resource, displays the dialog window, and handles all user input until the user dismisses the dialog window. The dialog procedure receives messages when the dialog window is created (WM_INITDLG) and other messages each time the user interacts with a dialog item (enters text in entry fields or selects a button, for example).

You must specify both the parent and owner windows when loading a dialog window using the WinDlgBox function. Generally, the parent window will be HWND_DESKTOP and the owner will be a client window in your application.

Dialog windows typically contain buttons that send WM_COMMAND messages when selected by the user. WM_COMMAND messages passed to the WinDefDlgProc function result in the WinDismissDlg function's being called, with the window identifier of the source button as the return code (from WinDismissDlg). Dialog windows with either **OK** or **Cancel** as their only button can ignore WM_COMMAND messages, allowing them to be passed to WinDefDlgProc. WinDefDlgProc calls WinDismissDlg to dismiss the dialog window and returns the DID_OK or DID_CANCEL code.

Passing WM_COMMAND messages to WinDefDlgProc means that all button presses in the dialog window dismiss the dialog window. If you want certain buttons to initiate operations without closing the dialog window, or if you want to perform some processing without closing the dialog window, handle the WM_COMMAND messages in the dialog procedure.

If you handle WM_COMMAND messages in the dialog procedure, you must call WinDismissDlg to dismiss the dialog window. Your dialog procedure passes the DID_OK code to WinDismissDlg if the user selects the **OK** button or the DID_CANCEL code if the user selects the **Cancel** button.

When you call WinDismissDlg or pass the WM_COMMAND message to WinDefDlgProc, the dialog window is dismissed, and the WinDlgBox function returns the value passed to WinDismissDlg. This return value identifies the button selected.

An alternative to using WinDlgBox is to call the individual functions that duplicate its functionality, as shown in the following code fragment:

```
HWND  hwndDlg;
ULONG ulResult;

hwndDlg = WinLoadDlg(...);
ulResult = WinProcessDlg(hwndDlg);
WinDestroyWindow(hwndDlg);
```

After calling the WinProcessDlg function, your dialog procedure must call WinDismissDlg to dismiss the dialog window. Although the dialog window is *dismissed* (hidden), it still exists. You must call the WinDestroyWindow function to destroy a dialog window if it was loaded using the WinLoadDlg function. WinDlgBox automatically destroys a dialog window before returning.

If you want to manipulate individual items in a dialog window, or add a menu after loading the dialog window (but before calling WinProcessDlg), it is better to make individual calls rather than call WinDlgBox. Individual calls also are useful for querying individual dialog items—to determine the contents of an entry-field control after a dialog window is closed but before it is destroyed, for example. Destroying a dialog window also destroys any dialog-item control windows that are child windows of the dialog window.

Creating a Modeless Dialog Window

To use a modeless dialog window in an application, create a dialog template in the resource file, just as for a modal dialog window. Modeless dialog windows share the screen equally with other frame windows. It is a good idea to give modeless dialog windows a title bar so they can be moved around the screen. The following Resource Compiler source-code fragment shows a dialog template for a dialog window with a title bar, system menu, and minimize button.

```
DLGTEMPLATE IDD_SAMP
BEGIN
    DIALOG "Modeless Dialog", IDD_SAMP, 80, 92, 126, 130,
        WS_VISIBLE | FS_DLGBORDER,
        FCF_TITLEBAR | FCF_SYSMENU | FCF_MINBUTTON

    BEGIN

    /* Put control-window definitions here. */

    END
END
```

The application loads the dialog resource from the resource file using the WinLoadDlg function, receiving in return a window handle to the dialog window. The application treats the dialog window as if it were an ordinary window. Messages for the dialog window are dispatched through the event loop the application uses for its other windows. In fact, an application can have a modeless dialog window as its only window.

The resource for a modeless dialog window is like the resource used for a modal dialog window. The difference between modal and modeless dialog windows is the way applications handle input to each. For a modal dialog, the WinDlgBox and WinProcessDlg functions handle all user input to the dialog window, preventing access to other windows in the application. For a modeless dialog window, the application does not call these functions, relying instead on a normal message loop to dispatch messages to the dialog procedure.

The primary difference between a modeless dialog window and a standard frame window with child control windows is that, for a modeless dialog window, an application can define child windows for the dialog window in a dialog template, automating the process of creating the window and its child windows. The same effect can be achieved by creating a standard frame window, but then, the child control windows must be created individually.

It is important that an application keep track of all open modeless dialog windows so that it can destroy all open windows before terminating.

Initializing a Dialog Window

Generally, an application defines a dialog template in its resource file and loads the dialog window by calling the WinLoadDlg function or the WinDlgBox function (which calls WinLoadDlg). The dialog window is created as an invisible window unless the window style WS_VISIBLE is specified in the dialog template. A WM_INITDLG message is sent to the dialog procedure before WinLoadDlg returns. As each control defined in the template is created, the dialog procedure might receive various control notifications before the function returns. WinLoadDlg returns a handle to the dialog window immediately after creating a dialog window.

In general, it is a good idea to define a dialog window as invisible, since this allows for optimization. For example, an experienced user might type ahead rapidly, anticipating the processing of a dialog-window command. In such a case, there is no need to display the dialog window, because the user has finished the interaction before the window can be displayed. This is how the WinProcessDlg function works—it does not display a dialog window while there still are WM_CHAR messages in the input queue; it lets these messages to be processed first.

As control windows in a dialog window are created from the template, strings in the template are processed by the WinSubstituteStrings function. Any WM_SUBSTITUTESTRING messages are sent to the dialog procedure before WinLoadDlg returns.

When child windows of a dialog window are created, WinSubstituteStrings is used so child windows can make substitutions in their window text. If any child-window text string contains the percent sign (%) substitution character, the length of the text string is limited to 256 characters after it is returned from the substitution.

Adding a Menu in a Dialog Window

To create a menu bar and menus in a dialog window, an application first must load the dialog window to get a handle to the dialog-frame window. The dialog-frame window can be associated with a menu resource by calling the WinLoadMenu function. This function requires arguments that specify the menu identifier and the handle of the parent window for the menu. Finally, the dialog-frame window must incorporate the menu by sending a WM_UPDATEFRAME message to the dialog window. The following code fragment illustrates these operations:

```
HWND hwndDialog, hwndMenu;

/* Get the dialog resource. */
hwndDialog = WinLoadDlg(...);

/ Get the menu resource and attach it to the dialog window. */
hwndMenu = WinLoadMenu(hwndDialog, ...);

/* Inform the dialog window that it has a new menu.        */
WinSendMsg(hwndDialog, WM_UPDATEFRAME, (MPARAM) NULL, (MPARAM) NULL);
```

Applications can create menus in both modal and modeless dialog windows. The preceding code fragment can be used for either type of dialog window. For a modal dialog window, your application must call the WinProcessDlg function to handle user input until the dialog window is dismissed. For a modeless dialog window, your application must call the WinShowWindow function to display the dialog window, enabling the message loop to direct messages to the dialog window.

Creating a Dialog Procedure

In contrast to window procedures, which receive WM_CREATE messages, dialog procedures receive WM_INITDLG messages, which are sent after a dialog window is created, but before it is displayed. WM_INITDLG can do the same type of initialization tasks that WM_CREATE handles, but is not the first message that is received.

For example, if a dialog window contains a list box, use WM_INITDLG to fill the list box with items. Also use this procedure to enable or disable buttons in a dialog window, depending on your application.

You also can call the WinSetDlgItemText or WinSetDlgItemShort functions during dialog initialization, to set up text items that reflect the current conditions in your application.

Another typical task for the WM_INITDLG message handler is centering a dialog window on the screen or within its owner window. The following code fragment illustrates how to center a dialog window on the screen using WM_INITDLG:

```
RECTL rclScreen,rclDialog;
LONG  sWidth,sHeight,sBLCx,sBLCy;

case WM_INITDLG:
    /* Center the dialog window and get the screen rectangle.   */
    WinQueryWindowRect(HWND_DESKTOP, &rclScreen);

    /* Get the dialog-window rectangle.                          */
    WinQueryWindowRect(hwnd, &rclDialog);

    /* Get the dialog-window width.                              */
    sWidth = (LONG) (rclDialog.xRight - rclDialog.xLeft);

    /* Get the dialog-window height.                             */
    sHeight = (LONG) (rclDialog.yTop - rclDialog.yBottom);

    /* Set the horizontal coordinate of the lower-left corner.  */
    sBLCx = ((LONG) rclScreen.xRight - sWidth) / 2;

    /* Set vertical coordinate of the lower-left corner.         */
    sBLCy = ((LONG) rclScreen.yTop - sHeight) / 2;

    /* Move, size, and show the window.                          */
    WinSetWindowPos(hwnd,
        HWND_TOP,
        sBLCx, sBLCy,
        0, 0,         /* Ignores size arguments                  */
        SWP_MOVE);

    return 0;
```

The dialog procedure receives notification messages from each control-window item in a dialog window whenever a user clicks an item or enters text in an entry field. Most dialog procedures wait for the user to select one or more dialog-window buttons to signal being finished with the dialog window. When the dialog procedure receives one of these messages, it calls the WinDismissDlg function, as shown in the following code fragment. The second argument to WinDismissDlg is the value returned by the WinDlgBox or WinProcessDlg functions. Generally, these functions return the identifier of the button that was pressed.

```
MRESULT EXPENTRY SampDialogProc(HWND hwnd,
                                ULONG ulMessage,
                                MPARAM mp1,
                                MPARAM mp2)
{
    switch (ulMessage) {
        case WM_COMMAND:
            switch (SHORT1FROMMP(mp1)) {
                case DID_OK:

                   /*
                    * Final dialog-item queries,
                    * dismiss the dialog.
                    */

                   WinDismissDlg(hwnd, DID_OK);
                   return 0;
            }
            break;
    }
    return (WinDefDlgProc(hwnd, ulMessage, mp1, mp2));
}
```

Other dialog-window items send notification messages specific to the type of control window. Your dialog procedure should respond to notification messages from any relevant or important dialog items, and pass the messages that your dialog procedure does not handle to the WinDefDlgProc function for default processing. The default dialog procedure is used similarly to the default frame-window procedure.

The WM_COMMAND message from the **OK** button indicates that the user has selected the **OK** button and is finished with the dialog window. If the dialog window has other controls, such as entry fields or check boxes, have your dialog procedure query the contents or state of each control upon receipt of a message from the **OK** button. Before dismissing a dialog window, have your dialog procedure collect input from each dialog-window control before closing the dialog window.

Manipulating Dialog Items

Dialog items are control windows and, as such, can be manipulated using standard window-management function calls. The window handle is obtained for each dialog item by calling the WinWindowFromID function and passing the window handle for the dialog window and the window identifier for the dialog item as defined in the dialog template. Include the following Resource Compiler source-code fragment in your dialog template:

```
DLGTEMPLATE IDD_ABOUT
BEGIN
    DIALOG "", IDD_ABOUT, 80, 92, 126, 130, FS_DLGBORDER, 0
    BEGIN
        PUSHBUTTON "My Button", ITEMID_MYBUTTON, 37, 107, 56, 12

        /* Other item definitions ... */

    END
END
```

Based on this code fragment, your application will receive the button-item handle by initiating the following call to WinWindowFromID:

```
    hwndItem = WinWindowFromID(hwndDialog, ITEMID_MYBUTTON);
```

Applications often change the contents, enabled state, or position of dialog items at run time. For example, in a dialog window that contains a list box of file names and an **Open** button, the **Open** button should be disabled until the user selects a file from the list. To do this, define the button as disabled in the dialog resource so that it is disabled when the dialog window first is displayed. At run time, the dialog procedure receives a notification message from the list box when the user selects a file. At that time, the dialog procedure should call the WinEnableWindow function to enable the **Open** button.

Applications also can change the text in static dialog items and buttons by calling the WinSetWindowText function and using the window handle of a particular dialog item.

Related Functions

This section covers the functions that are related to Dialog Windows.

WinAlarm

This function generates an audible alarm.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinAlarm  (HWND  hwndDeskTop, ULONG  flStyle)
```

Parameters

hwndDeskTop (HWND) – input
Desktop-window handle.

HWND_DESKTOP	The desktop window
Other	Specified desktop window.

flStyle (ULONG) – input
Alarm style. Different alarms are selected by use of these values:

WA_WARNING
WA_NOTE
WA_ERROR

Returns

rc (BOOL) – returns
Alarm-generated indicator.

TRUE	Alarm generated
FALSE	Alarm not generated.

WinCreateDlg

This function creates a dialog window.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

HWND WinCreateDlg  (HWND hwndParent, HWND hwndOwner,
                    PFNWP pfnDlgProc, PDLGTEMPLATE pdlgt,
                    PVOID pCreateParams)
```

Parameters

hwndParent (HWND) – input
Parent-window handle of the created dialog window.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

hwndOwner (HWND) – input
Requested owner-window handle of the created dialog window.

pfnDlgProc (PFNWP) – input
Dialog procedure for the created dialog window.

pdlgt (PDLGTEMPLATE) – input
Dialog template.

pCreateParams (PVOID) – input
Pointer to application-defined data area.

Returns

hwndDlg (HWND) – returns
Dialog-window handle.

NULLHANDLE	Dialog window not created
Other	Dialog-window handle.

WinDismissDlg

This function hides the modeless dialog window, or destroys the modal dialog window, and causes the WinProcessDlg or WinDlgBox functions to return.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinDismissDlg  (HWND  hwndDlg, ULONG  usResult)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

usResult (ULONG) – input
Reply value.

Returns

rc (BOOL) – returns
Dialog-dismissed indicator.

TRUE	Dialog successfully dismissed
FALSE	Dialog not successfully dismissed.

WinDlgBox

This function loads and processes a modal dialog window and returns the result value established by the WinDismissDlg call.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
ULONG WinDlgBox  (HWND hwndParent, HWND hwndOwner, PFNWP pfnDlgProc,
                  HMODULE hmod, ULONG idDlg, PVOID pCreateParams)
```

Parameters

hwndParent (HWND) – input
Parent-window handle of the created dialog window.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

hwndOwner (HWND) – input
Requested owner-window handle of the created dialog window.

pfnDlgProc (PFNWP) – input
Dialog procedure for the created dialog window.

hmod (HMODULE) – input
Resource identity containing the dialog template.

NULLHANDLE	Use the application's .EXE file.
Other	Module handle returned from the DosLoadModule or DosQueryModuleHandle call.

idDlg (ULONG) – input
Dialog-template identity within the resource file.

pCreateParams (PVOID) – input
Pointer to application-defined data area.

Returns

ulResult (ULONG) – returns
Reply value.

WinEnumDlgItem

This function returns the window handle of a dialog item within a dialog window.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinEnumDlgItem  (HWND hwndDlg, HWND hwnd, ULONG code)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

hwnd (HWND) – input
Child-window handle.

code (ULONG) – input
Item-type code.

EDI_PREVTABITEM	Previous item with style WS_TABSTOP. Wraps around to end of dialog item list when beginning is reached.
EDI_NEXTTABITEM	Next item with style WS_TABSTOP. Wraps around to beginning of dialog item list when end is reached.
EDI_FIRSTTABITEM	First item in dialog with style WS_TABSTOP. *hwnd* is ignored.
EDI_LASTTABITEM	Last item in dialog with style WS_TABSTOP. *hwnd* is ignored.
EDI_PREVGROUPITEM	Previous item in the same group. Wraps around to end of group when the start of the group is reached. For information on the WS_GROUP style, see "Window Styles" on page 2-14.
EDI_NEXTGROUPITEM	Next item in the same group. Wraps around to beginning of group when the end of the group is reached.
EDI_FIRSTGROUPITEM	First item in the same group.
EDI_LASTGROUPITEM	Last item in the same group.

Returns

hwndItem (HWND) – returns
Item-window handle.

WinGetDlgMsg

This function obtains a message from the application's queue associated with the specified dialog.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinGetDlgMsg  (HWND hwndDlg, PQMSG pqmsg)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

pqmsg (PQMSG) – output
Message structure.

Returns

rc (BOOL) – returns
Continue message indicator.

TRUE Message returned is not a WM_QUIT message and the dialog has not been dismissed.

FALSE Message returned is a WM_QUIT message or the dialog has been dismissed.

WinLoadDlg

This function creates a dialog window from the dialog template *idDlg* in *hmod* and returns the dialog window handle.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

HWND WinLoadDlg  (HWND hwndParent, HWND hwndOwner, PFNWP pfnDlgProc,
                  HMODULE hmod, ULONG idDlg, PVOID pCreateParams)
```

Parameters

hwndParent (HWND) – input

Parent-window handle of the created dialog window.

HWND_DESKTOP	The desktop window
HWND_OBJECT	Object window
Other	Specified window.

hwndOwner (HWND) – input

Requested owner-window handle of the created dialog window.

pfnDlgProc (PFNWP) – input

Dialog procedure for the created dialog window.

hmod (HMODULE) – input

Resource identity containing the dialog template.

NULLHANDLE	Use the application's .EXE file.
Other	Module handle returned from the DosLoadModule or DosQueryModuleHandle functions.

idDlg (ULONG) – input

Dialog-template identity within the resource file.

pCreateParams (PVOID) – input

Pointer to application-defined data area.

Returns

hwndDlg (HWND) – returns

Dialog-window handle.

NULL	Dialog window not created
Other	Dialog window handle.

WinMapDlgPoints

This function maps points from dialog coordinates to window coordinates, or from window coordinates to dialog coordinates.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinMapDlgPoints  (HWND hwndDlg, PPOINTL prgwptl, ULONG cwpt,
                       BOOL fCalcWindowCoords)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

prgwptl (PPOINTL) – in/out
Coordinate points to be mapped.

cwpt (ULONG) – input
Number of coordinate points.

fCalcWindowCoords (BOOL) – input
Calculation control.

TRUE The points are in dialog coordinates and are to be mapped into window coordinates relative to the window specified by the *hwndDlg* parameter.

FALSE The points are in window coordinates relative to the window specified by the *hwndDlg* parameter and are to be mapped into dialog coordinates.

Returns

rc (BOOL) – returns
Coordinates-mapped indicator.

TRUE Coordinates successfully mapped
FALSE Coordinates not successfully mapped.

WinMessageBox

This function creates, displays, and operates a message box window.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>

ULONG WinMessageBox  (HWND hwndParent, HWND hwndOwner, PSZ pszText,
                      PSZ pszCaption, ULONG idWindow, ULONG flStyle)
```

Parameters

hwndParent (HWND) – input
Parent-window handle of the created message-box window.

HWND_DESKTOP	The message box is to be main window.
Other	Parent-window handle.

hwndOwner (HWND) – input
Requested owner-window handle of the created message-box window.

pszText (PSZ) – input
Message-box window message.

pszCaption (PSZ) – input
Message-box window title.

NULL	The text Error is to be displayed as the title of the message-box window.
Other	The text to be displayed as the title of the message-box window.

idWindow (ULONG) – input
Message-box window identity.

flStyle (ULONG) – input
Message-box window style.

Returns

usResponse (ULONG) – returns
User-response value.

MBID_ENTER	ENTER push button was selected
MBID_OK	OK push button was selected
MBID_CANCEL	CANCEL push button was selected
MBID_ABORT	ABORT push button was selected
MBID_RETRY	RETRY push button was selected
MBID_IGNORE	IGNORE push button was selected
MBID_YES	YES push button was selected

MBID_NO	NO push button was selected
MBID_ERROR	Function not successful; an error occurred.

WinMessageBox2

This function creates a message-box window that can be used to display error messages and ask questions.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_GPI, INCL_WINWINDOWMGR, */
#include <os2.h>
ULONG WinMessageBox2  (HWND hwndParent, HWND hwndOwner, PSZ pszText,
                       PSZ pszTitle, ULONG ulWindow, PMB2INFO pmb2info)
```

Parameters

hwndParent (HWND) – input
Parent-window handle of the message-box window to be created.

HWND-DESKTOP	The message box is to be main window.
Other	Parent-window handle.

hwndOwner (HWND) – input
Requested owner-window handle of the message-box window to be created.

pszText (PSZ) – input
Message-box window message.

pszTitle (PSZ) – input
Message-box window title.

ulWindow (ULONG) – input
Message-box window identity.

pmb2info (PMB2INFO) – input
Input structure for mesage-box window.

Returns

ulButtonId (ULONG) – returns
Id of the button that was clicked, or MBID_ERROR.

WinProcessDlg

This function dispatches messages while a modal dialog window is displayed.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG WinProcessDlg  (HWND hwndDlg)
```

Parameters

hwndDlg (HWND) – input
Dialog-window handle.

Returns

ulReply (ULONG) – returns
Reply value.

WinQueryDlgItemShort

This function converts the text of a dialog item into an integer value.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
BOOL WinQueryDlgItemShort  (HWND hwndDlg, ULONG idItem,
                            PSHORT psResult, BOOL fSigned)
```

Parameters

hwndDlg (HWND) – input
Parent-window handle.

idItem (ULONG) – input
Identity of the child window whose text is to be converted.

psResult (PSHORT) – output
Integer value resulting from the conversion.

fSigned (BOOL) – input
Sign indicator.

TRUE	Signed text. It is inspected for a minus sign (–).
FALSE	Unsigned text.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful conversion
FALSE	Error occurred.

WinQueryDlgItemText

This function queries a text string in a dialog item.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */
#include <os2.h>
```

ULONG WinQueryDlgItemText (HWND hwndDlg, ULONG idItem, LONG lMaxText, PSZ pszText)

Parameters

hwndDlg (HWND) – input
Parent-window handle.

idItem (ULONG) – input
Identity of the child window whose text is to be queried.

lMaxText (LONG) – input
Length of *pszText*.

pszText (PSZ) – output
Output string.

Returns

ulRetLen (ULONG) – returns
Actual number of characters returned.

0	Error occurred
Other	Actual number of characters returned, not including the null-terminating character. The maximum value is (*lMaxText*–1).

WinQueryDlgItemTextLength

This function queries the length of the text string in a dialog item, not including any null termination character.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

LONG WinQueryDlgItemTextLength  (HWND hwndDlg, ULONG idItem)
```

Parameters

hwndDlg (HWND) – input
Parent-window handle.

idItem (ULONG) – input
Identity of the child window whose text is to be queried.

Returns

lRetLen (LONG) – returns
Length of text.

0	Error occurred
Other	Length of text.

WinSendDlgItemMsg

This function sends a message to the dialog item defined by *idItem* in the dialog window specified by *hwndDlg*.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

MRESULT WinSendDlgItemMsg  (HWND hwndDlg, ULONG idItem, ULONG msg,
                            MPARAM mp1, MPARAM mp2)
```

Parameters

hwndDlg (HWND) – input
Parent-window handle.

idItem (ULONG) – input
Identity of the child window.

msg (ULONG) – input
Message identity.

mp1 (MPARAM) – input
Message parameter 1.

mp2 (MPARAM) – input
Message parameter 2.

Returns

mresReply (MRESULT) – returns
Message-return data.

WinSetDlgItemShort

This function converts an integer value into the text of a dialog item.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, Also in COMON section */

#include <os2.h>

BOOL WinSetDlgItemShort  (HWND hwndDlg, ULONG idItem, USHORT usValue,
                          BOOL fSigned)
```

Parameters

hwndDlg (HWND) – input
Parent-window handle.

idItem (ULONG) – input
Identity of the child window whose text is to be changed.

usValue (USHORT) – input
Integer value used to generate the dialog item text.

fSigned (BOOL) – input
Sign indicator.

TRUE	Signed integer value
FALSE	Unsigned integer value.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetWindowText

This function sets the window text for a specified window.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetWindowText  (HWND  hwnd, PSZ  pszString)
```

Parameters

hwnd (HWND) – input
Window handle.

pszString (PSZ) – input
Window text.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Text updated
FALSE	Error occurred.

WinSubstituteStrings

This function performs a substitution process on a text string, replacing specific marker characters with text supplied by the application.

Syntax

```
#define INCL_WINDIALOGS /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinSubstituteStrings  (HWND hwnd, PSZ pszSrc, LONG lDestMax,
                            PSZ pszDest)
```

Parameters

hwnd (HWND) – input
Handle of window that processes the call.

pszSrc (PSZ) – input
Source string.

lDestMax (LONG) – input
Maximum number of characters returnable.

pszDest (PSZ) – output
Resultant string.

Returns

lDestRet (LONG) – returns
Actual number of characters returned.

Related Messages

This section covers the messages that are related to Dialog Windows.

WM_INITDLG

This message occurs when a dialog box is being created.

Parameters

param1

hwnd (HWND)

Focus window handle.

The handle of the control window that is to receive the input focus.

param2

pcreate (PVOID)

Application-defined data area.

This points to the data area and is passed by the WinLoadDlg, WinCreateDlg, and WinDlgBox functions in their *pCreateParams* parameter.

This parameter MUST be a pointer rather than a long.

The first 2 bytes in the data referenced by this pointer should be the total size of the data referenced by the pointer, (for example, see the ENTRYFDATA or the FRAMECDATA structure). PM requires this information to enable it to ensure that the referenced data is accessible to both 16-bit and 32-bit code.

Returns

rc (BOOL)

Focus set indicator.

TRUE	Focus window is changed. The dialog procedure can change the window to receive the focus, by issuing a WinSetFocus whose *hwndNewFocus* specifies the handle of another control within the dialog box.
FALSE	Focus window is not changed.

WM_MSGBOXDISMISS

This message notifies the owner of the message when a non-modal message box has been dismissed (the message box is no longer visible).

Parameters

param1

hwnd (HWND)
Non-modal window handle.

param2

ulButtonId (ULONG)
Identity of the selected button in the message box.

Returns

ulReserved (ULONG)
Reserved value, must be 0.

WM_MSGBOXINIT

This message notifies the owner of the message when a non-modal message box has been created and is currently being displayed.

Parameters

param1

hwnd (HWND)
Non-modal window handle.

param2

idWindow (LONG)
Window identity of the message box.

Returns

ulReserved (ULONG)
Reserved value, must be 0.

WM_SUBSTITUTESTRING

This message is sent from the WinSubstituteStrings call.

Parameters

param1

iindex (USHORT)
Substitution index.

A value corresponding to the decimal character in the substitution phrase.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

pString (PSZ)
String to be substituted.

0	No substitution string
Other	Substitution string.

Related Data Structures

This section covers the data structures that are related to Dialog Windows.

DLGTEMPLATE

Dialog-template structure.

Syntax

```
typedef struct _DLGTEMPLATE {
USHORT          cbTemplate;
USHORT          type;
USHORT          codepage;
USHORT          offadlgti;
USHORT          fsTemplateStatus;
USHORT          iItemFocus;
USHORT          coffPresParams;
DLGTITEM        adlgti[1];
 } DLGTEMPLATE;

typedef DLGTEMPLATE *PDLGTEMPLATE;
```

Fields

cbTemplate (USHORT)
Length of template.

type (USHORT)
Template format type.

codepage (USHORT)
Code page.

offadlgti (USHORT)
Offset to dialog items.

fsTemplateStatus (USHORT)
Template status.

iItemFocus (USHORT)
Index of item to receive focus initially.

coffPresParams (USHORT)
Count of presentation-parameter offsets.

adlgti[1] (DLGTITEM)
Start of dialog items.

DLGTITEM

Dialog-item structure.

Syntax

```
typedef struct _DLGTITEM {
USHORT       fsItemStatus;
USHORT       cChildren;
USHORT       cchClassLen;
USHORT       offClassName;
USHORT       cchTextLen;
USHORT       offText;
ULONG        flStyle;
SHORT        x;
SHORT        y;
SHORT        cx;
SHORT        cy;
USHORT       id;
USHORT       offPresParams;
USHORT       offCtlData;
 } DLGTITEM;

typedef DLGTITEM *PDLGTITEM;
```

Fields

fsItemStatus (USHORT)
Status.

cChildren (USHORT)
Count of children to this dialog item.

cchClassLen (USHORT)
Length of class name.

If zero, *offClassName* contains the hexadecimal equivalent of a preregistered class name.

offClassName (USHORT)
Offset to class name.

If *cchClassLen* is nonzero, this is the offset to a null-terminated ASCII string that contains the classname. If *cchClassLen* is zero, this is of the form 0xhhhh, where hhhh is the hexadecimal equivalent of the preregistered class name.

cchTextLen (USHORT)
Length of text.

offText (USHORT)
Offset to text.

flStyle (ULONG)
Dialog item window style.

The high-order 16 bits are the standard WS_* style bits. The low-order 16 bits are available for class-specific use.

x (SHORT)
X-coordinate of origin of dialog-item window.

y (SHORT)
Y-coordinate of origin of dialog-item window.

cx (SHORT)
Dialog-item window width.

cy (SHORT)
Dialog-item window height.

id (USHORT)
Identity.

offPresParams (USHORT)
Reserved.

offCtlData (USHORT)
Offset to control data.

MB2D

Array of button definitions.

Syntax

```
typedef struct _MB2D {
CHAR        achText[MAX_MB2DTEXT+1];
ULONG       idButtons;
ULONG       flStyle;
 } MB2D;

typedef MB2D *PMB2D;
```

Fields

achText[MAX_MB2DTEXT+1] (CHAR)

Text of the button.

For example, “Cancel.”

Currently, MAX_MB2DTEXT is equal to 70.

idButtons (ULONG)

Button Id returned when selected.

flStyle (ULONG)

Button style flags.

These style flags may be ORed with internal styles.

MB2INFO

Button information block.

Syntax

```
typedef struct _MB2INFO {
ULONG          cb;
HPOINTER       hIcon;
ULONG          cButtons;
ULONG          flStyle;
HWND           hwndNotify;
MB2D           mb2d[1];
 } MB2INFO;

typedef MB2INFO *PMB2INFO;
```

Fields

cb (ULONG)
Current size of the structure.

hIcon (HPOINTER)
Icon handle.

cButtons (ULONG)
Number of buttons.

flStyle (ULONG)
Icon style flags.

Possible values are described in the following list:

MB_APPLMODAL	Message box is application modal. This is the default case. Its owner is disabled; therefore, do not specify the owner as the parent if this option is used.
MB_ERROR	Message box contains a stop sign with a white background.
MB_ICONASTERISK	Message box contains a asterisk icon.
MB_CUSTOMICON	Message box contains a custom icon specified in *hIcon.*
MB_ICONEXCLAMATION	Message box contains a exclamation point icon.
MB_ICONHAND	Message box contains a hand icon.
MB_ICONQUERY	Message box contains a question mark in a box.
MB_ICONQUESTION	Message box contains a question mark icon.
MB_INFORMATION	Message box contains a black "i" in a box.

MB_MOVEABLE
Message box is moveable.

The message box is displayed with a title bar and a system menu, showing only the Move, Close, and Task Manager choices, which can be selected either by use of the pointing device or by accelerator keys.

MB_NOICON
Message box does not contain an icon.

MB_NONMODAL
Message box is nonmodal (the program continues after displaying the nonmodal message box).

The message box remains visible until the owner window destroys it. Two notification messages, WM_MSGBOXINIT and WM_MSGBOXDISMISS, are used to support this non-modality.

MB_SYSTEMMODAL
Message box is system modal.

MB_WARNING
Message box contains a black "!" in a box.

hwndNotify (HWND)
Owner notification handle.

mb2d[1] (MB2D)
Array of button definitions.

Summary

Following are the OS/2 functions, messages, and structures used with dialog windows.

Table 15-2. Dialog Functions

Function name	Description
WinAlarm	Generates an audible alarm.
WinCreateDlg	Creates a dialog window.
WinDefDlgProc	Invokes the default dialog procedure.
WinDestroyWindow	Destroys a window and its child windows.
WinDismissDlg	Hides the modeless dialog window, or destroys the modal dialog window, and causes the WinProcessDlg or WinDlgBox calls to return.
WinDlgBox	Loads and processes a modal dialog window and returns the result value established by the WinDismissDlg call.
WinEnumDlgItem	Returns the window handle of a dialog item within a dialog window.
WinGetDlgMsg	Obtains a message from the application's queue associated with the specified dialog.
WinLoadDlg	Creates a dialog window from the dialog template *Dlgid* in *Resource*.
WinMapDlgPoints	Maps points from dialog coordinates to window coordinates or from window coordinates to dialog coordinates.
WinMessageBox	Creates, displays, and operates a standard message box.
WinMessageBox2	Creates, displays, and operates an enhanced message box. This control supports customized icons and text within buttons and can be non-modal.
WinProcessDlg	Dispatches messages while a modal dialog window is displayed.
WinQueryDlgItemShort	Converts the text of a dialog item into an integer value.
WinQueryDlgItemText	Queries a text string in a dialog item.
WinQueryDlgItemTextLength	Queries the length of the text string in a dialog item.
WinSendDlgItemMsg	Sends a message to the dialog item defined by *item* in the dialog window specified by *Dlg*.
WinSetDlgItemShort	Converts an integer value into the text of a dialog item.
WinSetDlgItemText	Sets a text string in a dialog item.
WinSetWindowText	Sets the window text for a specified window.
WinSubstituteStrings	Performs a substitution process on a text string, replacing specific marker characters with text supplied by the application.

Table 15-3. Dialog Messages

Message	Description
WM_CHAR	Sent when a user presses a key.
WM_INITDLG	Occurs when a dialog box is being created.
WM_MSGBOXDISMISS	Notifies the owner of the message when a non-modal message box has been dismissed.
WM_MSGBOXINIT	Notifies the owner of the message when a non-modal message box has been created and is currently being displayed.
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message may be processed by the dialog manager or passed down to the control.
WM_SUBSTITUTESTRING	Sent from the WinSubstituteStrings call.

Table 15-4. Dialog Structures

Structure name	Description
DLGTEMPLATE	Dialog-template structure.
DLGTITEM	Dialog-item structure.
MB2D	Array of button definitions.
MB2INFO	Button information block.

Chapter 16. Control Windows

A *control window* is a window that an application uses in conjunction with another window to carry out simple input and output tasks. This chapter describes how to create and use control windows in PM applications.

About Control Windows

Control windows are used most often as part of a frame or dialog window, but they also can be used in a client window. An application can create control windows in a frame window by using frame-control flags in the WinCreateStdWindow function, or it can create control windows individually by calling the WinCreateWindow function.

Including control windows in a dialog window requires the use of a *dialog template*, which is a data structure that describes a dialog window and its control windows. The system uses the data in the dialog template to create the dialog window and control windows. An application can create a dialog template at run time, or it can use the system resource compiler to create a dialog-template resource.

The operating system provides many types of predefined control windows. An application can create a control of a particular type by specifying the appropriate control-window class name, either in the WinCreateWindow function or in a dialog template. The following is a list of the predefined control-window classes:

Table 16-1 (Page 1 of 2). Control Window Classes

Class name	Description
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_COMBOBOX	Creates a combination-box control, which combines a list-box control and an entry-field control. It allows the user to enter data by typing in the entry field or choosing from a list in the list box.
WC_CONTAINER	Creates a control for the user to group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A composite window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as action bars, or vertically as pull-down menus. Menus usually are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.

Table 16-1 (Page 2 of 2). Control Window Classes

Class name	Description
WC_SCROLLBAR	Consists of window scroll bars that let the user request to scroll the contents of an associated window.
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to the radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

A control window is always owned by another window, usually a frame or dialog window. This relationship is important because a control window sends WM_CONTROL messages to its owner whenever an input event occurs in the control window. Each WM_CONTROL message includes the identifier of the control window in which the event occurred and a notification code that specifies the nature of the event. An application specifies a control window's ID either in the WinCreateWindow function or in a dialog template. Each ID must be unique.

Control windows are like other predefined window classes in that they respond to standard window-management messages and functions, such as WinSetWindowText and WinShowWindow.

All control-window classes have a set of specific messages they send and receive. The summary at the end of this chapter lists the messages that all control windows have in common.

The system paints most control windows synchronously—that is, it redraws a control window as soon as any part of that window becomes invalid.

Using Control Windows

An application can use control windows in a dialog window, standard frame window, or client window. The following sections describe how to use control windows in an application.

Using Control Windows in a Dialog Window

To use a control window in a dialog window, an application specifies the control in a dialog template in the application's resource-definition file. A dialog template typically includes several control windows. When the application loads the dialog-template resource and displays the dialog window, the system automatically displays the control windows as part of the dialog window.

An application can send messages, through the dialog-window procedure, to a control window to change its state. The control window sends notification messages to the dialog-window procedure. The content of a notification message depends on the type of control window.

Using Control Windows in a Non-Dialog Window

To use a control window in a non-dialog window, an application must call the WinCreateWindow function, using the appropriate window class name. An application usually specifies one of its client windows as the owner of the control window. Therefore, the client-window procedure receives notification messages from the control window. In cases where a control is owned by the frame window (such as a menu control), the notification messages to the frame window are passed to the client window.

Creating a Custom Control Window

The operating system provides the following three ways to create custom control windows:

- Use ownerdraw list boxes and menus or buttons.
- Subclass an existing control-window class.
- Register and implement a window class from scratch.

List boxes and menus can have an *ownerdraw* style, and buttons can have a user-button style, which cause the system to send a message to the owner of the ownerdraw control whenever the control must be drawn. (If the owner is a frame window, it sends these messages on to its client windows for handling by the client window procedure.) This feature lets an application alter the appearance of a control window. For menus and list boxes, the owner window draws the items within the control, and the system draws the outline of the control. For buttons, the user-button style affects the drawing of the entire control. Subclassing an existing control window is an easy way to create a custom control. The subclass procedure can alter selected behavior of the control window by processing only those messages that affect the selected behaviors. All other messages pass to the original control-window procedure.

The techniques for defining a custom control-window class are the same as those used for creating a client-window class. When you create a custom control-window class, be sure the window procedure can send and receive the messages listed in Table 16-2 on page 16-7 and Table 16-3 on page 16-7.

If an application creates a private control-window class, the name of the private class could be used in the dialog template, just like a predefined window-class constant. For example, if an application defines and registers a window class called “MyControlClass”, it could create a dialog window that contains that type of control window by using the following resource definition:

```
DLGTEMPLATE IDD_CUSTOM_TEST
BEGIN
DIALOG "", IDD_CUSTOM_TEST, 1, 1, 126, 130, FS_DLGBORDER, 0
BEGIN
    CONTROL "This is Text", IDD_TITLE,
          37, 107, 56, 12,
          WC_STATIC,
          SS_TEXT | DT_CENTER | DT_TOP | DT_WORDBREAK
          | WS_VISIBLE
    CONTROL "Custom Control", IDD_CUSTOM,
          33, 68, 64, 13,
          "MyControlClass",
          WS_VISIBLE
    CONTROL "Okay", DID_OK,
          57, 10, 24, 14,
          WC_BUTTON,
          BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
    END
END
```

Related Messages

This section covers the messages that are related to Control Windows.

WM_CONTROL

This message occurs when a control has a significant event to notify to its owner.

Parameters

param1

id (USHORT)
Control-window identity.

This is either the *id* parameter of the WinCreateWindow function or the identity of an item in a dialog template.

usnotifycode (USHORT)
Notify code.

The meaning of the notify code depends on the type of the control. For details, refer to the section describing that control.

param2

ulcontrolspec (ULONG)
Control-specific information.

The meaning of the control-specific information depends on the type of the control. For details, refer to the section describing that control.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_QUERYDLGCODE

This message is sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and also to determine whether an input message may be processed by the dialog manager or passed down to the control.

Parameters

param1

pQmsg (PQMSG)
Message queue structure.

This points to a QMSG structure.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulDialogCode (ULONG)
Dialog code information flags.

DLGC_ENTRYFIELD	Identifies an entry field control. Assumed to understand the EM_SETSEL message.
DLGC_BUTTON	Identifies a button item. Assumed to understand the BM_CLICK message.
DLGC_RADIOBUTTON	Identifies a radio button control. Used with the DLGC_BUTTON code.
DLGC_STATIC	Identifies a static control. Static controls are not included in arrow key enumeration.
DLGC_DEFAULT	Identifies a default push-button control.
DLGC_PUSHBUTTON	Identifies a nondefault push button.
DLGC_CHECKBOX	Identifies a check-box item. Used with the DLGC_BUTTON code.
DLGC_SCROLLBAR	Identifies a scroll bar control.
DLGC_MENU	Identifies a menu control.
DLGC_TABONCLICK	Used by static controls to indicate that a mouse click on this control will cause focus to be placed on the next control in the dialog that has the WP_TABSTOP style. This should be useed in combination with the DLGC_STATIC code.
DLGC_MLE	Identifies a multiline entry field control.

Summary

Following are the OS/2 messages used with control windows.

Table 16-2. Messages Received by a Control Window

Message	Description
WM_ADJUSTWINDOWPOS	Sent by WinSetWindowPos to enable the window to adjust its new position or size when it is about to be moved.
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message may be processed by the dialog manager or passed down to the control.

Table 16-3. Messages Generated by a Control Window

Message	Description
WM_COMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table.
WM_CONTROL	This message occurs when a control has a significant event to report to its owner.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointing device pointer moves over the control window, allowing the owner to set the pointer.
WM_HELP	Notifies the owner that an event, usually a keystroke, has been translated by an accelerator table into a WM_HELP message.
WM_SYSCOMMAND	Notifies the owner that an event, usually a keystroke, has been translated by an accelerator table into a WM_SYSCOMMAND message.

Chapter 17. Title-Bar Controls

A *title bar* is one of several control windows that comprise a standard frame window, giving the frame window its distinctive look and performance capabilities. This chapter describes how to create and use title-bar control windows in PM applications.

About Title Bars

The title bar in a standard frame window performs the following four functions:

- Displays the title of the window across the top of the frame window.
- Changes its highlighted appearance to show whether the frame window is active. (Ordinarily, the topmost window on the screen is the active window.)
- Responds to the actions of the user—for example, dragging the frame window to a new location on the screen.
- Flashes (as a result of the WinFlashWindow function) to get the attention of the user.

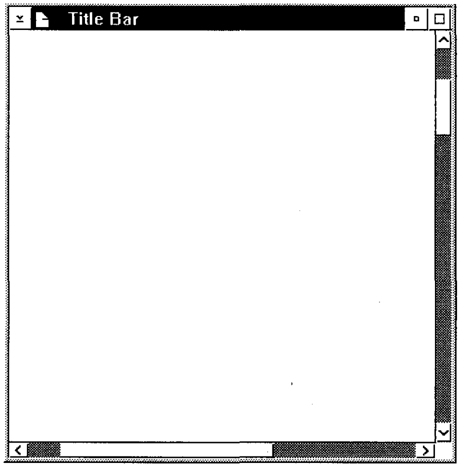


Figure 17-1. Title Bar in a Standard Frame Window

Once the frame controls are in place in the frame window, an application typically ignores them, because the system handles frame controls. In some cases, however, an application can take control of the title bar by sending messages to the title-bar control window.

Default Title-Bar Behavior

A title-bar control window sends messages to its owner (the frame window) when the control receives user input. Following are the messages that the title-bar control processes. Each message is described in terms of how the title-bar control responds to that message.

Table 17-1. Messages Processed by Title-Bar Control

Message	Description
TBM_QUERYHILITE	Returns the highlighted state of the title bar.
TBM_SETHILITE	Sets the highlighted state of the title bar, repainting the title bar if the state is changing.
WM_BUTTON1DBLCLK	Restores the title bar if the owner window is minimized or maximized. If the window is neither minimized nor maximized, this message maximizes the window.
WM_BUTTON1DOWN	Sends the WM_TRACKFRAME message to the owner window to start the tracking operation for the frame window.
WM_CREATE	Sets the text for the title bar. Returns FALSE if the text is already set.
WM_DESTROY	Frees the window text for the title bar.
WM_HITTEST	Always returns HT_NORMAL, so that the title bar does not beep when it is disabled. (It is disabled when the frame window is maximized.)
WM_PAINT	Draws the title bar.
WM_QUERYDLGCODE	Returns the predefined constant DLGC_STATIC. The user cannot use the Tab key to move to the title bar in a dialog window.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETWINDOWPARAMS	Sets the specified windcw parameters.
WM_WINDOWPOSCHANGED	Returns FALSE. Processes this message to prevent the WinDefWindowProc function from sending the size and show messages.

Using Title-Bar Controls

This section explains how to:

- Include a title bar in a frame window.
- Alter the dragging action of a title bar.

Including a Title Bar in a Frame Window

An application can include a title bar in a standard frame window by specifying the FCF_TITLEBAR flag in the WinCreateStdWindow function.

The following code fragment shows how to create a standard frame window with a title bar, minimize and maximize (window-sizing) buttons, size border, system menu, and an application menu.

```
#define ID_MENU_RESOURCE 101

HWND hwndFrame,hwndClient;
UCHAR szClassName[255];

ULONG flControlStyle = FCF_TITLEBAR | FCF_MINMAX | FCF_SIZEBORDER |
                       FCF_SYSMENU  | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE | FS_ACCELTABLE,
                               &flControlStyle, szClassName, "",
                               0, (HMODULE) NULL, ID_MENU_RESOURCE,
                               &hwndClient);
```

To get the window handle of a title-bar control, an application calls WinWindowFromID, specifying the frame-window handle and a constant identifying the title-bar control, as shown in the following code fragment:

```
hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

To set the text of a title bar, an application can use the WinSetWindowText function. The frame window passes the new text to the title-bar control in a WM_SETWINDOWPARAMS message.

Altering Dragging Action

When the user clicks the title bar, the title-bar control sends a WM_TRACKFRAME message to its owner (the frame window). When the frame window receives the WM_TRACKFRAME message, the frame sends a WM_QUERYTRACKINFO message to itself to fill in a TRACKINFO structure that defines the tracking parameters and boundaries. To modify the default behavior, an application must subclass the frame window, intercept the WM_QUERYTRACKINFO message, and modify the TRACKINFO structure. If the application returns TRUE for the WM_QUERYTRACKINFO message, the tracking operation proceeds according to the information in the TRACKINFO structure. If the application returns FALSE, no tracking occurs.

Related Functions

This section covers the functions that are related to Title Bar Controls.

WinFlashWindow

This function starts or stops a window flashing.

Syntax

```
#define INCL_WINFRAMEMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinFlashWindow  (HWND  hwndFrame, BOOL  fFlash)
```

Parameters

hwndFrame (HWND) – input
Handle of window to be flashed.

fFlash (BOOL) – input
Start-flashing indicator.

TRUE	Start window flashing
FALSE	Stop window flashing.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to Title Bar Controls.

TBM_QUERYHILITE

This message returns the highlighting state of a title-bar control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Highlighting state.

TRUE	Title-bar control is highlighted
FALSE	Title-bar control is not highlighted.

TBM_SETHILITE

This message is used to highlight or unhighlight a title-bar control.

Parameters

param1

usHighlighted (USHORT)
Highlighting indicator.

TRUE	Highlight the title-bar control
FALSE	Remove highlight from the title-bar control.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Data Structures

This section covers the data structures that are related to Title Bar Controls.

SWP

Set-window-position structure.

Syntax

```
typedef struct _SWP {
ULONG       fl;
LONG        cy;
LONG        cx;
LONG        y;
LONG        x;
HWND        hwndInsertBehind;
HWND        hwnd;
ULONG       ulReserved1;
ULONG       ulReserved2;
 } SWP;

typedef SWP *PSWP;
```

Fields

fl (ULONG)

Options.

In alphabetic order:

SWP_ACTIVATE
SWP_DEACTIVATE
SWP_HIDE
SWP_MAXIMIZE
SWP_MINIMIZE
SWP_MOVE
SWP_NOADJUST
SWP_NOERASEWINDOW
SWP_NOREDRAW
SWP_RESTORE
SWP_SHOW
SWP_SIZE
SWP_ZORDER

cy (LONG)

Window height.

cx (LONG)

Window width.

y (LONG)
Y-coordinate of origin.

x (LONG)
X-coordinate of origin.

hwndInsertBehind (HWND)
Window behind which this window is placed.

hwnd (HWND)
Window handle.

ulReserved1 (ULONG)
Reserved value, must be 0.

ulReserved2 (ULONG)
Reserved value, must be 0.

TRACKINFO

Tracking-information structure.

Syntax

```
typedef struct _TRACKINFO {
LONG         cxBorder;
LONG         cyBorder;
LONG         cxGrid;
LONG         cyGrid;
LONG         cxKeyboard;
LONG         cyKeyboard;
RECTL        rclTrack;
RECTL        rclBoundary;
POINTL       ptlMinTrackSize;
POINTL       ptlMaxTrackSize;
ULONG        fs;
 } TRACKINFO;

typedef TRACKINFO *PTRACKINFO;
```

Fields

cxBorder (LONG)
Border width.

The width of the left and right tracking sides.

cyBorder (LONG)
Border height.

The height of the top and bottom tracking sides.

cxGrid (LONG)
Grid width.

The horizontal bounds of the tracking movements.

cyGrid (LONG)
Grid height.

The vertical bounds of the tracking movements.

cxKeyboard (LONG)
Character cell width movement for arrow key.

cyKeyboard (LONG)
Character cell height movement for arrow key.

rclTrack (RECTL)
Starting tracking rectangle.

This is modified as the rectangle is tracked and holds the new tracking position, when tracking is complete.

rclBoundary (RECTL)
Boundary rectangle.

This is an absolute bounding rectangle that the tracking rectangle cannot extend; see also TF_ALLINBOUNDARY.

ptlMinTrackSize (POINTL)
Minimum tracking size.

ptlMaxTrackSize (POINTL)
Maximum tracking size.

fs (ULONG)
Tracking options.

In alphabetic order:

TF_ALLINBOUNDARY
The default tracking is such that some part of the tracking rectangle is within the bounding rectangle defined by *rclBoundary*. This minimum size is defined by *cxBorder* and *cyBorder*.

If TF_ALLINBOUNDARY is specified, the tracking is performed so that no part of the tracking rectangle ever falls outside of the bounding rectangle.

TF_BOTTOM
Track the bottom side of the rectangle.

TF_GRID
Tracking is restricted to the grid defined by *cxGrid* and *cyGrid*.

TF_LEFT
Track the left side of the rectangle.

TF_MOVE
Track all sides of the rectangle.

TF_RIGHT
Track the right side of the rectangle.

TF_SETPOINTERPOS
The pointer is repositioned according to other flags as follows:

none	Pointer is centered in the tracking rectangle.
TF_MOVE	Pointer is centered in the tracking rectangle.
TF_LEFT	Pointer is vertically centered at the left of the tracking rectangle.
TF_TOP	Pointer is horizontally centered at the top of the tracking rectangle.
TF_RIGHT	Pointer is vertically centered at the right of the tracking rectangle.
TF_BOTTOM	Pointer is horizontally centered at the bottom of the tracking rectangle.

TF_STANDARD
cx, *cy*, *cxGrid*, and *cyGrid* are all multiples of *cxBorder* and *cyBorder*.

TF_TOP
Track the top side of the rectangle.

Summary

Following are the OS/2 functions, messages, and structures used with title-bar controls.

Table 17-2. Title-Bar Functions

Function name	Description
WinCreateStdWindow	Creates a standard window.
WinFlashWindow	Starts or stops the flashing of a window.
WinSetWindowText	Sets the window text for a specified window.
WinWindowFromID	Returns the handle of the child window with the specified identity.

Table 17-3. Title-Bar Messages

Message	Description
TBM_QUERYHILITE	Returns the highlighting state of a title-bar control.
TBM_SETHILITE	Used to highlight or unhighlight a title-bar control.
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_CREATE	Occurs when an application requests the creation of a window.
WM_DESTROY	Occurs when an application requests the destruction of a window.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message can be processed by the dialog manager or passed down to the control.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the title-bar control window procedure window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the title-bar control window procedure window parameters.
WM_TRACKFRAME	Sent to a window whenever it is to be moved or sized.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Table 17-4. Title-Bar Structures

Structure name	Description
SWP	Set window position structure.
TRACKINFO	Tracking information structure.

Chapter 18. Scroll-Bar Controls

Scroll bars are control windows that convert mouse and keyboard input into integers; they are used by an application to scroll the contents of a client window. This chapter describes how to create and use scroll bars in PM applications.

About Scroll Bars

A scroll bar has three main parts: the bar, its arrows, and a slider (see Figure 18-1).

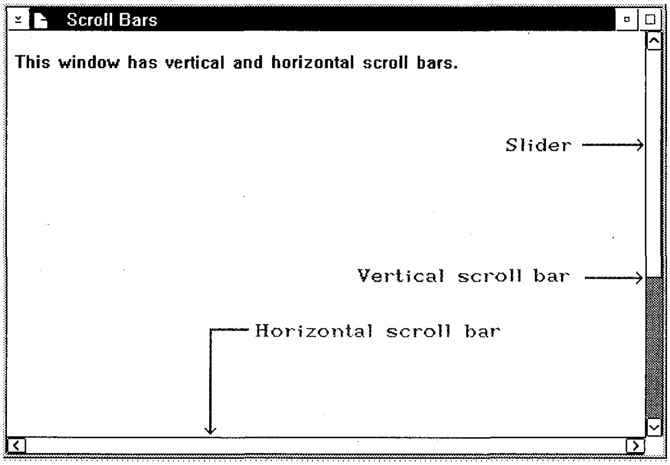


Figure 18-1. Scroll Bars in a Window

The arrows are located at each end of the scroll bar. The left scroll arrow, on the left side of a horizontal scroll bar, enables the user to scroll to the left in a document. The right scroll arrow lets the user scroll to the right.

On a vertical scroll bar, the upper scroll arrow enables the user to scroll upward in the document; the lower scroll arrow, downward. The slider, which lies between the two scroll arrows, reflects the current value of the scroll bar. Scroll bars monitor the slider and send notification messages to the owner window when the slider position changes as a result of mouse or keyboard input.

Although, typically, scroll bars are used in frame windows, an application can use stand-alone scroll bars of any size or shape, at any position, in a window of almost any class. Scroll bars can be used as parts of other control windows; for example, a list box uses a scroll bar to enable the user to view items when the list box is too small to display all the items.

Scroll-Bar Creation

An application can include a scroll bar in a standard frame window by specifying the FCF_HORZSCROLL or FCF_VERTSCROLL flag in the WinCreateStdWindow function. To create a scroll bar in another type of window, an application can specify the predefined (preregistered) window class WC_SCROLLBAR in the WinCreateWindow function or in the CONTROL statement in a resource file.

Although most applications specify an owner window when creating a scroll bar, an owner is not required. If an application does not specify an owner, the scroll bar does not send notification messages.

Scroll-Bar Styles

A scroll bar has styles that determine what it looks like and how it responds to input. Styles are specified in the WinCreateWindow function or the CONTROL statement. A scroll-bar can have the following styles:

Table 18-1. Scroll-Bar Styles

Style	Meaning
SBS_AUTOTRACK	Causes the entire slider to track the movement of the mouse pointer when the user scrolls the window. Without this style, only an outlined image of the slider tracks the movement of the mouse pointer, and the slider jumps to the new location when the user releases the mouse button.
SBS_HORZ	Creates a horizontal scroll bar.
SBS_THUMBSIZE	Used to calculate the size of the scroll-bar slider from the SBCDATA passed to WinCreateWindow.
SBS_VERT	Creates a vertical scroll bar.

Scroll-Bar Range and Position

Every scroll bar has a range and a slider position. The range specifies the minimum and maximum values for the slider position. As the user moves the slider in a scroll bar, the scroll bar reports the slider position as an integer in this range. If the slider position is the minimum value, the slider is at the top of a vertical scroll bar or at the left end of a horizontal scroll bar. If the slider position is the maximum value, the slider is at the bottom or right end of the vertical or horizontal scroll bar, respectively.

An application can adjust the range to convenient integers by using SBM_SETSCROLLBAR or WM_SETWINDOWPARAMS, or by using the SBCDATA structure during creation of the scroll bar. This enables you to easily translate the slider position into a value that corresponds to the data being scrolled. For example, an application attempting to display 100 lines of text (numbered 0 to 99) in a window that can show only 20 lines at a time could set the vertical scroll-bar range from 0–99, permitting any line to be the top line, and requiring blank lines to fill the viewing area when there are not sufficient lines of information to fill the area (lines 80–99). More likely, the range would be set to 0–79, so that only the first 80 lines could be the top line; this guarantees that there would always be 20 lines of text to fill the window.

The current settings can be obtained using SBM_QUERYRANGE or WM_QUERYWINDOWPARAMS.

To establish a useful relationship between the scroll-bar range and the data, an application must adjust the range whenever the data or the size of the window changes. This means the application should adjust the range as part of processing WM_SIZE messages.

An application must move the slider in a scroll bar. Although the user requests scrolling in a scroll bar, the scroll bar does not update the slider position. Instead, it passes the request to the owner window, which scrolls the data and updates the slider position using the SBM_SETPOS message. The application controls the slider movement and can move the slider in the increments best suited for the data being scrolled.

An application can retrieve the current slider position of a scroll bar by sending the SBM_QUERYPOS message to the scroll bar.

If a scroll bar is a descendant of a frame window, its position relative to its parent can change when the position of the frame window changes. Frame windows draw scroll bars relative to the upper-left corner of the frame window (rather than the lower-left corner). The frame window can adjust the *y* coordinate of the scroll-bar position, which would be desirable if the scroll bar is a child of the frame window, but would be undesirable if the scroll bar is not a child window.

Scroll-Bar Slider Size

The slider can be displayed either as a square (the default), or as a portion of the scroll bar if SBCDATA and the SBS_THUMBSIZE style are specified at creation. Displaying the slider as a proportional rectangle permits the size of the slider to be proportional to the amount of data being viewed in the visible range. The size is set based on the visible range and the number of values in the range. As an example, where the viewing area is 20 items and the range is 100, the slider size would be 20% of the potential slider area. Note that there is no direct connection between the scroll bar range and the range value used to set the slider size. It is possible to set the scroll-bar range from 0–99, and base the slider size on a viewing area of 500 and a range of 1000. This will set the scroll-bar to have 100 positions and will display a slider that is half the size of the scroll bar.

The slider size can be set using SBM_SETTHUMBSIZE or WM_SETWINDOWPARAMS, and obtained using WM_QUERYWINDOWPARAMS.

Scroll-Bar Notification Messages

A scroll bar sends notification messages to its owner whenever the user clicks the scroll bar. WM_VSCROLL and WM_HSCROLL are the notification messages for vertical and horizontal scroll bars, respectively. If the scroll bar is a frame control window, the frame window passes the message to its client window.

Each notification message includes the scroll-bar identifier, scroll-bar command code corresponding to the action of the user, and, in some cases, the position of the slider. If an application creates a scroll bar as part of a frame control window, the scroll-bar identifier is the predefined constant FID_VERTSCROLL or FID_HORZSCROLL. Otherwise, it is the identifier given in the WinCreateWindow function.

The scroll-bar command codes specify the action the user has taken. Operating system user-interface guidelines recommend certain responses for each action. Figure 18-2 illustrates the SBM_*xxx* messages your application can send to a scroll bar.

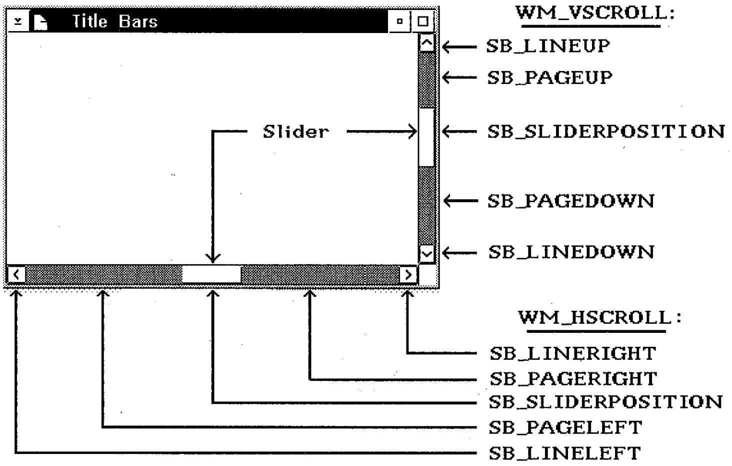


Figure 18-2. Standard Window Scroll Bar and Command Codes

Following is a list of the command codes; for each code, the user action is specified, followed by the application's response. In each case, a scrolling unit, appropriate for the given data, must be defined by the application. For example, for scrolling text vertically, the typical unit is a line.

Table 18-2. Scroll-Bar Command Codes

Command Code	Description
SB_LINEUP	Indicates that the user clicked the top scroll arrow. Decrement the slider position by one, and scroll toward the top of the data by one unit.
SB_LINEDOWN	Indicates that the user clicked the bottom scroll arrow. Increment the slider position by one, and scroll toward the bottom of the data by one unit.
SB_LINELEFT	Indicates that the user clicked the left scroll arrow. Decrement the slider position by one, and scroll toward the left end of the data by one unit.
SB_LINERIGHT	Indicates that the user clicked the right scroll arrow. Increment the slider position by one, and scroll toward the right end of the data by one unit.
SB_PAGEUP	Indicates that the user clicked the scroll-bar background above the slider. Decrement the slider position by the number of data units in the window, and scroll toward the top of the data by the same number of units.
SB_PAGEDOWN	Indicates that the user clicked the scroll-bar background below the slider. Increment the slider position by the number of data units in the window, and scroll toward the bottom of the data by the same number of units.
SB_PAGELEFT	Indicates that the user clicked the scroll-bar background to the left of the slider. Decrement the slider position by the number of data units in the window, and scroll toward the left end of the data by the same number of units.
SB_PAGERIGHT	Indicates that the user clicked the scroll-bar background to the right of the slider. Increment the slider position by the number of data units in the window, and scroll toward the right end of the data by the same number of units.
SB_SLIDERTRACK	Indicates that the user is dragging the slider. Applications that draw data quickly can set the slider to the position given in the message, and scroll the data by the same number of units the slider has moved. Applications that cannot draw data quickly should wait for the SB_SLIDERPOSITION code before moving the slider and scrolling the data.
SB_SLIDERPOSITION	Indicates that the user released the slider after dragging it. Set the slider to the position given in the message, and scroll the data by the same number of units the slider was moved.
SB_ENDSCROLL	Indicates that the user released the mouse after holding it on an arrow or in the scroll-bar background. No response is necessary.

If the command code is SB_SLIDERTRACK or SB_SLIDERPOSITION, indicating that the user is moving the scroll-bar slider, the notification message also contains the current position of the slider.

The owner window can send a message to the scroll bar to read or reset the current value and range of the scroll bar. To reflect any changes in the state of the scroll bar, the owner window also can adjust the data the scroll bar controls.

An application can use the WinEnableWindow function to disable a scroll bar. A disabled scroll bar ignores the actions of the user, sending out no notification messages when the user tries to manipulate it. If an application has no data to scroll, or if all data fits in the client window, the application should disable the scroll bar.

Scroll Bars and the Keyboard

When a scroll bar has the keyboard focus, it generates notification messages for the following keys:

Table 18-3. Scroll-bar Notification Messages

Keys	**Response**
UP	SB_LINEUP or SB_LINELEFT
LEFT	SB_LINEUP or SB_LINELEFT
DOWN	SB_LINEDOWN or SB_LINERIGHT
RIGHT	SB_LINEDOWN or SB_LINERIGHT
PGUP	SB_PAGEUP or SB_PAGELEFT
PGDN	SB_PAGEDOWN or SB_PAGERIGHT

If an application uses scroll bars to scroll data but does not give the scroll bar the input focus, the window with the focus must process keyboard input. The window can generate scroll-bar notification messages or carry out the indicated scrolling. The following table shows the responses to keys that a window must process:

Table 18-4 (Page 1 of 2). Focus Window Message Responses to Keys

Key	**Response**
UP	SB_LINEUP
DOWN	SB_LINEDOWN
PGUP	SB_PAGEUP
PGDN	SB_PAGEDOWN
CTRL+HOME	SB_SLIDERTRACK, with the slider set to the minimum position
CTRL+END	SB_SLIDERTRACK, with the slider set to the maximum position
LEFT	SB_LINELEFT

Table 18-4 (Page 2 of 2). Focus Window Message Responses to Keys

Key	Response
RIGHT	SB_LINERIGHT
CTRL+PGUP	SB_PAGELEFT
CTRL+PGDN	SB_PAGERIGHT
HOME	SB_SLIDERTRACK, with the slider set to the minimum position
END	SB_SLIDERTRACK, with the slider set to the maximum position

For vertical scroll bars that are part of list boxes, the following table shows the responses to keys:

Table 18-5. List Box Responses to Keys

Key	Command
CTRL+UP	SB_SLIDERTRACK, with the slider set to the minimum position
CTRL+DOWN	SB_SLIDERTRACK, with the slider set to the maximum position
F7	SB_PAGEUP
F8	SB_PAGEDOWN

Using Scroll Bars

This section explains how to perform the following tasks:

- Create scroll bars.
- Retrieve a scroll-bar handle.
- Initialize, adjust, and read the scroll-bar range and position.

Creating Scroll Bars

When creating a frame window, you can add scroll bars by specifying the FCF_HORZSCROLL flag, FCF_VERTSCROLL flag, or both flags in the WinCreateStdWindow function. This adds horizontal, vertical, or both (as specified) scroll bars to the frame window. The frame window owns the scroll bars and passes notification messages from the scroll bars to the client window.

The following code fragment adds scroll bars to a frame window:

```
/* Set flags for a main window with scroll bars. */
ULONG ulFrameControlFlags =
    FCF_STANDARD | FCF_HORZSCROLL | FCF_VERTSCROLL;

/* Create the window.                            */
hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &ulFrameControlFlags,
    szClientClass,
    szFrameTitle,
    0,
    (HMODULE) NULL,
    0,
    &hwndClient);
```

Scroll bars created this way have the window identifier FID_HORZSCROLL or FID_VERTSCROLL. To determine the size and position of the scroll bars, the frame window uses the standard size specified by the system values SV_CXVSCROLL and SV_CYHSCROLL. The position always is defined by the right and bottom edges of the frame window.

Another way to create scroll bars is using the WinCreateWindow function. This method is most commonly used for stand-alone scroll bars. Creating scroll bars this way lets you set the size and position of the scroll bars. You also can specify which window should receive notification messages.

The following code fragment creates a stand-alone scroll bar:

```
#define ID_SCROLL_BAR 1

HWND hwndScroll,hwndClient;
hwndScroll = WinCreateWindow(
    hwndClient,                     /* Scroll-bar parent window       */
    WC_SCROLLBAR,                   /* Preregistered scroll-bar class */
    (PSZ) NULL,                     /* No window title                */
    SBS_VERT | WS_VISIBLE,          /* Vertical style and visible     */
    10, 10,                         /* Position & Size               */
    20, 100,                        /* Size                           */
    hwndClient,                     /* Owner                          */
    HWND_TOP,                       /* Z-order position               */
    ID_SCROLL_BAR,                  /* Scroll-bar identifier          */
    NULL,                           /* No class-specific data         */
    NULL);                          /* No presentation parameters     */
```

Retrieving a Scroll-Bar Handle

If you use the WinCreateStdWindow function to create a scroll bar as a child of the frame window, you must be able to retrieve the scroll-bar handle. One way to do this is to use the WinWindowFromID function, the frame-window handle, and a predefined identifier (such as FID_HORZSCROLL or FID_VERTSCROLL), as shown in the following code fragment:

```
HWND hwndFrame,hwndHorzScroll,hwndVertScroll;

hwndHorzScroll = WinWindowFromID(hwndFrame, FID_HORZSCROLL);
hwndVertScroll = WinWindowFromID(hwndFrame, FID_VERTSCROLL);
```

If the standard frame window includes a client window, you can use that handle to access the scroll bars. The idea is to get the frame-window handle first; then, the scroll-bar handle.

```
HWND hwndScroll,hwndClient;

/* Get a handle to the horizontal scroll bar. */
hwndScroll = WinWindowFromID(
    WinQueryWindow(hwndClient, QW_PARENT),
    FID_HORZSCROLL);
```

Using the Scroll-Bar Range and Position

You can initialize the current value and range of a scroll bar to non-default values by sending the SBCDATA structure with class-specific data for a call to WinCreateWindow:

```
#define ID_SCROLL_BAR 1

SBCDATA sbcd;
HWND hwndScroll,hwndClient;

/* Set up scroll-bar control data.                     */
sbcd.posFirst = 200;
sbcd.posLast  = 400;
sbcd.posThumb = 300;

/* Create the scroll bar.                              */
hwndScroll = WinCreateWindow(hwndClient,
    WC_SCROLLBAR,
    (PSZ) NULL,
    SBS_VERT | WS_VISIBLE,
    10, 10,
    20, 100,
    hwndClient,
    HWND_TOP,
    ID_SCROLL_BAR,
    &sbcd,                     /* Class-specific data  */
    NULL);
```

You can adjust a scroll-bar value and range by sending it an SBM_SETSCROLLBAR message:

```
/* Set the scroll-bar value and range. */

WinSendMsg(hwndScroll, SBM_SETSCROLLBAR,
    (MPARAM)300,
    MPFROM2SHORT(200, 400));
```

You can read a scroll-bar value by sending it an SBM_QUERYPOS message:

```
USHORT usSliderPos;

/* Read the scroll-bar value. */
usSliderPos = (USHORT) WinSendMsg(hwndScroll,
    SBM_QUERYPOS, (MPARAM) NULL, (MPARAM) NULL);
```

Similarly, you can set a scroll-bar value by sending an SBM_SETPOS message:

```
/* Set the vertical scroll-bar value. */
WinSendMsg(hwndScroll, SBM_SETPOS, (MPARAM)300, (MPARAM) NULL);
```

You can read a scroll-bar range by sending it an SBM_QUERYRANGE message:

```
MRESULT mr;
USHORT  usMinimum, usMaximum;

/* Read the vertical scroll-bar range.                               */
mr = WinSendMsg(hwndScroll, SBM_QUERYRANGE, (MPARAM) NULL, (MPARAM) NULL);

usMinimum = SHORT1FROMMR(mr);            /* minimum in the low word  */
usMaximum = SHORT2FROMMR(mr);            /* maximum in the high word */
```

Related Messages

This section covers the messages that are related to Scroll Bar Controls.

SBM_QUERYPOS

This message returns the current slider position in a scroll bar window.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sslider (SHORT)
Slider position.

SBM_QUERYRANGE

This message returns the scroll bar range minimum and maximum values.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ReturnCode

sfirst (SHORT)
First bound.

slast (SHORT)
Last bound.

SBM_SETPOS

This message sets the position of the slider in a scroll bar window.

Parameters

param1

sslider (SHORT)
Position of slider.

If this value is outside the scroll-bar range, the slider is moved to the nearest valid position within the range.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred

SBM_SETSCROLLBAR

This message sets the scroll-bar range and slider position.

Parameters

param1

sslider (SHORT)
Position of slider.

If this value is outside the scroll-bar range, the slider is moved to the nearest valid position within the range.

param2

sfirst (SHORT)
First bound.

This value must not be less than 0. If a value less than 0 is supplied, 0 is used as the value.

slast (SHORT)
Last bound.

The value must not be less than 0 or *sfirst*. If a value less than this is supplied, the higher of 0 or *sfirst* is used as the value.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

SBM_SETTHUMBSIZE

This message sets the scroll bar slider size.

Parameters

param1

svisible (SHORT)
Size of the visible part of the document.

stotal (SHORT)
Size of the entire document.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_HSCROLL

This message occurs when a horizontal scroll bar control has a significant event to notify to its owner.

Parameters

param1

usidentifier (USHORT)
Scroll bar control window identifier.

param2

sslider (SHORT)
Slider position.

0 Either the operator is not moving the slider with the pointer device, or for the instance where *uscmd* is SB_SLIDERPOSITION the pointer is outside the tracking rectangle when the button is released.

Other Slider position.

uscmd (USHORT)
Command.

SB_LINELEFT Sent if the operator clicks on the left arrow of the scroll bar, or depresses the VK_LEFT key.

SB_LINERIGHT Sent if the operator clicks on the right arrow of the scroll bar, or depresses the VK_RIGHT key.

SB_PAGELEFT Sent if the operator clicks on the area to the left of the slider, or depresses the VK_PAGELEFT key.

SB_PAGERIGHT Sent if the operator clicks on the area to the right of the slider, or depresses the VK_PAGERIGHT key.

SB_SLIDERPOSITION Sent to indicate the final position of the slider.

SB_SLIDERTRACK If the operator moves the scroll bar slider with the pointer device, this is sent every time the slider position changes.

SB_ENDSCROLL Sent when the operator has finished scrolling, but only if the operator has not been doing any absolute slider positioning.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_VSCROLL

This message occurs when a vertical scroll-bar control has a significant event to notify to its owner.

Parameters

param1

usidentifier (USHORT)
Scroll bar-control window identifier.

param2

sslider (SHORT)
Slider position.

0 Either the operator is not moving the slider with the pointer device, or for the instance when *uscmd* is SB_SLIDERPOSITION the pointer is outside the tracking rectangle when the button is released.

Other Slider position.

uscmd (USHORT)
Command.

SB_LINEUP Sent if the operator clicks on the up arrow of the scroll bar, or presses the VK_UP key.

SB_LINEDOWN Sent if the operator clicks on the down arrow of the scroll bar, or presses the VK_DOWN key.

SB_PAGEUP Sent if the operator clicks on the area above the slider, or presses the VK_PAGEUP key.

SB_PAGEDOWN Sent if the operator clicks on the area below the slider, or presses the VK_PAGEDOWN key.

SB_SLIDERPOSITION Sent to indicate the final position of the slider.

SB_SLIDERTRACK If the operator moves the scroll bar slider with the pointer device, this is sent every time the slider position changes.

SB_ENDSCROLL Sent when the operator has finished scrolling, but only if the operator has not been doing any absolute slider positioning.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Scroll Bar Controls.

SBCDATA

Scroll-bar control data structure.

Syntax

```
typedef struct _SBCDATA {
USHORT       cb;
USHORT       sHilite;
SHORT        posFirst;
SHORT        posLast;
SHORT        posThumb;
SHORT        cVisible;
SHORT        cTotal;
  } SBCDATA;

typedef SBCDATA *PSBCDATA;
```

Fields

cb (USHORT)
Length of control data in bytes.

The length of the control data for a scroll-bar control.

This indicates which part of the scroll bar is to be highlighted, if any.

sHilite (USHORT)
Highlighting code.

ZERO	No highlighting
SB_LINEUP	Line up arrow
SB_LINELEFT	Line left arrow
SB_LINEDOWN	Line down arrow
SB_LINERIGHT	Line right arrow
SB_PAGEUP	Page up arrow
SB_PAGELEFT	Page left arrow
SB_PAGEDOWN	Page down arrow
SB_PAGERIGHT	Page right arrow
SB_SLIDERTRACK	Slider.

posFirst (SHORT)
First bound of the scroll-bar range.

posLast (SHORT)
Last bound of the scroll-bar range.

posThumb (SHORT)
Slider position.

cVisible (SHORT)
Number of data items visible.

cTotal (SHORT)
Number of data items available.

Summary

Following are the operating system messages and structure used with scroll bars.

Table 18-6. Messages Received by a Scroll Bar

Message	Description
SBM_QUERYPOS	Returns the slider position.
SBM_QUERYRANGE	Returns the scroll bar range.
SBM_SETPOS	Sets the position of the slider.
SBM_SETSCROLLBAR	Sets the scroll-bar range and slider positions.
SBM_SETTHUMBSIZE	Sets the scroll bar slider size.

Table 18-7. Messages Generated by a Scroll Bar

Message	Description
WM_HSCROLL	Occurs when a horizontal scroll bar control has a significant event to notify to its owner.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the scroll bar control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the scroll bar control window.
WM_VSCROLL	Occurs when a vertical scroll bar control has a significant event to notify to its owner.

Table 18-8. Scroll-Bar Structure

Structure name	Description
SBCDATA	Scroll-bar control data structure.

Chapter 19. Button Controls

A *button* is a type of control window used to initiate an operation or to set the attributes of an operation. This chapter describes how to create and use buttons in PM applications.

About Button Controls

A button control can appear alone or with a group of other buttons. When buttons are grouped, the user can move from button to button within the group by pressing the Arrow keys. The user also can move among groups by pressing the Tab key.

A user can select a button by clicking it with the mouse, pressing the spacebar when the button has the keyboard focus, or sending a BM_CLICK message. In most cases, a button changes its appearance when selected.

A button control is always owned by another window, usually a dialog window or an application's client window. A button control posts WM_COMMAND messages or sends WM_CONTROL notification messages to its owner when a user selects the button. For further information on messages generated, see "Button Notification Messages" on page 19-7. The owner window receives messages from a button control and can send messages to the button to alter its position, appearance, and enabled/disabled state.

To use a button control in a dialog window, an application specifies the control in a dialog template in the application's resource-definition file. The application processes button messages in the dialog-window procedure.

An application creates a button control in a client window by calling WinCreateWindow, specifying a window class of WC_BUTTON, and identifying the client window as the owner of the button control.

Button Types

There are four main *types* of buttons: push buttons, radio buttons, check boxes, and three-state check boxes. A button's type determines how the button looks and behaves.

A radio button, check box, or three-state check box *controls* an operation; a push button *initiates* an operation. For example, a user might set printing options (such as paper size, print quality, and printer type) in a print-command dialog window containing an array of radio buttons and check boxes. After setting the options, the user would select a push button to tell an application that printing should begin (or be canceled). Then, the application would query the state of each check box and radio button to determine the printing parameters.

Push Buttons

A *push button* is a rectangular window typically used to enable the user to start or stop an operation. When selected, a push button control posts a WM_COMMAND message to its owner window.

The push button can contain a text string, an icon or mini-icon, or a combination of text and images. A push button containing a text string is shown in Figure 19-1 on page 19-2.

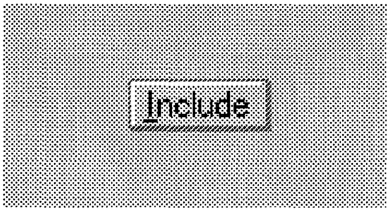


Figure 19-1. Push Button with Text in a Dialog Box

A push button containing a combination of text and a mini-icon is shown in Figure 19-2.

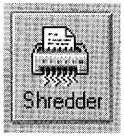


Figure 19-2. Push Button with Icon and Text

A push button containing text and a custom icon is shown in Figure 19-3.

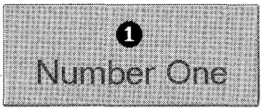
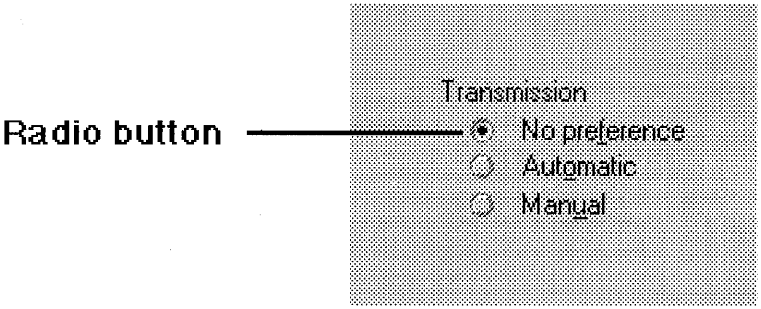


Figure 19-3. Push Button with Text and Custom Icon

Radio Buttons

A *radio button* is a window with text displayed to the right of a small circular indicator. Each time the user selects a radio button, that button's state toggles between *selected* and *unselected*. This state remains until the next time the user selects the button. An application typically uses radio buttons in groups, as shown in Figure 19-4 on page 19-3.



Transmission

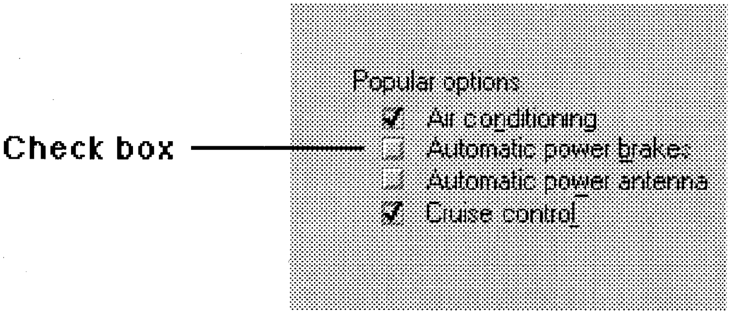
● No preference
○ Automatic
○ Manual

Figure 19-4. Radio Buttons in a Dialog Box

Within a group, usually one button is selected by default, and the user can move the selection to another button by using the cursor keys; however, only one button can be selected at a time. Radio buttons are appropriate if an *exclusive* choice is required from a fixed list of related options. For example, applications often use radio buttons to allow the user to select the screen foreground and background colors. A radio-button control sends WM_CONTROL messages to its owner window.

Check Boxes

Check boxes are similar to radio buttons, except that they can offer *multiple-choice* selection as well as individual choice. Figure 19-5 offers the user a fixed list of choices, with the option of selecting more than one, or even all.



Popular options

☑ Air conditioning
□ Automatic power brakes
□ Automatic power antenna
☑ Cruise control

Figure 19-5. Check Boxes in a Dialog Box

Check boxes also toggle application features *on* or *off.* For example, a word processing application might use a check box to let the user turn word wrapping on or off. A check-box control sends WM_CONTROL messages to its owner window.

Three-State Check Boxes

Three-state check boxes are similar to check boxes, except that they can be displayed in *halftone* as well as selected and unselected. An application might use the halftone state to indicate that, currently, the checkbox is not selectable. A three-state check-box control sends WM_CONTROL messages and posts WM_COMMAND messages to its owner window.

Application-defined Buttons

In addition to using the four predefined button-control types, an application can create button controls that appear as defined by the owner window. When they must be drawn or highlighted, these button controls send WM_CONTROL messages with BN_PAINT as the notification code to their owner windows.

Button Styles

The following table describes the button styles an application can use when creating button controls:

Table 19-1 (Page 1 of 2). Button Styles

Style	Description.
BS_3STATE	Creates a three-state check box (see also BS_CHECKBOX). When the user selects the check box, it sends a WM_CONTROL message to the owner window. The owner should set the check box to the appropriate state: selected, unselected, or halftone.
BS_AUTO3STATE	Creates an auto-three-state check box (see also BS_CHECKBOX). When the user selects the check box, the system automatically sets the check box to the appropriate state: selected, unselected, or halftone.
BS_AUTOCHECKBOX	Creates an auto-check box (see also BS_CHECKBOX). The system automatically toggles the check box between the selected and unselected states each time the user selects the box.
BS_AUTORADIOBUTTON	Creates an auto-radio button (see also BS_RADIOBUTTON). When the user selects an auto-radio button, the system automatically selects the button and removes the selection from the other auto-radio buttons in the group.
BS_AUTOSIZE	Creates a button that is sized automatically to ensure that the contents fit. Note: The *cx* or *cy* parameter of WinCreateWindow must be specified as *-1* to implement the autosize feature.
BS_BITMAP	Creates a push button containing a bit map instead of text. This style can only be implemented with BS_PUSHBUTTON.
BS_CHECKBOX	Creates a check box—a small square that has text displayed to its right. When the user selects a check box, the check box sends a WM_CONTROL message to the owner window. The owner window should toggle the check box between selected and unselected states.
BS_DEFAULT	Creates a push button that has a heavy black border. The user can select this push button by pressing the spacebar. This style is useful for letting the user quickly select the most likely set of options in a dialog window. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.

Table 19-1 (Page 2 of 2). Button Styles

Style	Description.
BS_HELP	Creates a push button that posts a WM_HELP message (instead of a WM_COMMAND message) to its owner window when the user selects the button. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_ICON	Creates a push button containing an icon instead of text.
BS_MINIICON	Creates a push button containing a mini-icon instead of text.
BS_NOBORDER	Creates a push button that has no border. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_NOCURSORSELECT	Creates an auto-radio button that will not be selected automatically when the user moves the cursor to the button using the cursor-movement keys. This style is valid only in combination with the BS_AUTORADIOBUTTON style or the AUTORADIOBUTTON statement in a resource-definition file.
BS_NOPOINTERFOCUS	Creates a radio button or check box that does not receive the keyboard focus when the user selects it. This style is valid in combination with the BS_AUTORADIOBUTTON, BS_RADIOBUTTON, BS_3STATE, BS_AUTO3STATE, BS_AUTOCHECKBOX, and BS_CHECKBOX styles, or the AUTORADIOBUTTON, RADIOBUTTON, AUTOCHECKBOX, or CHECKBOX statements in a resource-definition file.
BS_PUSHBUTTON	Creates a push button—a round-cornered rectangle with text displayed inside it. When selected, the push button posts a WM_COMMAND message to its owner window.
BS_RADIOBUTTON	Creates a radio button—a small circle that has text displayed to its right. Radio buttons usually are used in groups of related, but exclusive, choices. When the user selects a radio button, the button sends a WM_CONTROL message to its owner window. The user should select the button and remove the selection from the other radio buttons in the group.
BS_SYSCOMMAND	Creates a button that posts a WM_SYSCOMMAND message (instead of a WM_COMMAND message) to the owner window when the user selects the button. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_TEXT	Creates a push button containing both text and icons/mini-icons.
BS_USERBUTTON	Creates a user-defined button that sends a WM_CONTROL message to the owner window when the button needs to be drawn, highlighted, or disabled. A user-defined button also posts WM_COMMAND messages to the owner window when the user selects the button.

Default Button Behavior

Following are the messages processed by the predefined button-control window class (WC_BUTTON). Each message is described in terms of how a button control responds to that message.

Table 19-2 (Page 1 of 2). Messages Processed by the WC_BUTTON Class

Message	Description
BM_CLICK	Sends a WM_BUTTON1DOWN and WM_BUTTON1UP message to itself to simulate a user button selection.
BM_QUERYCHECK	Returns the checked state of the button.
BM_QUERYCHECKINDEX	Returns the 0-based index to the selected button in a group. Returns *-1* if no button in the group is selected or if the button receiving the message is not a radio button or an auto-radio button.
BM_QUERYHILITE	Returns the highlighted state of the button.
BM_SETCHECK	Sets the checked state of the button and returns the previous checked state.
BM_SETDEFAULT	Sets the default button state and redraws the button.
BM_SETHILITE	Sets the highlighted state of the button and returns the previous highlighted state.
WM_BUTTON1DBLCLK	Highlights the button and sends a BN_DBLCLICKED notification code when the button-up message arrives.
WM_BUTTON1DOWN	Sets the button window so it can capture mouse input.
WM_BUTTON1UP	If the button window is set to capture mouse input, and if the mouse pointer is inside the button window when the mouse button is released, this message releases the mouse and sends a notification message to the owner window. If the button is a push button, the push button control posts a WM_COMMAND message; otherwise, the button control sends a WM_CONTROL message with the BN_CLICKED notification code.
WM_CHAR	Sets the button window so it can capture mouse input when the spacebar is pressed; releases the mouse when the spacebar is released. Passes other key messages to the default window procedure.
WM_CREATE	Validates the requested button style and sets the window text.
WM_DESTROY	Frees the memory containing the window's text.

Table 19-2 (Page 2 of 2). Messages Processed by the WC_BUTTON Class

Message	Description
WM_ENABLE	Sent when an application changes the enabled state of a window.
WM_MATCHMNEMONIC	Returns TRUE if *mp1* matches a mnemonic in the control window's text.
WM_MOUSEMOVE	Sets the default mouse pointer. If the button has the mouse captured, the button's highlighted state changes as the mouse pointer moves in and out of the button boundary.
WM_PAINT	Draws the button according to its style and current state.
WM_QUERYDLGCODE	Returns the DLGC_BUTTON code combined with other DLGC_ codes that designate the button's type.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETFOCUS	Creates a cursor if the button-control window is receiving the focus. Destroys the cursor if the button-control window is losing the focus.
WM_SETWINDOWPARAMS	Sets the requested window parameters and redraws the button, including the cursor, if the button-control window has the focus.

Button Notification Messages

A button, regardless of its style or type, posts a message to its owner when selected by the user. The message posted by push buttons is ordinarily WM_COMMAND. However, for buttons created with the BS_PUSHBUTTON or BS_USERBUTTON style, the message posted can be changed to WM_HELP or WM_SYSCOMMAND by additionally specifying either the BS_HELP or BS_SYSCOMMAND styles, respectively, when creating the button. A button control that has a style other than BS_PUSHBUTTON or BS_USERBUTTON sends WM_CONTROL messages to its owner when the user selects it.

When the user selects a push button using the mouse pointer, the system automatically highlights the button. The button's window procedure tracks the movement of the pointer until the user releases the button. If the user moves the pointer so that it is outside the button boundary, the system turns off the highlight. The push button control does not post a WM_COMMAND message until the user releases the pointer button, and then, only if the button is released inside the push button boundary. When the owner window receives a WM_COMMAND message from a push button, the low word of the first parameter in the message contains the identifier of the button as specified either in the dialog template or in the WinCreateWindow function when the button was created.

An application should avoid duplicating identifiers for menu items and button controls, because both the items and the controls post identifiers to owner windows as WM_COMMAND messages. However, the application can determine whether a WM_COMMAND message came from a menu or a push button control by looking for the value CMDSRC_MENU or CMDSRC_PUSHBUTTON in the low word of the message's second parameter.

When the user selects any button other than a push button, that button sends a WM_CONTROL message. The application can examine SHORT1FROMMP(*mp1*) in the WM_CONTROL message to find the button identifier, and can examine SHORT2FROMMP(*mp1*) to determine the notification code for the control message. The notification code can be one of the following:

Table 19-3. Notification Code for Button Control Messages

Code	Description
BN_CLICKED	The user selected the button.
BN_DBLCLICKED	The user double-clicked the button.
BN_PAINT	A user-defined button needs to be drawn. Buttons with the BS_USERBUTTON style send this notification code to instruct the owner window to draw the button control. The second message parameter of the WM_CONTROL message contains a pointer to a USERBUTTON structure that contains the information necessary for drawing the button.

When the user selects a check box or radio button, the button control sends the WM_CONTROL message with the BN_CLICKED notification code to the owner window. In response, the owner window should set the display state of the button by sending the appropriate message back to the button.

An application need not respond to WM_CONTROL messages sent by an auto-check box or an auto-radio button; the system automatically sets the states of these buttons.

Button States

An application can query and set the highlighted and checked states of its buttons by sending messages to them. An application can obtain the handle of a button by calling WinWindowFromID, using the parent window handle and the identifier of the button. In the case of a dialog window, the parent window would be the dialog window, and the identifier would be the button identifier from the dialog template.

Button-control text is stored as window text. An application can set and retrieve this text by using the WinSetWindowText and WinQueryWindowText functions. To set the size, position, and visibility of a button control, an application uses the standard window functions.

Custom Buttons

An application can customize the appearance of a button by using the BS_USERBUTTON style in combination with other button styles. The owner window receives WM_CONTROL messages for these custom buttons whenever they must be drawn, highlighted, or disabled.

When a button must be drawn, the owner window receives a WM_CONTROL message with the high word of the first parameter equal to BN_PAINT. The second parameter is a pointer to a USERBUTTON structure that contains information the application needs to draw the button.

An application uses the *hwnd* member of the USERBUTTON structure in a call to the WinQueryWindowRect function to find the bounding rectangle for the button. The *hps* member is used as a presentation space for any drawing. The *fsState* member contains flags that tell an application how to draw the button: highlighted, unhighlighted, or disabled. The *fsStateOld* member contains flags that describe the current highlighted, unhighlighted, or disabled state of the button.

Using Button Controls

This section explains how to perform the following tasks:

- Create a dialog template for a button resource.
- Create a button for a client window.

An application creates a group by setting the WS_GROUP style bit for the first member of the group.

Using Buttons in a Dialog Window

You can define dialog-window buttons as part of a dialog template in a resource-definition file, as shown in the following Resource Compiler source-code fragment.

```
DLGTEMPLATE IDD_BUTTON
BEGIN
    DIALOG  "", 2, 10, 10, 235, 180, WS_VISIBLE, FCF_DLGBORDER
    BEGIN
        AUTORADIOBUTTON "Radio~1", ID_RADIO1, 15, 80, 45, 12, WS_GROUP
        AUTORADIOBUTTON "Radio~2", ID_RADIO2, 15, 60, 45, 12
        AUTORADIOBUTTON "Radio~3", ID_RADIO3, 15, 40, 45, 12
        AUTORADIOBUTTON "Radio~4", ID_RADIO4, 15, 20, 45, 12

        PUSHBUTTON "Button 1", ID_PUSH1,  20  100, 50, 14, WS_GROUP
        PUSHBUTTON "Button 2", ID_PUSH2,  75, 100, 50, 14, WS_GROUP
        PUSHBUTTON "Button 3", ID_PUSH3, 130, 100, 50, 14, WS_GROUP

        CHECKBOX "Check Box 1",     ID_CHECK1, 150, 65, 65, 12, WS_GROUP
        CHECKBOX "no toggle",       ID_CHECK2, 150, 40, 58, 12, WS_GROUP
        AUTOCHECKBOX "Check Box 3", ID_CHECK3, 150, 20, 65, 12, WS_GROUP

        DEFPUSHBUTTON "OK",         DID_OK,     75, 26, 46, 20, WS_GROUP
    END
END
```

Figure 19-6. Defining Dialog-Window Buttons in a Dialog Template

Each button in a dialog window has an identifier (for example, ID_RADIO1) that allows an application to identify the source of the WM_COMMAND and WM_CONTROL messages. An application can use the identifier as the second argument of the WinWindowFromID function to retrieve the button-window handle.

The dialog template also contains the text for each button. For push buttons, this text is displayed in a rectangular box. If the text is too long to fit in the box, the text is clipped. For radio buttons and check boxes, text is displayed to the right of the button. A user selects the button by clicking either the button or the text itself.

The WS_GROUP style identifies the beginning of each new group of buttons. In the preceding example, the four auto-radio buttons are in the same group, and each of the other buttons is in its own group. The auto-radio buttons in the first group can be selected one at a time only. An application must ensure that only one check box in a group is selected at a time. The order in which items can be selected in the group can wrap around from the end of the item list to its beginning.

Notice that the DEFPUSHBUTTON style in the preceding example has the identifier DID_OK. It is customary to include an **OK** button with this identifier in most dialog windows to provide a uniform user interface. The DEFPUSHBUTTON style draws a thick border around a button and allows a user to select the button by pressing the spacebar.

The dialog-window procedure for a dialog window that contains buttons must respond to WM_COMMAND and WM_CONTROL messages. A common strategy is to use auto-radio buttons and auto-check boxes to let the user set a list of capabilities for a command, and, then, let the user execute the command by choosing an **OK** push button. With this strategy, the dialog-window procedure ignores all WM_CONTROL messages that come from auto-radio buttons and auto-check boxes. When the dialog-window procedure receives a WM_COMMAND message for the **OK** push button, the procedure should query the auto-radio buttons and auto-check boxes to determine which options have been selected.

Using Buttons in a Client Window

An application can create a button control using an application client window as the owner. The following code fragment shows how an application can use buttons in client windows:

```
#define ID_PBWINDOW 110
HWND hwndButton,hwndClient;

/* Create a button window. */
hwndButton = WinCreateWindow(hwndClient,   /* Parent window  */
    WC_BUTTON,                             /* Class window   */
    "Test Button"                          /* Button text    */
    WS_VISIBLE |                           /* Visible style  */
    BS_PUSHBUTTON,                         /* Button style   */
    10, 10,                                /* x, y           */
    70, 60,                                /* cx, cy         */
    hwndClient,                            /* Owner window   */
    HWND_TOP,                              /* Top of z-order */
    ID_PBWINDOW,                           /* Identifier     */
    NULL,                                  /* Control data   */
    NULL);                                 /* parameters     */
```

Figure 19-7. Creating a Button Control for a Client Window

Once created in the client window, the button control posts a WM_COMMAND message or sends a WM_CONTROL message to the client-window procedure. This window procedure should examine the message identifier to determine which button posted or sent the message.

An application that has client-window buttons can move and size the buttons when the client window receives a WM_SIZE message. An application can move and size a window by using the WinSetWindowPos function. An application can obtain a window handle for a button control by calling the WinWindowFromID function, specifying the handle of the parent window and the window identifier for each button.

Creating Buttons with Icons and Icon/Text Combinations

The following styles generate buttons containing images or icons:

- BS_ICON
- BS_MINIICON
- BS_BITMAP

The image or icon is activated by specifying the image ID in the button text string. For example, to load an icon (#define ICON_ID 300) and display it with the button, the button text string is set to "#300".

Where text is to be combined with an image, BS_TEXT is selected. To display an icon (#define ICON_ID 300) with the words "My button", the button text string is set to "#300\tMy button". Notice that "\t" is used to separate text from the image ID.

The following code example creates a customized button with text.

```
// presparm.c  -- demonstrates presentation parameters
//                creates a button as a child window
//                and sets its text color

#define  INCL_WIN
#define  INCL_GPI
#include <os2.h>
#include <string.h>
#include "presparm.h"
#include "migrate.h"

int main ( int argc, char *argv[]);

// Internal function prototypes


MRESULT EXPENTRY MyWindowProc( HWND hwnd, MSGID msg
                              , MPARAM mp1, MPARAM mp2 );
int main ( int argc, char *argv[]);
```

Figure 19-8 (Part 1 of 4). Creating a Customized Button with Text

```
// global variables

    HAB  hab;                           // Anchor block handle

int main ( int argc, char *argv[])
{
    HMQ  hmq;                           // Message queue handle
    HWND hwndFrame;                     // Frame window handle
    HWND hwndClient;                    // Client window handle
    QMSG qmsg;                          // Message from message queue
    ULONG flCreate;                     // Window creation control flags
    hab = WinInitialize( 0 );
    hmq = WinCreateMsgQueue( hab, 0 );

    WinRegisterClass( hab, "presparm", MyWindowProc, 0L,  0 );

    flCreate = FCF_SYSMENU | FCF_SIZEBORDER | FCF_TITLEBAR |
               FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

    hwndFrame = WinCreateStdWindow( HWND_DESKTOP, WS_VISIBLE, &flCreate,
        "presparm", "", 0L, 0, ID_WINDOW, &hwndClient );

    while( WinGetMsg( hab, &qmsg, 0, 0, 0 ) )
      WinDispatchMsg( hab, &qmsg );

    WinDestroyWindow( hwndFrame );

   WinDestroyMsgQueue( hmq );
    WinTerminate( hab );
    return 0;
}
```

Figure 19-8 (Part 2 of 4). Creating a Customized Button with Text

```
//
MRESULT EXPENTRY MyWindowProc( HWND hwnd, MSGID msg
                               , MPARAM mp1, MPARAM mp2 )
{
    HPS     hps;                            // PS handle
    BTNCDATA btn;

    typedef struct _FORECOLORPARAM
    {
        ULONG   id;
        ULONG   cb;

        ULONG   ulColor;
    } FORECOLORPARAM;

    typedef struct _FONTPARAM
    {
        ULONG   id;
        ULONG   cb;
        CHAR    szFontNameSize[20];
    } FONTPARAM;

    struct  _PRES                           // pres. params
    {
        ULONG   cb;                         // length
        FORECOLORPARAM fcparam;             // foreground color
        FONTPARAM      fntparam;            // font name & size
    }   pres;
   static  HWND    hwndButton;              // button window handle
    static  POINTL  pt;                     // window size

    switch( msg )
    {
        case WM_CLOSE:
            WinPostMsg( hwnd, WM_QUIT, 0L, 0L );
            return ( (MRESULT) 0 );

    case WM_CREATE:

          // set the foreground color to CLR_RED in
          // the button's presentation parameters
            pres.fcparam.id = PP_FOREGROUNDCOLORINDEX;
            pres.fcparam.cb = sizeof ( pres.fcparam.ulColor );

          pres.fcparam.ulColor = CLR_RED;
```

Figure 19-8 (Part 3 of 4). Creating a Customized Button with Text

```
     // set the font used by the button to 12 point Courier
       pres.fntparam.id = PP_FONTNAMESIZE;
       pres.fntparam.cb = 20;
       strcpy ( pres.fntparam.szFontNameSize, "24.Helv" );

       pres.cb = sizeof ( pres.fcparam ) + sizeof ( pres.fntparam )
       hwndButton = WinCreateWindow ( hwnd     // parent
                       , WC_BUTTON             // class
                       , "#300\tNumber One"    // window text
                       , BS_PUSHBUTTON  |
                         BS_ICON | BS_TEXT     // style
                       , 100, 100              // x, y
                       , 400, 400              // cx, cy
                       , hwnd                  // owner
                       , HWND_TOP              // sibling
                       , 255                   // ID
                       , NULL                  // ctrl data
                       , &pres );          // pres. params
                                               // pmassert
       ( hwndButton, hab );
       return (MRESULT)FALSE;


   case WM_SIZE:
       pt.x = (LONG) SHORT1FROMMP ( mp2 );
       pt.y = (LONG) SHORT2FROMMP ( mp2 );
       WinSetWindowPos ( hwndButton, HWND_TOP
                       , (SHORT)pt.x / 3
                       , (SHORT)pt.y / 2
                       , (SHORT)pt.x / 2
                        , (SHORT)pt.y / 3
                        , SWP_SIZE | SWP_MOVE  | SWP_SHOW );
        return (MRESULT)0;


    case WM_PAINT:
        hps = WinBeginPaint ( hwnd , 0 , NULL );
        GpiErase ( hps );
        WinEndPaint ( hps );
        return ( (MRESULT) 0 );

    default:
        return ( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
return ( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
```

Figure 19-8 (Part 4 of 4). Creating a Customized Button with Text

Related Functions

This section covers the functions that are related to Button Controls.

WinQueryWindowText

This function copies window text into a buffer.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
LONG WinQueryWindowText  (HWND hwnd, LONG lLength, PCH pchBuffer)
```

Parameters

hwnd (HWND) – input
Window handle.

lLength (LONG) – input
Length of *pchBuffer*.

pchBuffer (PCH) – output
Window text.

Returns

lRetLen (LONG) – returns
Length of returned text including the null terminator.

Related Messages

This section covers the messages that are related to Button Controls.

BM_CLICK

An application sends this message to cause the effect of the operator clicking a push button.

Parameters

param1

usUp (USHORT)
Up and down indicator.

TRUE	Perform the default upclick action
FALSE	Perform the default downclick action.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

BM_QUERYCHECK

This message returns the checked state of a button control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

usCheck (USHORT)
Check indicator.

0 The button control is in unchecked state.
1 The button control is in checked state.
2 The button control is in indeterminate state.

BM_QUERYCHECKINDEX

This message returns the zero-based index of a checked radio button.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sIndex (SHORT)
Radio-button index.

–1 No radio button of the group is checked, or this button control does not have the style BS_RADIOBUTTON or BS_AUTORADIOBUTTON.

Other Zero-based index of the checked radio button of the group.

BM_QUERYHILITE

This message returns the highlighting state of a button control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Highlight indicator.

TRUE	The button control is displayed in highlighted state.
FALSE	The button control is displayed in unhighlighted state.

BM_SETCHECK

This message sets the checked state of a button control.

Parameters

param1

uscheck (USHORT)

Check state.

0 Display the button control in the unchecked state
1 Display the button control in the checked state
2 Display a 3-state button control in the indeterminate state.

param2

ulReserved (ULONG)

Reserved value, should be 0.

Returns

usoldstate (USHORT)

Old check state of the button control.

0 Unchecked
1 Checked
2 Indeterminate.

BM_SETDEFAULT

This message sets the default state of a button control.

Parameters

param1

usdefault (USHORT)
Default state.

TRUE	Display the button control in the default state
FALSE	Display the button control in the nondefault state.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful operation
FALSE	Error occurred.

BM_SETHILITE

This message sets the highlight state of a button control.

Parameters

param1

ushilite (USHORT)
Highlight indicator.

TRUE	Display the button control in the highlighted state
FALSE	Display the button control in the unhighlighted state.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

foldstate (BOOL)
Old highlight state.

TRUE	The button control was in highlighted state
FALSE	The button control was in unhighlighted state.

WM_MATCHMNEMONIC

This message is sent by the dialog box to a control window to determine whether a typed character matches a mnemonic in its window text.

Parameters

param1

usmatch (USHORT)
Match character.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Match indicator.

TRUE	Mnemonic found
FALSE	Mnemonic not found, or an error occurred.

Related Data Structures

This section covers the data structures that are related to Button Controls.

BTNCDATA

Button-control-data structure.

Syntax

```
typedef struct _BTNCDATA {
USHORT        cb;
USHORT        fsCheckState;
USHORT        fsHiliteState;
LHANDLE       hImage;
 } BTNCDATA;

typedef BTNCDATA *PBTNCDATA;
```

Fields

cb (USHORT)

Length of the control data in bytes.

This is the length of the control data for a button control.

fsCheckState (USHORT)

Check state of button.

This is the same value as returned by the BM_QUERYCHECK message and passed to the BM_SETCHECK message.

fsHiliteState (USHORT)

Highlighting state of button.

This is the same value as returned by the BM_QUERYHILITE message and passed to the BM_SETHILITE message.

hImage (LHANDLE)

Resource handle for icon or bit map.

USERBUTTON

User-button data structure.

Syntax

```
typedef struct _USERBUTTON {
HWND        hwnd;
HPS         hps;
ULONG       fsState;
ULONG       fsStateOld;
 } USERBUTTON;

typedef USERBUTTON *PUSERBUTTON;
```

Fields

hwnd (HWND)
Window handle.

hps (HPS)
Presentation-space handle.

fsState (ULONG)
New state of user button.

fsStateOld (ULONG)
Old state of user button.

Summary

Following are the OS/2 functions, messages, and structures used with button controls:

Table 19-4. Button-Control Functions

Function Name	Description
WinCreateWindow	Creates a new window.
WinQueryWindowText	Copies window text into a buffer.
WinSetWindowText	Sets the window text for the specified window.
WinWindowFromID	Returns the handle of the child window with the specified identify.

Table 19-5. Messages Received by a Button Control

Message	Description
BM_CLICK	Application sends this message to cause the effect of the user clicking a push button.
BM_QUERYCHECK	Returns the zero-based index of a checked radio button.
BM_QUERYCHECKINDEX	Returns the zero-based index of a checked radio button.
BM_QUERYHILITE	Returns the highlighting state of a button control.
BM_SETCHECK	Sets the checked state of a button control.
BM_SETDEFAULT	Sets the default state of a button control.
BM_SETHILITE	Sets the highlight state of a button control.

Table 19-6. Messages Generated by a Button Control

Message	Description
WM_COMMAND	The button control generates this message when a push button of style BS_PUSHBUTTON is pressed or when it receives a BM_CLICK message. The button control posts the message to the queue of the control owner.
WM_CONTROL	The button control generates this message and sends it to its owner when: • Its style is not BS_PUSHBUTTON and the button is pressed. • It receives a BM_CLICK message. • Its style is BS_PUSHBUTTON and the button is clicked or double clicked.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointer moves over the control window, allowing the owner to set the pointer.
WM_ENABLE	Sets the enable state of a window.
WM_HELP	The button control generates this message and posts it to the queue of its owner, if it has the style of BS_HELP and a pushbutton is pressed, or when it receives a BM_CLICK message.
WM_MATCHMNEMONIC	Sent by the dialog box to a control window to determine whether a typed character matches a mnemonic in its window text.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the button control window procedure window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the button control window procedure window parameters.
WM_SYSCOMMAND	If the button control is specified with a style of BS_SYSCOMMAND but not with BS_HELP, the button control generates this message and posts it to the queue of its owner when a pushbutton is pressed, or when it receives a BM_ CLICK message.

Table 19-7. Button-Control Structures

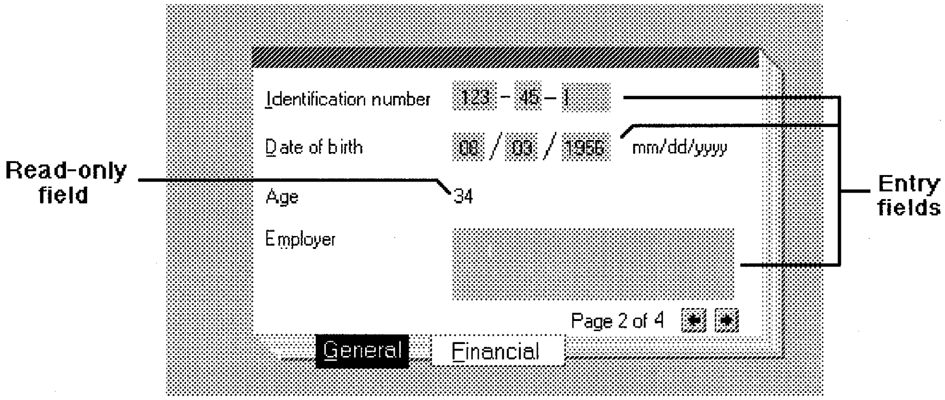
Structure Name	Description
BTNCDATA	Button-control data structure.
USERBUTTON	User-button structure.

Chapter 20. Entry-Field Controls

An *entry field* is a control window that enables a user to view and edit a single line of text. This chapter describes how to create and use entry-field controls in your PM applications.

About Entry Fields

An entry field provides the text-editing capabilities of a simple text editor and is useful whenever an application requires a short line of text from the user as illustrated in Figure 20-1.



Identification number 123 - 45 - 1

Date of birth 08 / 03 / 1956 mm/dd/yyyy

Age 34

Employer

Figure 20-1. Example of Entry Fields

If the application requires more sophisticated text-editing capabilities and multiple lines of text from the user, the application can use a multiple-line entry (MLE) field. See *Presentation Manager Programming Guide - Advanced Topics* for more information about MLE controls.

Both the user and the application can edit text in an entry field. Applications typically use entry fields in dialog windows, although they can be used in non-dialog windows as well.

An application creates an entry field by specifying either the WC_ENTRYFIELD window class in the WinCreateWindow function or the ENTRYFIELD statement in a resource-definition file.

Entry-Field Styles

An entry field has a style that determines how it appears and behaves. An application specifies the style in either the WinCreateWindow function or the ENTRYFIELD statement in a resource-definition file. An application can specify a combination of the following styles for an entry field.

Table 20-1. Entry-Field Styles

Style	Description
ES_ANY	Allows the entry-field text to contain a mixture of double-byte and single-byte characters.
ES_AUTOSCROLL	Automatically scrolls text horizontally to show the insertion point.
ES_AUTOSIZE	Automatically sets the size of the entry field, based on the width of the field's text string and the metrics of the current system font. This style can set the width, height, or both–whichever has a value of *-1* in the WinCreateWindow function or resource-definition file. This style affects only the initial size of the entry field; it does not adjust the size as the font or text-string width changes.
ES_AUTOTAB	Automatically moves the cursor to the next control window when the user enters the maximum number of characters.
ES_CENTER	Centers text within the entry field.
ES_DBCS	Specifies that the entry-field text consist of double-byte characters only.
ES_LEFT	Left-aligns text within the entry field.
ES_MARGIN	Draws a border around the entry field. The border is 1/2-character wide and 1/4-character high. Without this style, the application draws no border around the entry field. The width of the entry-field rectangle is increased on all sides by the width of this margin. After an entry field with the ES_MARGIN style is created, the WinQueryWindowRect function returns a larger rectangle that includes this margin and whose origin, therefore, is different from the origin specified when the entry field was created. If an application does not adjust for this size difference when moving or sizing an entry field, the entry field becomes larger after each moving and sizing operation.
ES_MIXED	Allows the entry-field text to contain a mixture of single-byte and double-byte characters. Unlike the ES_ANY style, this style lets ASCII DBCS data be converted to EBCDIC DBCS data without causing an overflow condition.
ES_READONLY	Prevents the user from entering or editing text in the entry field.
ES_RIGHT	Right-aligns text within the entry field.
ES_SBCS	Specifies that the entry-field text must consist of single-byte characters only.
ES_UNREADABLE	Displays each character as an asterisk (*). This style is useful when obtaining a password from the user.

Entry-Field Notification Codes

An entry field is always owned by another window. A WM_CONTROL notification message is sent to the owner whenever an event occurs in the entry field. This message contains a notification code that specifies the exact nature of the event. An entry field can send the notification codes described in the following table to its owner.

Table 20-2. Notification of Entry-Field Events

Notification Code	Description
EN_CHANGE	Indicates that the contents of an entry field have changed.
EN_INSERTMODETOGGLE	Indicates that the insert mode has been toggled.
EN_KILLFOCUS	Indicates that an entry field has lost the keyboard focus.
EN_MEMERROR	Indicates that an entry field cannot allocate enough memory to perform the requested operation, such as extending the text limit.
EN_OVERFLOW	Indicates that either the user or the application attempted to exceed the text limit.
EN_SCROLL	Indicates that the text in an entry field is about to scroll.
EN_SETFOCUS	Indicates that an entry field received the keyboard focus.

An application typically ignores notification messages from an entry field, thereby allowing default text editing to occur. For more specialized uses, an application can use notification messages to filter input. For example, if an entry field is intended for numbers only, an application can use the EN_CHANGE notification code to check the contents of the entry field each time the user enters a non-numeric character.

As an alternative, an application can prevent inappropriate characters from reaching an entry field by using EN_SETFOCUS and EN_KILLFOCUS, in filter code, placed in the main message loop. Whenever the entry field has the keyboard focus, the filter code can intercept and filter WM_CHAR messages before the WinDispatchMsg function passes them to the entry field. An application also can respond to certain keystrokes, such as the Enter key, as long as the entry-field control has the keyboard focus.

Default Entry-Field Behavior

The following table lists and describes all the messages specifically handled by the predefined entry-field control-window class (WC_ENTRYFIELD).

Table 20-3 (Page 1 of 3). Messages Handled by WC_ENTRYFIELD Class

Message	Description
EM_CLEAR	Deletes the current text selection from the control window.
EM_COPY	Copies the current text selection to the system clipboard, in CF_TEXT format.
EM_CUT	Copies the current text selection to the system clipboard, in CF_TEXT format, and deletes the selection from the control window.
EM_PASTE	Copies the current contents of the system clipboard that have CF_TEXT format, replacing the current text selection in the control window.
EM_QUERYCHANGED	Returns TRUE if the text has changed since the last EM_QUERYCHANGED message.

Table 20-3 (Page 2 of 3). Messages Handled by WC_ENTRYFIELD Class

Message	**Description**
EM_QUERYFIRSTCHAR	Returns the offset to the first character visible at the left edge of the control window.
EM_QUERYREADONLY	Determines whether the entry field is in the read-only state.
EM_QUERYSEL	Returns a long word that contains the offsets for the first and last characters of the current selection in the control window.
EM_SETFIRSTCHAR	Scrolls the text so that the character at the specified offset is the first character visible at the left edge of the control window.
EM_SETINSERTMODE	Toggles the text-entry mode between insert and overstrike.
EM_SETREADONLY	Sets the entry field to the read-only state.
EM_SETSEL	Sets the current selection to the specified character offsets.
EM_SETTEXTLIMIT	Allocates memory from the control heap for the specified maximum number of characters, returning TRUE if it is successful and FALSE if it is not. Failure causes the entry field to send a WM_CONTROL message with the EN_MEMERROR notification code to the owner window.
WM_ADJUSTWINDOWPOS	Changes the size of the control rectangle if the control has the ES_MARGIN style.
WM_BUTTON1DBLCLK	Occurs when the user presses mouse button 1 twice.
WM_BUTTON1DOWN	Sets the mouse capture and keyboard focus to the entry field, and prepares to track the movement of the mouse during WM_MOUSEMOVE messages.
WM_BUTTON1UP	Releases the mouse.
WM_BUTTON2DOWN	Returns TRUE to prevent this message from being processed further.
WM_BUTTON3DOWN	Returns TRUE to prevent this message from being processed further.
WM_CHAR	Handles text entry and other keyboard input events.
WM_CREATE	Validates the requested style and sets the window text.
WM_DESTROY	Frees the memory used for the window text.
WM_ENABLE	Sent when an application changes the enabled state of a window.
WM_MOUSEMOVE	If the mouse button is down, the entry field tracks the text selection. If the mouse button is up, the entry field sets the mouse pointer to the default arrow shape.
WM_PAINT	Draws the entry field and text.
WM_QUERYDLGCODE	Returns the predefined DLGC_ENTRYFIELD constant.

Table 20-3 (Page 3 of 3). Messages Handled by WC_ENTRYFIELD Class

Message	Description
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETFOCUS	If the entry field is gaining the focus, it creates a cursor and sends the owner window a WM_CONTROL message with the EN_SETFOCUS notification code. If the entry field is losing the focus, it destroys the current cursor and sends the owner window a WM_CONTROL message with the EN_KILLFOCUS notification code.
WM_SETSELECTION	Toggles the current selection status.
WM_SETWINDOWPARAMS	Sets the specified window parameters, redraws the entry field, and sends the owner window a WM_CONTROL message with the EN_CHANGE notification code.
WM_TIMER	Blinks the insertion point if the entry field has the focus. The entry field scrolls the text, if necessary, while extending the selection to text that becomes visible in the window.

Entry-Field Text Editing

The user can insert (type) text or numeric values in an entry field when that entry field has the keyboard focus. An application can insert text by using the WinSetWindowText function. An application can insert numeric values by using the WinSetDlgItemShort function. The text or numeric value is inserted into the entry field at the cursor position.

The entry field's entry mode, either insert or overstrike, determines what happens when the user enters text. The user sets the entry mode by pressing the Insert key; the entry mode toggles each time the Insert key is pressed. The application can set the entry mode by sending the EM_SETINSERTMODE message to the entry field.

The cursor position, identified by a blinking bar, is specified by a character offset relative to the beginning of the text. The user can set the cursor position by using the mouse or the Arrow keys. An application can set the cursor position by using the EM_SETSEL message. This message directs the entry field to move the blinking bar to the given character position.

The EM_SETSEL message also sets the selection. The selection is one or more characters of text on which the entry field carries out an operation, such as deleting or copying to the clipboard. The user selects text by pressing the Shift key while moving the cursor, or by pressing mouse button 1 while moving the mouse. An application selects text by using the EM_SETSEL message to specify the cursor position and the anchor point. The selection includes all text between the cursor position and the anchor point. If the cursor position and anchor point are equal, there is no selection. An application can retrieve the selection (cursor position and anchor point) by using the EM_QUERYSEL message.

The user can delete characters, one at a time, by pressing the Delete key or the Backspace key. The Delete key deletes the character to the right of the cursor; the Backspace key deletes the character to the left of the cursor. The user also can delete a group of characters by selecting them and pressing the Delete key. An application can delete selected text by using the EM_CLEAR message.

An application can use the EM_QUERYCHANGED message to determine whether the contents of an entry field have changed.

An application can prevent the user from editing an entry field by setting the ES_READONLY style in the WinCreateWindow function or in the ENTRYFIELD statement in the resource-definition file. The application also can set and query the read-only state by using the EM_SETREADONLY and ES_QUERYREADONLY messages.

If text extends beyond the left or right edges of an entry field, the user can scroll the text by using the Arrow keys. An application can scroll the text by using the EM_SETFIRSTCHAR message to specify the first character visible at the left edge of the entry field. For scrolling to occur, the entry field must have the ES_AUTOSCROLL style. An application can use the EM_QUERYFIRSTCHAR message to obtain the first character that is currently visible.

Entry-Field Control Copy and Paste Operations

The user can cut, copy, and paste text in an entry field by using the Shift+Delete and Ctrl+Insert key combinations. An application, either by itself or in response to the user, can cut, copy, and paste text by using the EM_CUT, EM_COPY, and EM_PASTE messages. An application can use the ES_CUT and EM_COPY messages to copy the selected text to the clipboard. The EM_CUT message also deletes the text (EM_COPY does not). The EM_PASTE message copies the text on the clipboard to the current position in the entry field, replacing any existing text with the copied text. An application can delete the selected text, without copying it to the clipboard, by using the EM_CLEAR message.

Entry-Field Text Retrieval

An application can retrieve selected text from an entry field by calling WinQueryWindowText and then sending an EM_QUERYSEL message to retrieve the offsets to the first and last characters of the text selection. These offsets are used to retrieve selected text.

An application can retrieve numeric values by calling WinQueryDlgItemShort, passing the entry-field identifier and the handle of the owner window. WinQueryDlgItemShort converts the entry-field text to a signed or unsigned integer and returns the value in a specified variable. The application can use the WinWindowFromID function to retrieve the handle of the control window. The entry-field identifier is specified in the dialog template in the application's resource-definition file.

Using Entry-Field Controls

This section explains how to perform the following tasks:

- Create an entry field in a dialog or client window.
- Change the default size of the entry field.

Creating an Entry Field in a Dialog Window

A dialog window usually serves as the parent and owner of an entry field. The dialog window often includes a button that indicates whether the user wants to carry out an operation. When the user selects the button, the application queries the contents of the entry field and proceeds with the operation.

The definition of an entry field in an application's resource-definition file sets the initial text, window identifier, size, position, and style of the entry field. The following example shows how to define an entry field as part of a dialog template:

```
DLGTEMPLATE IDD_SAMPLE
BEGIN
    DIALOG "Sample Dialog", ID_DLG, 7, 7, 253, 145, FS_DLGBORDER,0
    BEGIN
        DEFPUSHBUTTON "~OK", DID_OK, 8, 151, 50, 23, WS_GROUP
        ENTRYFIELD "Here is some text", ID_ENTFLD, 42, 46, 68, 15,
            ES_MARGIN | ES_AUTOSCROLL
    END
END
```

Creating an Entry Field in a Client Window

To create an entry field in a non-dialog window, an application calls WinCreateWindow with the window class WC_ENTRYFIELD. The entry field is owned by an application's client window, whose window procedure receives notification messages from the entry field.

The following code fragment shows how to create an entry field in a client window:

```
#define ID_ENTRYFIELD 5

HWND hwnd, hwndEntryField1, hwndClient;
LONG xPos   =  50, yPos    = 100;
LONG xWidth = 100, yHeight =  20;

hwndEntryField1 = WinCreateWindow(
    hwndClient,             /* Parent-window handle */
    WC_ENTRYFIELD,          /* Window class         */
    "initial text",         /* Initial text         */
    WS_VISIBLE    |         /* Visible when created */
    ES_AUTOSCROLL |         /* Scroll text          */
    ES_MARGIN,              /* Create a border      */
    xPos, yPos,             /* x and y position     */
    xWidth, yHeight,        /* Width and height     */
    hwnd,                   /* Owner-window handle  */
    HWND_TOP,               /* Z-order position     */
    ID_ENTRYFIELD,          /* Window identifier    */
    NULL,                   /* No control data      */
    NULL);                  /* No pres. parameters  */
```

Figure 20-2. Code for Creating an Entry Field in a Client Window

Changing the Default Size of an Entry Field

The default text limit of an entry field is 32 characters. An application can set a non-default size when creating an entry field by setting the *cchEditLimit* member of an ENTRYFDATA structure and supplying a pointer to the structure as the *pCtlData* parameter to WinCreateWindow.

The following code fragment creates an entry field with a text limit of 12 characters:

```
HWND hwndEntryField2;
HWND hwndClient;
ENTRYFDATA efd;
LONG xPos   = 50, yPos    = 50;
LONG xWidth = -1, yHeight = -1;
                           /* must be -1 for
                                    ES_AUTOSIZE */

                           /* Initialize the ENTRYFDATA
                                        structure. */
efd.cb = sizeof(ENTRYFDATA);
efd.cchEditLimit = 12;
efd.ichMinSel = 0;
efd.ichMaxSel = 0;

                           /* Create the entry
                                            field. */
hwndEntryField2 = WinCreateWindow(
    hwndClient,            /* Parent-window handle */
    WC_ENTRYFIELD,         /* Window class         */
    "projects.xls",        /* No initial text      */
    WS_VISIBLE |           /* Visible when created */
    ES_MARGIN  |           /* Create a border.     */
    ES_AUTOSIZE,           /* System sets the size */
    xPos, yPos,            /* x and y positions    */
    xWidth, yHeight,       /* Width and height     */
    hwndClient,            /* Owner-window handle  */
    HWND_TOP,              /* Z-order position     */
    0,                     /* Window identifier    */
    &efd,                  /* Control data         */
    NULL);                 /* No pres. parameters  */
```

Figure 20-3. Code for Creating Entry Field with 12-Character Text Limit

To expand or reduce the text limit after creating the entry field, an application can send an EM_SETTEXTLIMIT message specifying a new maximum text limit for the entry field. The following code fragment increases to 20 characters the text limit of the entry field created in the previous example:

```
WinSendMsg(hwndEntryField2, EM_SETTEXTLIMIT,
    (MPARAM)20, (MPARAM)0);
```

Figure 20-4. Code for Creating Entry Field with 20-Character Text Limit

Retrieving Text From an Entry Field

An application can use the WinQueryWindowTextLength and WinQueryWindowText functions to retrieve the text from an entry field. WinQueryWindowTextLength returns the length of the text; WinQueryWindowText copies the window text to a buffer.

Typically, an application needs to retrieve the text from an entry field only if the user changes the text. An entry field sends an EN_CHANGE notification code in the low word of the first message parameter of the WM_CONTROL message whenever the text changes. The following code fragment sets a flag when it receives the EN_CHANGE code, checks the flag during the WM_COMMAND message and, if it is set, retrieves the text of the entry field:

```
HWND hwnd;
ULONG msg;
MPARAM mp1;
CHAR chBuf[64];
HWND hwndEntryField;
LONG cbTextLen;
LONG cbTextRead;
static BOOL fFieldChanged = FALSE;

switch (msg) {
    case WM_CONTROL:
        switch (SHORT1FROMMP(mp1)) {
            case IDD_ENTRYFIELD:

                /* Check if the user changed the entry-field text. */
                if ((USHORT) SHORT2FROMMP(mp1) == EN_CHANGE)
                    fFieldChanged = TRUE;
                return 0;
        }

    case WM_COMMAND:
        switch (SHORT1FROMMP(mp1)) {
            case DID_OK:

                /* If the user changed the entry-field text,       */
                /* obtain the text and store it in a buffer.       */
                if (fFieldChanged) {
                    hwndEntryField = WinWindowFromID(hwnd,
                        IDD_ENTRYFIELD);
                    cbTextLen = WinQueryWindowTextLength(hwndEntryField);
                    cbTextRead = WinQueryWindowText(hwndEntryField,
                        sizeof(chBuf), chBuf);
                    .
                    . /* Do something with the text.                */
                    .
                }
                WinDismissDlg(hwnd, 1);
                return 0;
        }
}
```

Figure 20-5. Code for Flagging a Text Change in an Entry Field

Related Functions

This section covers the functions that are related to Entry Field Controls.

WinQueryWindowTextLength

This call returns the length of the window text, excluding any null termination character.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
LONG WinQueryWindowTextLength  (HWND hwnd)
```

Parameters

hwnd (HWND) – input
Window handle.

Returns

lRetLen (LONG) – returns
Length of the window text.

Related Messages

This section covers the messages that are related to Entry Field Controls.

EM_CLEAR

This message deletes the text that forms the current selection.

Parameters

param1

ulReserve (ULONG)
Reserved value, should be 0.

param2

ulReserve (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

EM_COPY

This message copies the current selection to the clipboard.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

EM_CUT

This message copies the text that forms the current selection to the clipboard, and then deletes it from the entry field control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

EM_PASTE

This message replaces the text that forms the current selection with text from the clipboard.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred. For example, if the text to be inserted does not fit in the entry field control without overflowing the text limit set by the EM_SETTEXTLIMIT message, in which instance no text is inserted.

EM_QUERYCHANGED

This message enquires if the text of the entry field control has been changed since the last enquiry.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Changed indicator.

TRUE	The text in the entry field control has been changed since the last time it received this message or a WM_QUERYWINDOWPARAMS message.
FALSE	All other situations.

EM_QUERYFIRSTCHAR

This message returns the zero-based offset of the first character visible at the left edge of an entry-field control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sOffset (SHORT)
Zero-based offset.

EM_QUERYREADONLY

This message returns the read only state of an entry field control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Read only state indicator.

TRUE	Read only state is enabled.
FALSE	Read only state is disabled.

EM_QUERYSEL

This message gets the zero-based offsets of the bounds of the text that forms the current selection.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ReturnCode

sMinSel (SHORT)
Offset of the first character in the selection.

sMaxSel (SHORT)
Offset of the first character after the selection.

EM_SETFIRSTCHAR

This message specifies the offset of the character to be displayed in the first position of the entry field control.

Parameters

param1

sOffset (SHORT)
Zero-based offset of the first character to be displayed.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred. For example, because *sOffset* is not valid.

EM_SETINSERTMODE

This message sets the insert mode of an entry field.

Parameters

param1

usInsert (USHORT)

Insert mode indicator.

TRUE	Enable insert mode.
FALSE	Enable overtype mode.

param2

ulReserved (ULONG)

Reserved value, should be 0.

Returns

rc (BOOL)

Previous insert mode indicator.

TRUE	Insert mode was previously enabled.
FALSE	Overtype mode was previously enabled.

EM_SETREADONLY

This message sets the read only state of an entry field control.

Parameters

param1

usReadOnly (USHORT)
Read only state indicator.

TRUE	Enable read only state
FALSE	Disable read only state.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Previous read only state indicator.

TRUE	Read only state was previously enabled.
FALSE	Read only state was previously disabled.

EM_SETSEL

This message sets the zero-based offsets of the bounds of the text that forms the current selection.

Parameters

param1

usminsel (USHORT)
Offset of the first character in the selection.

usmaxsel (USHORT)
Offset of the first character after the selection.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

EM_SETTEXTLIMIT

This message sets the maximum number of bytes that an entry field control can contain.

Parameters

param1

sTextLimit (SHORT)
Maximum number of characters in the entry field control.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred. For example, because not enough storage can be allocated.

WM_CONTROL (in Entry Fields)

For the cause of this message, see "WM_CONTROL" on page 16-5.

Parameters

param1

id (USHORT)
Control window identity.

usnotifycode (USHORT)
Notify code.

EN_CHANGE
The content of the entry field control has changed, and the change has been displayed on the screen.

EN_KILLFOCUS
The entry field control is losing the focus.

EN_MEMERROR
The entry field control cannot allocate the storage necessary to accommodate window text of the length implied by the EM_SETTEXTLIMIT message.

EN_OVERFLOW
The entry field control cannot insert more text than the current text limit. The text limit may be changed with the EM_SETTEXTLIMIT message.

If the recipient of this message returns TRUE, then the entry field control retries the operation, otherwise it terminates the operation.

EN_SCROLL
The entry field control is about to scroll horizontally. This can happen in these circumstances:

- The application has issued a WinScrollWindow call
- The content of the entry field control has changed
- The caret has moved
- The entry field control must scroll to show the caret position.

EN_SETFOCUS
The entry field control is receiving the focus.

param2

hwndcontrolspec (HWND)
Entry field control window handle.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Related Data Structures

This section covers the data structures that are related to Entry Field Controls.

ENTRYFDATA

Entry-field control data structure.

Syntax

```
typedef struct _ENTRYFDATA {
USHORT       cb;
USHORT       cchEditLimit;
USHORT       ichMinSel;
USHORT       ichMaxSel;
 } ENTRYFDATA;

typedef ENTRYFDATA *PENTRYFDATA;
```

Fields

cb (USHORT)

Length of control data in bytes.

The length of the control data for an entry field control.

cchEditLimit (USHORT)

Edit limit.

This is the maximum number of characters that can be entered into the entry field control.

If the operator tries to enter more text into an entry field control than is specified by the text limit set by the EM_SETTEXTLIMIT message, the entry field control indicates the error by sounding the alarm and does not accept the characters.

ichMinSel (USHORT)

Minimum selection.

ichMaxSel (USHORT)

Maximum selection.

The *ichMinSel* and *ichMaxSel* parameters identify the current selection within the entry field control. Characters within the text with byte offsets less than the *ichMaxSel* parameter and greater than or equal to the *ichMinSel* parameter are the current selection. The cursor is positioned immediately before the character identified by the *ichMaxSel* parameter.

If the *ichMinSel* parameter is equal to the *ichMaxSel* parameter, the current selection becomes the insertion point.

If the *ichMinSel* parameter is equal to 0 and the *ichMaxSel* is greater than or equal to text limit set by the EM_SETTEXTLIMIT message, the entire text is selected.

Summary

Following are the OS/2 functions, messages, and data structures used with entry-field controls.

Table 20-4. Entry-Field Functions

Function Name	Description
WinQueryDlgItemShort	Converts the text of a dialog item into an integer value.
WinQueryWindowText	Copies window text into a buffer.
WinQueryWindowTextLength	Returns the length of the window text, excluding any NULL termination character.
WinSetDlgItemShort	Converts an integer value into the text of a dialog item.
WinSetWindowText	Sets the window text for a specified window.

Table 20-5. Messages Received by an Entry Field

Message	Description
EM_CLEAR	Deletes the text that forms the current selection.
EM_COPY	Copies the current selection to the clipboard.
EM_CUT	Copies the text that forms the current selection to the clipboard, then deletes it from the entry field control.
EM_PASTE	Replaces the text that forms the current selection with text from the clipboard.
EM_QUERYCHANGED	Queries whether the text of the entry field control has been changed since the last inquiry.
EM_QUERYFIRSTCHAR	Returns the zero-based offset of the first character displayed in the entry field control.
EM_QUERYREADONLY	Returns the read-only state of an entry field control.
EM_QUERYSEL	Gets the zero-based offsets of the bounds of the text that forms the current selection.
EM_SETFIRSTCHAR	Specifies the offset of the character to be displayed in the first position of the entry field control.
EM_SETINSERTMODE	Sets the insert mode of an entry field.
EM_SETREADONLY	Sets the read-only state of an entry field control.
EM_SETSEL	Sets the zero-based offsets of the bounds of the text that forms the current selection.
EM_SETTEXTLIMIT	Sets the maximum number of bytes that an entry field control can contain.

Table 20-6. Message Generated by an Entry Field

Message	Description
WM_CHAR	Occurs when the user presses a key.
WM_CONTROL	Occurs when a control has a significant event to notify to its owner.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the entry field control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the entry field control window parameters.

Table 20-7. Entry-Field Structure

Structure Name	Description
ENTRYFDATA	Entry-field data structure

Chapter 21. List-Box Controls

A *list box* is a control window that displays several text items at a time, one or more of which can be selected by the user. This chapter explains how to create and use list-box controls in PM applications.

About List Boxes

An application uses a list box when it requires a list of selectable fields that is too large for the display area or a list of choices that can change dynamically. Each list item contains a text string and a handle. Usually, the text string is displayed in the list-box window; but the handle is available to the application to reference other data associated with each of the items in the list.

A list box always is owned by another window that receives messages from the list box when events occur, such as when a user selects an item from the list box. Typically, the owner is a dialog window (as shown in Figure 21-1,) or the client window of an application frame window. The client- or dialog-window procedure defined by the application responds to messages sent from the list box.

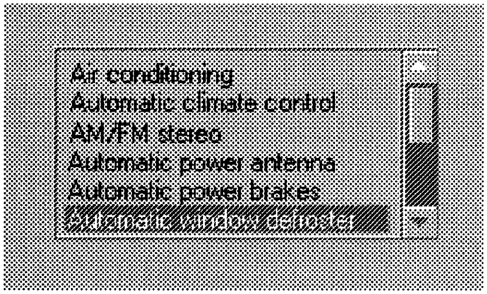


Figure 21-1. List Box in a Dialog Box

A list box always contains a scroll bar for use when the list box contains more items than can be displayed in the list-box window. The list box responds to mouse clicks in the scroll bar by scrolling the list; otherwise, the scroll bar is disabled.

The maximum number of items permitted in a list box is 32767.

Using List Boxes

An application uses a list-box control to display a list in a window. List boxes can be displayed in standard application windows, although they are more commonly used in dialog windows. In either case, notification messages are sent from the list box to its owner window, enabling the application to respond to user actions in the list.

Once a list box is created, the application controls the insertion and deletion of list items. Items can be inserted at the end of the list, automatically sorted into the list, or inserted at a specified index position. Applications can turn list drawing on and off to speed up the process of inserting numerous items into a list.

The owner-window procedure of the list box receives messages when a user manipulates the list-box data. Most default list actions (for example, highlighting selections and scrolling) are handled automatically by the list box itself. The application controls the responses when the user chooses an item in the list, either by double-clicking the item or by pressing Enter after an item is highlighted. The list box also notifies the application when the user changes the selection or scrolls the list.

Normally, list items are text strings drawn by a list box. An application also can draw and highlight the items in a list. This enables the application to create customized lists that contain graphics. When an application creates a list box with the LS_OWNERDRAW style, the owner of the list box receives a WM_DRAWITEM message for each item that should be drawn or highlighted. This is similar to the owner-drawn style for menus, except that the owner-drawn style applies to the entire list rather than to individual items.

Creating a List-Box Window

List boxes are WC_LISTBOX class windows and are predefined by the system. Applications can create list boxes by calling WinCreateWindow, using WC_LISTBOX as the window-class parameter.

A list box passes notification messages to its owner window, so an application uses its client window, rather than the frame window, as the owner of the list. The client-window procedure receives the messages sent from the list box. For example, to create a list box that completely fills the client area of a frame window, an application would make the client window the owner and parent of the list-box window, and make the list-box window the same size as the client window. This is shown in the following code fragment.

```
#define ID_LISTWINDOW   250

HWND hwndClient,hwndList;
RECTL rcl;

                                          /* How big is the
                                           client window? */
WinQueryWindowRect(hwndClient, &rcl);

                                          /* Make a list-box
                                            window.       */
hwndList = WinCreateWindow(hwndClient,    /* Parent       */
    WC_LISTBOX,                           /* Class        */
    "",                                   /* Name         */
    WS_VISIBLE | LS_NOADJUSTPOS,          /* Style        */
    0, 0,                                 /* x, y         */
    rcl.xRight, rcl.yTop,                 /* cx, cy       */
    hwndClient,                           /* Owner        */
    HWND_TOP,                             /* Behind       */
    ID_LISTWINDOW,                        /* ID           */
    NULL,                                 /* Control data */
    NULL);                                /* parameters   */
```

Because the list box draws its own border, and a frame-window border already surrounds the client area of a frame window due to the adjacent frame controls, the effect is a double-thick border around the list box. You can change this effect by calling WinInflateRect to overlap the list-box border with the surrounding frame-window border, resulting in only one list-box border.

Notice that the code specifies the list-box window style LS_NOADJUSTPOS. This ensures that the list box is created exactly the specified size. If the LS_NOADJUSTPOS style is not specified, the list-box height is rounded down, if necessary, to make it a multiple of the item height. Enabling a list box to adjust its height automatically is useful for preventing partial items being displayed at the bottom of a list box.

Using a List Box in a Dialog Window

List boxes most commonly are used in dialog windows. A list box in a dialog box is a control window, like a push button or an entry field. Typically, the application defines a list box as one item in a dialog template in the resource-definition file, as shown in the following resource compiler source-code fragment.

```
DLGTEMPLATE IDD_OPEN
BEGIN
   DIALOG "Open...", IDD_OPEN, 35, 35, 150, 135,
            FS_DLGBORDER, FCF_TITLEBAR
      BEGIN
          LISTBOX        IDD_FILELIST, 15, 15, 90, 90
          PUSHBUTTON     "Drive", IDD_DRIVEBUTTON, 115, 70, 30, 14
          DEFPUSHBUTTON  "Open", IDD_OPENBUTTON, 115, 40, 30, 14
          PUSHBUTTON     "Cancel", IDD_CANCELBUTTON, 115, 15, 30, 14
      END
END
```

Once the dialog resource is defined, the application loads and displays the dialog box as it would normally. The application inserts items into the list when processing the WM_INITDLG message.

A dialog window with a list box usually has an **OK** button. The user can select items in the list, and then indicate a final selection by double-clicking, pressing Enter, or clicking the **OK** button. When the dialog-window procedure receives a message indicating that the user has clicked the **OK** button, it queries the list box to determine the current selection (or selections, if the list allows multiple selections), and then responds as though it had received a WM_CONTROL message with the LN_ENTER notification code.

Adding or Deleting an Item in a List Box

Applications can add items to a list box by sending an LM_INSERTITEM or LM_INSERTMULTITEMS message to the list-box window; items are deleted using the LM_DELETEITEM message. Items in a list are specified with a 0-based index (beginning at the top of the list). A new list is created empty; the application initializes the list by inserting items. LM_INSERTMULTITEMS allows up to 32767 items to be inserted as a group, while LM_INSERTITEM adds items one-by-one to a list.

The application specifies the text and position for each new item. It can specify an *absolute-position* index or one of the following predefined index values:

Table 21-1. List Item Position Index

Value	Meaning
LIT_END	Insert item at end of list.
LIT_SORTASCENDING	Insert item alphabetically ascending into list.
LIT_SORTDESCENDING	Insert item alphabetically descending into list.

If a large number of items are to be inserted into a list box at one time, use of LM_INSERTMULTITEMS is more efficient than use of LM_INSERTITEM. The same positioning flags are used. When LIT_SORTASCENDING or LIT_SORTDESCENDING is specified with LM_INSERTMULTITEMS, new items are inserted before the updated list is sorted. If items are being added using several LM_INSERTMULTITEMS messages, LIT_END should be specified for all messages except the last; this will avoid unnecessary multiple sorts of the list.

If no text array is specified, empty items are inserted into the list. This is very useful for list boxes created with LS_OWNERDRAW style, which do not use text strings.

The application must send an LM_DELETEITEM message and supply the absolute-position index of the item when deleting items from a list. The LM_DELETEALL message deletes all items in a list.

One way an application can speed up the insertion of list items is to suspend drawing until it has finished inserting items. This is a particularly valuable approach when using a sorted insertion process (when inserting one item can cause rearrangement of the entire list). You can turn off list drawing by calling WinEnableWindowUpdate, specifying FALSE for the *enable* parameter, and then calling WinShowWindow. This forces a total update when insertion is complete. The following code fragment illustrates this concept:

```
HWND hwndFileList;

/* Disable updates while filling the list. */
WinEnableWindowUpdate(hwndFileList, FALSE);
     .
     . /* Send LM_INSERTITEM messages to insert all new items. */
     .

/* Now cause the window to update and show the new information.  */
WinShowWindow(hwndFileList, TRUE);
```

Notice that this optimization is unnecessary if an application is adding list items while processing a WM_INITDLG message, because the list box is not visible, and the list-box routines are internally optimized.

Responding to a User Selection in a List Box

When a user chooses an item in a list, the primary notification an application receives is a WM_CONTROL message, with the LN_ENTER control code sent to the owner window of the list. Within the window procedure for the owner window, the application responds to the LN_ENTER control code by querying the list box for the current selection (or selections, in the case of an LS_MULTIPLESEL or LS_EXTENDEDSEL list box).

The LN_ENTER control code notifies the application that the user has selected a list item. A WM_CONTROL message with an LN_SELECT control code is sent to the list-box owner whenever a selection in a list changes, such as when a user moves the mouse pointer up and down a list while pressing the mouse button. In this case, items are selected but not yet *chosen*. An application can ignore LN_SELECT control codes when the selection changes, responding only when the item is actually chosen. Or an application can use LN_SELECT to display context-dependent information that changes rapidly with each selection made by the user.

Handling Multiple Selections

When a list box has the style LS_MULTIPLESEL or LS_EXTENDEDSEL, the user can select more than one item at a time. An application must use different strategies when working with these types of lists. For example, when responding to an LN_ENTER control code, it is not sufficient to send a single LM_QUERYSELECTION message, because that message will find only the first selection. To find all current selections, an application must continue sending LM_QUERYSELECTION messages, using the return index of the previous message as the starting index of the next message, until no items are returned.

Creating an Owner-Drawn List Item

To draw its own list items, an application must create a list that has the style LS_OWNERDRAW: the owner window of the list box must respond to the WM_MEASUREITEM and WM_DRAWITEM messages.

When the owner window receives a WM_MEASUREITEM message, it must return the height of the list item. All items in a list must have the same height (greater than or equal to 1). The WM_MEASUREITEM message is sent when the list box is created, and every time an item is added. You can change the item height by sending an LM_SETITEMHEIGHT message to the list-box window. The maximum width of a list box created with the LM_HORZSCROLL style can be set using an LM_SETITEMWIDTH message.

The owner window receives a WM_DRAWITEM message whenever an item in an owner-drawn list should be drawn or highlighted. Although it is quite common for an owner-drawn list to draw items, it is less common to override the system-default method of highlighting. (This method inverts the rectangle that contains the item.) Do not create your own highlighting unless, for some reason, the system-default method is unacceptable to you.

The WM_DRAWITEM message contains a pointer to an OWNERITEM data structure. The OWNERITEM structure contains the window identifier for the list box, a presentation-space handle, a bounding rectangle for the item, the position index for the item, and the application-defined item handle. This structure also contains two fields that determine whether a message draws, highlights, or removes the highlighting from an item. The OWNERITEM structure has the following form:

```
typedef struct _OWNERITEM { /* oi */
    HWND    hwnd;
    HPS     hps;
    ULONG   fsState;
    ULONG   fsAttribute;
    ULONG   fsStateOld;
    ULONG   fsAttributeOld;
    RECTL   rclItem;
    LONG    idItem;
    ULONG   hItem;
} OWNERITEM;
```

When the item must be drawn, the owner window receives a WM_DRAWITEM message with the *fsState* field set differently from the *fsStateOld* field. If the owner window draws the item in response to this message, it returns TRUE, telling the system not to draw the item. If the owner window returns FALSE, the system draws the item, using the default list-item drawing method.

You can get the text of a list item by sending an LM_QUERYITEMTEXT message to the list-box window. You should draw the item using the *hps* and *rclItem* arguments provided in the OWNERITEM structure.

If the item being drawn is currently selected, the *fsState* and *fsStateOld* fields are both TRUE; they both will be FALSE if the item is not currently selected. The window receiving a WM_DRAWITEM message can use this information to highlight the selected item at the same time it draws the item. If the owner window highlights the item, it must leave the *fsState* and *fsStateOld* fields equal to each other. If the system provides default highlighting for the item (by inverting the item rectangle), the owner window must set the *fsState* field to *1* and the *fsStateOld* field to *0* before returning from the WM_DRAWITEM message.

The owner window also receives a WM_DRAWITEM message when the highlight state of a list item changes. For example, when a user clicks an item, the highlighting must be removed from the currently selected item, and the new selection must be highlighted. If these items are owner-drawn, the owner window receives one WM_DRAWITEM message for each unhighlighted item and one message for the newly highlighted item. To highlight an item, the *fsState* field must equal TRUE, and the *fsStateOld* field must equal FALSE. In this case, the application should highlight the item and return the *fsState* and *fsStateOld* fields equal to FALSE, which tells the system not to highlight the item. The application also can return the *fsState* and *fsStateOld* fields with two different (unequal) values and the list box will highlight the item (the default action).

To remove highlighting from an item, the *fsState* field must equal FALSE and the *fsStateOld* field must equal TRUE. In this case, the application removes the highlighting and returns both the *fsState* and the *fsStateOld* fields equal to FALSE. This tells the system not to attempt to remove the highlighting. The application also can return the *fsState* and *fsStateOld* fields with two different (unequal) values, and the list box will remove the highlighting (the default response). The following code fragment shows these selection processes:

```
OWNERITEM *poi;

case WM_DRAWITEM:

    /* Convert mp2 into an OWNERITEM structure pointer.               */
    poi = (POWNERITEM) PVOIDFROMMP(mp2);

    /* Test to see if this is drawing or highlighting/unhighlighting.  */
    if (poi->fsState != poi->fsStateOld) {

        /* This is either highlighting or unhighlighting.              */
        if (poi->fsState) {
            .
            . /* Highlight the item.                                   */
            .
        }
        else {
            .
            . /* Remove the highlighting.                              */
            .
        }

        /* Set fsState = fsStateOld to tell system you did it.         */
        poi->fsState = poi->fsStateOld = 0;

        return TRUE;  /* Tells list box you did the highlighting.      */

    }
    else {
        .
        . /* Draw the item.                                            */
        .
        if (poi->fsState) {   /* Checks to see if item is selected     */
            .
            . /* Highlight the item.                                   */
            .
            /* Set fsState = fsStateOld to tell system you did it.     */
       }
       return TRUE;  /* Tells list box you did the drawing.            */
    }
```

Default List-Box Behavior

The following table lists all the messages handled by the predefined list-box window-class procedure.

Table 21-2 (Page 1 of 3). Messages Handled by WC_LISTBOX Class

Message	Description
LM_DELETEALL	Deletes all items in the list.
LM_DELETEITEM	Removes the specified item from the list, redrawing the list as necessary. Returns the number of items remaining in the list.

Table 21-2 (Page 2 of 3). Messages Handled by WC_LISTBOX Class

Message	Description
LM_INSERTITEM	Inserts a new item into the list according to the position information passed with the message.
LM_INSERTMULTITEMS	Inserts one or more items into a list box at one time.
LM_QUERYITEMCOUNT	Returns the number of items in the list.
LM_QUERYITEMHANDLE	Returns the specified item handle.
LM_QUERYITEMTEXT	Copies the text of the specified item to a buffer supplied by the message sender.
LM_QUERYITEMTEXTLENGTH	Returns the text length of the specified item.
LM_QUERYSELECTION	For a single-selection list box, returns the zero-based index of the currently selected item. For a multiple-selection list box, returns the next selected item or LIT_NONE if no more items are selected.
LM_QUERYTOPINDEX	Returns the zero-based index to the item currently visible at the top of the list.
LM_SEARCHSTRING	Searches the list for a match to the specified string.
LM_SELECTITEM	Selects the specified item. If the list is a single-selection list, deselects the previous selection. Sends a WM_CONTROL message (with the LN_SELECT code) to the owner window.
LM_SETITEMHANDLE	Sets the specified item handle.
LM_SETITEMHEIGHT	Sets the item height for the list. All items in the list have the same height.
LM_SETITEMTEXT	Sets the text for the specified item.
LM_SETITEMWIDTH	Sets the maximum width of a list box created with the LS_HORZSCROLL style.
LM_SETTOPINDEX	Shows the specified item as the top item in the list window, scrolling the list as necessary.
WM_ADJUSTWINDOWPOS	If the list box has the style LS_NOADJUSTPOS, makes no changes to the SWP structure and returns FALSE. Otherwise, adjusts the height of the list box so that a partial item is not shown at the bottom of the list. Returns TRUE if the SWP structure is changed.
WM_BUTTON2DOWN	Returns TRUE; the message is ignored.
WM_BUTTON3DOWN	Returns TRUE; the message is ignored.
WM_CHAR	Processes virtual keys for line and page scrolling. Sends an LN_ENTER notification code for the Enter key. Returns TRUE if the key is processed; otherwise, passes the message to the WinDefWindowProc function.
WM_CREATE	Creates an empty list box with a scroll bar.
WM_DESTROY	Destroys the list and deallocates any memory allocated during its existence.

Table 21-2 (Page 3 of 3). Messages Handled by WC_LISTBOX Class

Message	Description
WM_ENABLE	Enables the scroll bar if there are more items than can be displayed in a list-box window.
WM_MOUSEMOVE	Sets the mouse pointer to the arrow shape and returns TRUE to show that the message was processed.
WM_PAINT	Draws the list box and its items.
WM_HSCROLL	Handles scrolling indicated by the list-box horizontal scroll bar.
WM_VSCROLL	Handles scrolling indicated by the list-box vertical scroll bar.
WM_SETFOCUS	If the list box is gaining the focus, creates a cursor and sends an LN_SETFOCUS notification code to the owner window. If the list box is losing the focus, this message destroys the cursor and sends an LN_KILLFOCUS notification code to the owner window.
WM_TIMER	Uses timers to control automatic scrolling that occurs when a user drags the mouse pointer outside the window.

Related Functions

This section covers the functions that are related to List Box Controls.

WinDeleteLboxItem

This macro deletes the indexed item from the List Box. It returns the number of items left.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinDeleteLboxItem  (HWND hwndLbox, LONG index)
```

Parameters

hwndLbox (HWND) – input
Listbox handle.

index (LONG) – input
Index of the listbox item.

Returns

lItems (LONG) – returns
Number of items left.

WinEnableWindowUpdate

This function sets the window visibility state for subsequent drawing.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinEnableWindowUpdate  (HWND hwnd, BOOL fEnable)
```

Parameters

hwnd (HWND) – input
Window handle.

fEnable (BOOL) – input
New visibility state.

TRUE	Set window state visible
FALSE	Set window state invisible.

Returns

rc (BOOL) – returns
Visibility-changed indicator.

TRUE	Window visibility successfully changed
FALSE	Window visibility not successfully changed.

WinInflateRect

This function expands a rectangle.

Syntax

```
#define INCL_WINRECTANGLES /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinInflateRect  (HAB hab, PRECTL prcl, LONG cx, LONG cy)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prcl (PRECTL) – in/out
Rectangle to be expanded.

cx (LONG) – input
Horizontal expansion.

cy (LONG) – input
Vertical expansion.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinInsertLboxItem

This macro inserts text into a list box at index, index may be a LIT_ constant. The macro returns the actual index where it was inserted.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinInsertLboxItem  (HWND hwndLbox, LONG index, PSZ psz)
```

Parameters

hwndLbox (HWND) – input
List box handle.

index (LONG) – input
Index of the list box item.

psz (PSZ) – input
Text to be inserted.

Returns

lRetIndex (LONG) – returns
Actual index where it was inserted.

WinQueryLboxCount

This macro returns the number of items in the List Box.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
LONG WinQueryLboxCount  (HWND  hwndLbox)
```

Parameters

hwndLbox (HWND) – input

Listbox handle.

Returns

lRetNumIt (LONG) – returns

Number of items in the list box.

WinQueryLboxItemText

This macro fills the buffer with the text of the indexed item. It returns the length of the text.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG WinQueryLboxItemText  (HWND hwndLbox, SHORT index, PSZ psz,
                            SHORT cchMax)
```

Parameters

hwndLbox (HWND) – input
List box handle.

index (SHORT) – input
Index of the listbox item.

psz (PSZ) – input
Pointer to a null terminated string.

cchMax (SHORT) – input
Maximum number of characters allocated to the string.

Returns

lRetTxtL (LONG) – returns
Actual text length copied.

WinQueryLboxItemTextLength

This macro returns the length of the text of the indexed item in the List Box.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
SHORT WinQueryLboxItemTextLength  (HWND hwndLbox, SHORT index)
```

Parameters

hwndLbox (HWND) – input
Listbox handle.

index (SHORT) – input
Index of the item in the List Box.

Returns

sRetLen (SHORT) – returns
Text length of the indexed item.

WinQueryLboxSelectedItem

This macro returns the index of the selected item in the List Box (for single selection only).

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
LONG WinQueryLboxSelectedItem  (HWND hwndLbox)
```

Parameters

hwndLbox (HWND) – input
List box handle.

Returns

lRetIndex (LONG) – returns
Index of the selected item.

WinSetLboxItemText

This macro sets the text of the list box indexed item to buffer.

Syntax

```
#define INCL_WINWINDOWMGR /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinSetLboxItemText  (HWND hwndLbox, LONG index, PSZ psz)
```

Parameters

hwndLbox (HWND) – input
List box handle.

index (LONG) – input
Index of the list box item.

psz (PSZ) – input
Pointer to a null terminated string.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Messages

This section covers the messages that are related to List Box Controls.

LM_DELETEALL

This message is sent to a list box control to delete all the items in the list box.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

LM_DELETEITEM

This message deletes an item from the list box control.

Parameters

param1

sItemIndex (SHORT)
Item index.

The zero-based index of the item to be deleted.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemsLeft (SHORT)
Number remaining.

LM_INSERTITEM

This message inserts an item into a list box control.

Parameters

param1

sItemIndex (SHORT)
Item index.

LIT_END	Add the item to the end of the list.
LIT_SORTASCENDING	Insert the item into the list sorted in ascending order.
LIT_SORTDESCENDING	Insert the item into the list sorted in descending order.
Other	Insert the item into the list at the offset specified by this zero-based index.

param2

pszItemText (PSZ)
Item text.

This points to a string containing the item text.

Returns

sIndexInserted (SHORT)
Index of inserted item.

LIT_MEMERROR	The list box control cannot allocate space to insert the list item in the list.
LIT_ERROR	An error, other than LIT_MEMERROR, occurred.
Other	The zero-based index of the offset of the item within the list.

LM_INSERTMULTITEMS

This message inserts one or more items into a list box.

Parameters

param1

pListboxInfo (PLBOXINFO)
Pointer to a structure containing list box information.

param2

papszText (PSZ *)
Pointer to an array of pointers to text strings.

This parameter is a pointer to an array of pointers to zero-terminated strings. The array must contain at least *ulItemCount* items. (*ulItemCount* is a field in LBOXINFO.)

If this parameter is set to NULL, a *ulItemCount* number of empty items are inserted into the list. This is useful for ownerdraw listboxes that do not make use of text strings.

Returns

lCount (LONG)
Number of items successfully inserted into the list.

LM_QUERYITEMCOUNT

This message returns a count of the number of items in the list box control.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemCount (SHORT)
Item count.

LM_QUERYITEMHANDLE

This message returns the handle of the indexed item of the list box control.

Parameters

param1

sItemIndex (SHORT)
Item index.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulItem (ULONG)
Item handle.

0	The indexed item does not exist.
Other	Item handle.

LM_QUERYITEMTEXT

This message returns the text of the specified list box item.

Parameters

param1

sItemIndex (SHORT)
Item index.

smaxcount (SHORT)
Maximum count.

0	No text is copied.
Other	Copy the item text as a null-terminated string, but limit the number of characters copied, including the null termination character, to this value.

param2

pszItemText (PSZ)
Buffer into which the item text is to be copied.

This points to a string (character) buffer.

Returns

sTextLength (SHORT)
Length of item text.

LM_QUERYITEMTEXTLENGTH

This message returns the length of the text of the specified list box item.

Parameters

param1

sItemIndex (SHORT)
Item index.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sTextLength (SHORT)
Length of item text.

LIT_ERROR	Error occurred. For example, the item specified by its index does not exist.
Other	Length of item text.

LM_QUERYSELECTION

This message is used to enumerate the selected item, or items, in a list box.

Parameters

param1

sItemStart (SHORT)
Index of the start item.

If the list box allows multiple selected items, that is, if it has a style of LS_MULTIPLESEL, then this parameter indicates the index of the item from which the search for the next selected item is to begin. Therefore, to get all the selected items of the list, this message is sent repeatedly, each time setting this parameter to the index of the item returned by the previous usage of this message.

If this parameter is set to LIT_CURSOR the index of the item in the list box which currently has the cursor is returned.

If the list box only allows a single selection, this parameter is ignored.

LIT_CURSOR	Return the index of the item in the list box which currently has the cursor.
LIT_FIRST	Start the search at the first item.
Other	Start the search after the item specified by this index.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemSelected (SHORT)
Index of the selected item.

LIT_NONE No selected item.

For a single selection list box, this implies that there is no selected item in the list box. For a multiple selection list box, this implies that there is no selected item in the list box whose index is higher than the index specified by the *sItemStart* parameter.

Other Index of selected item. For a single selection list box, this is the index of the only selected item in the list box. For a multiple selection list box, this is the index of the next selected item in the list box whose index is higher than the index specified by the *sItemStart* parameter.

If *sItemStart* is set to LIT_CURSOR, the index of the list-box item which currently has the cursor is returned.

LM_QUERYTOPINDEX

This message obtains the index of the item currently at the top of the list box.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

sItemTop (SHORT)
Index of the item currently at the top of the list box.

LIT_NONE	No items in the list box
Other	Index of the item currently at the top of the list box.

LM_SEARCHSTRING

This message returns the index of the list box item whose text matches the string.

Parameters

param1

uscmd (USHORT)
Command.

Defines the criteria by which the string specified by the *pszSearchString* parameter is to be compared with the text of the items, to determine the index of the first matching item.

These values can be combined using the logical-OR operator:

LSS_CASESENSITIVE	Matching occurs if the item contains the characters specified by the *pszSearchString* parameter exactly. *This value is mandatory.*
LSS_PREFIX	Matching occurs if the leading characters of the item contain the characters specified by the *pszSearchString* parameter. If this value is specified, LSS_SUBSTRING must not be specified.
LSS_SUBSTRING	Matching occurs if the item contains a substring of the characters specified by the *pszSearchString* parameter. If this value is specified, LSS_PREFIX must not be specified.

sItemStart (SHORT)
Index of the start item.

LIT_FIRST	Start the search at the first item.
Other	Start the search after the item specified by this index.

param2

pszSearchString (PSZ)
Search string.

This points to the string to search for.

Returns

sItemMatched (SHORT)
Index item whose text matches the string.

LIT_ERROR	Error occurred
LIT_NONE	No item found

Other Index item whose text matches the string.

LM_SELECTITEM

This message is used to set the selection state of an item in a list box.

Parameters

param1

sItemIndex (SHORT)
Index of the item to be selected or deselected:

LIT_NONE	All items are to be deselected
Other	Index of the item to be selected or deselected.

param2

usselect (USHORT)
Select flag.

(Ignored if *sItemIndex* is set to LIT_NONE).

TRUE	The item is selected. If the control is a single selection list box (that is, it does not have the style of LS_MULTIPLESEL), any previously selected item is deselected.
FALSE	The item is deselected.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred. For example, when the item does not exist in the list box, or when an item that is not selected is deselected.

LM_SETITEMHANDLE

This message sets the handle of the specified list box item.

Parameters

param1

sItemIndex (SHORT)
Item index.

param2

ulItemHandle (ULONG)
Item handle.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

LM_SETITEMHEIGHT

This message sets the height of the items in a list box.

Parameters

param1

flNewHeight (ULONG)
Height of items in list box.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful operation
FALSE	Error occurred.

LM_SETITEMTEXT

This message sets the text into the specified list box item.

Parameters

param1

sItemIndex (SHORT)
Item index.

param2

pszItemText (PSZ)
Item text.

This points to a string containing the text to set the list-box item to.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

LM_SETITEMWIDTH

This message sets the width of the items in a list box.

Parameters

param1

lNewWidth (ULONG)
Width of items in list box.

param2

reserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

LM_SETTOPINDEX

This message is used to scroll a particular item to the top of the list box.

Parameters

param1

sItemIndex (SHORT)
Index of the item to be made top.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

rc (BOOL)
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WM_CONTROL (in List Boxes)

For the cause of this message, see "WM_CONTROL" on page 16-5.

Parameters

param1

id (USHORT)
Control-window identity.

usnotifycode (USHORT)
Notify code.

The list box control window procedure uses these notification codes:

LN_ENTER	Either the Enter or Return key has been pressed while the list box control has the focus, or the list box control has been double-clicked.
LN_KILLFOCUS	The list box control loses the focus.
LN_SCROLL	The list box control is about to scroll horizontally. This can happen when the application has issued a WinScrollWindow function.
LN_SETFOCUS	The list box control receives the focus.
LN_SELECT	An item is being selected (or deselected). **Note:** To discover the index of the selected item, the application must use the LM_QUERYSELECTION message.

param2

hwndcontrolspec (HWND)
List box control window handle.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_DRAWITEM (in List Boxes)

This notification is sent to the owner of a list box control each time an item is to be drawn.

Parameters

param1

idListBox (USHORT)
Window identifier.

The window identity of the list box control sending this notification message.

param2

pOwnerItem (POWNERITEM)
Owner-item structure.

This points to an owner-item structure; see "OWNERITEM" on page 21-41.

Returns

rc (BOOL)
Item-drawn indicator.

TRUE	The owner draws the item, so the list box control does not draw it.
FALSE	If the item contains text and the owner does not draw the item, the owner returns this value, and the list box control draws the item.

WM_MEASUREITEM (in List Boxes)

This notification is sent to the owner of a list box control to establish the height and width for an item in that control.

Parameters

param1

sListBox (SHORT)
List-box identifier.

param2

sItemIndex (SHORT)
Item index.

The zero-based index of the item which has changed.

Returns

ReturnCode

sHeight (SHORT)
Height of item.

sWidth (SHORT)
Width of item.

This value is required only if the list box control is scrollable horizontally, that is, it has a style of LS_HORZSCROLL.

Related Data Structures

This section covers the data structures that are related to List Box Controls.

LBOXINFO

List box information structure.

Syntax

```
typedef struct _LBOXINFO {
LONG        lItemIndex;
ULONG       ulItemCount;
ULONG       reserved;
ULONG       reserved2;
 } LBOXINFO;

typedef LBOXINFO *PLBOXINFO;
```

Fields

lItemIndex (LONG)

Index of the item to insert after.

Possible values are described in the following list:

LIT_ENT	Add items to the end of the list.
LIT_SORTASCENDING	Add items to the list and sort the complete list in ascending order.
LIT_SORTDESCENDING	Add items to the list and sort the complete list in descending order.
Other	Add the items to the list after the specified zero-based index. Valid range is 0 to 32 767.

ulItemCount (ULONG)

Number of items to be inserted into the list.

A maximum of 32 768 can be inserted into the list at one time.

reserved (ULONG)

Reserved value, must be 0.

reserved2 (ULONG)

Reserved value, must be 0.

OWNERITEM

Owner item.

Syntax

```
typedef struct _OWNERITEM {
HWND        hwnd;
HPS         hps;
ULONG       fsState;
ULONG       fsAttribute;
ULONG       fsStateOld;
ULONG       fsAttributeOld;
RECTL       rclItem;
LONG        idItem;
ULONG       hItem;
 } OWNERITEM;

typedef OWNERITEM *POWNERITEM;
```

Fields

hwnd (HWND)
Window handle.

hps (HPS)
Presentation-space handle.

fsState (ULONG)
State.

fsAttribute (ULONG)
Attribute.

fsStateOld (ULONG)
Old state.

fsAttributeOld (ULONG)
Old attribute.

rclItem (RECTL)
Item rectangle.

idItem (LONG)
Item identity.

hItem (ULONG)
Item.

Summary

Following are the functions, messages, and data structures used with list boxes.

Table 21-3. List-Box Functions

Function Name	Description
WinDeleteLboxItem	Deletes the indexed item from the list box. Returns the number of items left.
WinEnableWindowUpdate	Sets the window visibility state for subsequent drawing.
WinInflateRect	Expands a rectangle.
WinInsertLboxItem	Inserts text into a list box at index. Returns the actual index where it was inserted.
WinQueryLboxCount	Returns the number of items in the list box.
WinQueryLboxItemText	Fills the buffer with the text of the indexed item. Returns the length of the text.
WinQueryLboxItemTextLength	Returns the length of the text of the indexed item in the list box.
WinQueryLboxSelectedItem	Returns the index of the selected item in the list box. For single selection only.
WinSetLboxItemText	Sets the text of the list-box indexed item to buffer.

Table 21-4. Messages Generated by a List Box

Message	Description
WM_CONTROL	Occurs when a list-box control has a significant event to notify to its owner.
WM_DRAWITEM	Notification sent to the owner of a list-box control each time an item is to be drawn.
WM_MEASUREITEM	Notification sent to the owner of a specific list-box control to establish the height and width of an item in that control.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the list-box control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the list-box control window parameters.

Table 21-5. Messages Received by a List Box

Message	Description
LM_DELETEALL	Sent to a list-box control to delete all the items in the list box.
LM_DELETEITEM	Deletes an item from the list-box control.
LM_INSERTITEM	Inserts an item into a list-box control.
LM_INSERTMULTITEMS	Inserts one or more items into a list box at one time.
LM_QUERYITEMCOUNT	Returns a count of the number of items in the list-box control.
LM_QUERYITEMHANDLE	Returns the handle of the indexed item of the list-box control.
LM_QUERYITEMTEXT	Returns the text of the specified list-box item.
LM_QUERYITEMTEXTLENGTH	Returns the length of the text of the specified list-box item.
LM_QUERYSELECTION	Used to enumerate the selected item, or items, in a list box.
LM_QUERYTOPINDEX	Obtains the index of the item currently at the top of the list box.
LM_SEARCHSTRING	Returns the index of the list-box item whose text matches the string.
LM_SELECTITEM	Used to set the selection state of an item in a list box.
LM_SETITEMHANDLE	Sets the handle of the specified list-box item.
LM_SETITEMHEIGHT	Sets the height of the items in a list box.
LM_SETITEMTEXT	Sets the text into the specified list-box item.
LM_SETITEMWIDTH	Sets the maximum width of a list box created with the LS_HORZSCROLL style.
LM_SETTOPINDEX	Used to scroll a particular item to the top of the list box.

Table 21-6. List-Box Data Structures

Structure Name	Description
OWNERITEM	Owner item.
LBOXINFO	List-box information structure.

Chapter 22. Clipboards

The *clipboard* is a small amount of system-managed random-access memory (RAM) used for *user-driven* data exchange. This is in contrast with dynamic data exchange (DDE), which is application driven. While the clipboard stores only *pointers* or *handles* to data, its associated set of functions can be used in applications to move and exchange data. This chapter describes how to use the clipboard in PM applications.

About the Clipboard

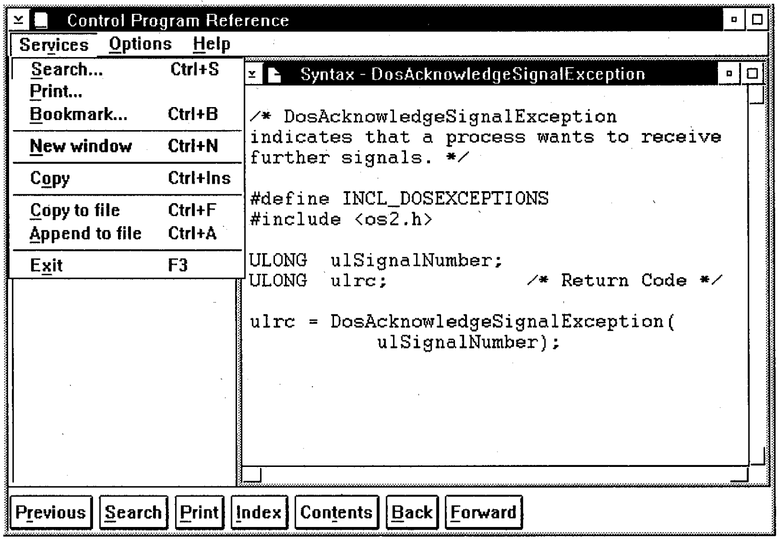
The clipboard enables the user to move data in a single application or exchange data between applications. Typically, a user selects data in the application using the mouse or keyboard, and then initiates a cut, copy, or paste operation on that selection.

Descriptions of these operations are in the following table:

Table 22-1. Operations on Clipboard Data

Operation	Description
Cut	Deletes the selected data from the application and copies it to the clipboard. Any previous contents of the clipboard are destroyed.
Copy	Copies the selected data to the clipboard. The selection remains unchanged. Previous contents of the clipboard are destroyed.
Paste	Deletes the selected data from the application and replaces it with the contents of the clipboard. The contents of the clipboard are not changed.

Figure 22-1 on page 22-2 is an example of copying data from one application, and Figure 22-2 on page 22-2 illustrates pasting that same data into another application by way of the clipboard.



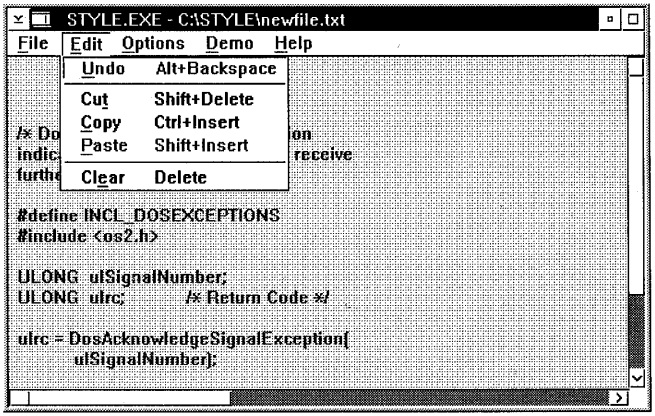
```
/* DosAcknowledgeSignalException
indicates that a process wants to receive
further signals. */

#define INCL_DOSEXCEPTIONS
#include <os2.h>

ULONG  ulSignalNumber;
ULONG  ulrc;           /* Return Code */

ulrc = DosAcknowledgeSignalException(
           ulSignalNumber);
```

Figure 22-1. A Copy Operation Between Applications Using the Clipboard



```
/* Do                              on
indic                              receive
furth

#define INCL_DOSEXCEPTIONS
#include <os2.h>

ULONG  ulSignalNumber;
ULONG  ulrc;         /* Return Code */

ulrc = DosAcknowledgeSignalException(
        ulSignalNumber);
```

Figure 22-2. A Paste Operation Between Applications Using the Clipboard

An application should not perform any clipboard operations unless the user initiates them explicitly. Other OS/2 features, such as pipes, queues, shared memory, and especially DDE should be used when data exchange is needed without user involvement. For example, an application that continuously passes remotely collected data to a data-analysis application must not use the clipboard. Such an application, instead, should use the other interprocess data-communication capabilities of the operating system.

The data on the clipboard is maintained in memory only. Clipboard data is lost when the computer is turned off.

Shared Memory and the Clipboard

An application must store, in shared memory, text data that is destined for the clipboard. To do so, the application calls the DosAllocSharedMem function with the OBJ_GIVEABLE attribute to allocate a shared memory object, and then copies the text data to the object. The application passes the clipboard a pointer, which the clipboard uses to access the shared memory object. Clipboard functions use the CFI_POINTER flag to indicate text data stored in a shared memory object.

To pass a bit map or metafile to the clipboard, an application passes the clipboard a bit map or metafile handle. The clipboard functions make the bit map or metafile *shareable*. The CFI_HANDLE flag is used in clipboard functions to indicate bit map or metafile data.

After closing the clipboard, an application no longer can access the data it passed to the clipboard. Likewise, when an application requests data from the clipboard, it receives a pointer or handle that is good only until the application closes the clipboard. Typically, the application either uses the data immediately before closing the clipboard, or it copies the data to local memory for future use, then closes the clipboard.

Clipboard Operations

An application uses the clipboard when cutting, copying, or pasting data. Typically, an application places data on the clipboard for cut and copy operations and removes data from the clipboard for paste operations. The following paragraphs describe all these clipboard operations.

Cut and Copy Operations

To put data on the clipboard, an application first calls the WinOpenClipbrd function to verify that other applications are not trying to retrieve or set clipboard data. The WinOpenClipbrd function does not return if another thread has the clipboard open; it waits until either the clipboard is free or there is a message in the message queue of the calling thread. In practice, the WinOpenClipbrd function waits until the clipboard is available or until the calling application responds to a message. If the clipboard cannot be opened before a message arrives, the application receives the message, and the WinOpenClipbrd function continues to try to open the clipboard. The WinOpenClipbrd function does not return until the clipboard is open. However, the application continues to receive messages.

Once an application successfully opens the clipboard, it must remove any previously stored data on the clipboard by calling the WinEmptyClipbrd function. If the clipboard is not cleared, writing an existing format on the clipboard replaces the old data in that format with the new data. Old data in other formats remains on the clipboard.

After emptying the clipboard, an application should write its data to the clipboard in as many standard formats as possible. For each format, the application passes the data to the clipboard by calling the WinSetClipbrdData function, specifying each data format. The clipboard is not cleared when a new format is written to it; all new data formats coexist on the clipboard until it is cleared by the next clipboard user.

If an application passes NULL as the *ulData* parameter of the WinSetClipbrdData function, applications must render the data on request.

Finally, when an application finishes writing the clipboard data, it must release the clipboard by calling the WinCloseClipbrd function so that other applications can use the clipboard.

Paste Operation

To retrieve data from the clipboard, an application first must call the WinOpenClipbrd function to verify that no other applications are trying to retrieve or set the clipboard data.

Once an application successfully opens the clipboard, it calls the WinQueryClipbrdData function, specifying a preferred format. If that format is not available (indicated by a NULL return from the WinQueryClipbrdData function) the application should continue to call WinQueryClipbrdData for other possible formats until it either receives the data or runs out of format choices.

If the clipboard contains one of the requested formats, the WinQueryClipbrdData function returns a 32-bit integer, the meaning of which depends on the particular format. For text data, the return value is a pointer to a shareable memory object containing the text. For bit map data, the return value is a bit map handle. For metafile data, the return value is a metafile handle.

It is important that an application use the WinCloseClipbrd function to close the clipboard as soon as possible so that other applications can access it.

Standard Clipboard-Data Formats

The clipboard can accept data in three standard formats: text, bit map, and metafile. Applications can either use these formats or create their own private formats.

All PM applications can access the clipboard, so applications can copy to the clipboard the same selection of data in many different formats. For example, a word processor that supports multiple fonts might write the same selection of text to the clipboard in three different formats: straight text, *rich* text, and metafile. Then, another application (pasting from the clipboard) could choose the appropriate format.

Applications can use the following constants to specify the standard clipboard-data formats:

Table 22-2 (Page 1 of 2). Clipboard Data Formats

Format	Description
CF_BITMAP	Specifies that the data in the clipboard is a bit map.
CF_DSPBITMAP	Specifies that the data in the clipboard is a bit map representation of a private-data format. The clipboard viewer uses this format to display a private format.
CF_DSPMETAFILE	Specifies that the data in the clipboard is a metafile representation of a private-data format. The clipboard viewer uses this format to display a private format.

Table 22-2 (Page 2 of 2). Clipboard Data Formats

Format	Description
CF_DSPTEXT	Specifies that the data in the clipboard is a text representation of a private-data format. The clipboard viewer uses this format to display a private format.
CF_METAFILE	Specifies that the data in the clipboard is a metafile.
CF_TEXT	Specifies that the data in the clipboard is an array of text characters. These characters can include *newline* characters to indicate line breaks. The NULL character indicates the end of the text data.

Private Clipboard-Data Formats

Applications that use the clipboard to move data within the documents of the application can use private clipboard-data formats when standard formats are insufficient for representing clipboard data. For example, a word processor might have a rich-text format that contains font and style information in addition to the usual text characters. Clearly, if the word processor uses the clipboard to support cut, copy, and paste operations for moving data in its documents, a standard text format will be inadequate.

In such case, the word processor should write at least two formats to the clipboard for each cut or copy operation: a standard text format representing the text of the current selection and a private rich-text format representing the true state of the selection. If the word processor performs a paste operation by using clipboard data, it can use the rich-text format to retain all formatting. If another application requests the same data, it can use the standard-text format if it does not recognize the private format. Also, the word processor should be able to render data in CF_BITMAP and CF_METAFILE formats for painting and drawing applications.

Format Identification Number

Each private format must have a unique identification number. To obtain an identification number, the application registers the name of the private format in the system atom table. The system assigns a unique identification number for the format name. Other applications having access to the format name can query the system atom table for the format identification number.

An application can interpret its own private formats and request them from the clipboard for cutting and pasting its own data. Other applications that know the private-format identification number also can interpret the formatted data.

Display Formats

The OS/2 operating system provides three standard display formats for applications that use private formats: CF_DSPTEXT, CF_DSPBITMAP, and CF_DSPMETAFILE. These formats correspond to the standard text, bit map, and metafile formats, with the exception that they are intended for use only by the clipboard viewer. An application that uses a private format should write one of the DSP formats that approximates the appearance of the private data so that the clipboard viewer can display the data regardless of the format. For example, a word processor using the rich-text format also would write a CF_DSPBITMAP formatted picture of the selected text that contains all the type fonts and styles.

Notice that you can choose delayed rendering for DSP formats because there might not always be a clipboard viewer active on the screen. With delayed rendering, an application actually does not render the format unless it is requested to do so.

Delayed Rendering

An application can pass NULL as the *ulData* parameter of the WinSetClipbrdData function instead of a pointer or a handle. This indicates that the data is rendered only when another application requests it from the clipboard. This is useful if an application supports several clipboard formats that are time-consuming to render. With delayed rendering, an application can send NULL handles for each clipboard format that it supports and render individual formats only when the format actually is requested from the clipboard. An application can either write data for standard formats or choose delayed rendering for more complex formats.

When an application uses delayed rendering for one or more of its clipboard formats, it must become the clipboard owner. As long as the application is the clipboard owner, it receives a WM_RENDERFMT message whenever a request is received by the clipboard for a format using delayed rendering. When the application receives such a message, it renders the data and passes the pointer or handle to the clipboard by calling the WinSetClipbrdData function. The rules for shared-memory access for rendered data are the same as those for standard clipboard data. This simply is a delayed execution of the operation that occurs if the data does not have delayed rendering.

The clipboard owner, with one or more delayed-rendering formats on the clipboard, receives a WM_RENDERALLFMTS message just before the clipboard-owner application terminates. This ensures that the application renders all of its data before terminating.

Clipboard Viewer

A window can become a clipboard viewer and display the current contents of the clipboard. The clipboard viewer is informed whenever the clipboard contents change. Typically, the clipboard viewer is a window that can draw the standard clipboard formats. The clipboard viewer is a convenience for the user; it does not have any effect on the data-transaction functions of the clipboard.

To create a clipboard viewer, an application calls WinSetClipbrdViewer, specifying the window in which the clipboard data will be displayed. Usually this is the client window of an application. There can be only one clipboard viewer at any time in the system, so setting a clipboard viewer replaces any previous clipboard viewer. The WinQueryClipbrdViewer function receives the handle to the current clipboard viewer so that the application can reset it when finished with the clipboard viewer.

Once a window becomes the clipboard viewer, it receives WM_DRAWCLIPBOARD messages whenever the contents of the clipboard change. The window should respond to these messages by drawing the contents of the clipboard.

The clipboard viewer displays all the standard formats and should process CFI_OWNERDISPLAY items by sending the appropriate message to the clipboard owner.

The clipboard viewer cannot display private-format data. For this reason, an application that writes private-format data to the clipboard also must write the data in one of the three standard-display formats: CF_DSPTEXT, CF_DSPBITMAP, or CF_DSPMETAFILE.

If a standard format is not provided in addition to the private formats, the clipboard owner must draw the clipboard data in the clipboard-viewer window. An application uses the CFI_OWNERDRAW flag to identify clipboard data that the clipboard owner draws. When the clipboard viewer encounters data with the CFI_OWNERDRAW flag set, it sends WM_PAINTCLIPBOARD messages to the clipboard owner whenever the data must be drawn, scrolled, or sized.

The clipboard viewer determines the attributes of a particular clipboard format by calling the WinQueryClipbrdFmtInfo function. The identity of the current owner is found by calling the WinQueryClipbrdOwner function.

Clipboard Owner

The *clipboard owner* is any application window connected to the clipboard data. Following are situations in which an application would call WinSetClipbrdOwner to become the clipboard owner:

- The application calling WinSetClipbrdData passes a NULL pointer or handle to the clipboard, indicating that the application renders the data in a particular format on request. As a result, the system sends rendering requests to the current clipboard owner.
- The application calling WinSetClipbrdData passes data with the CFI_OWNERFREE attribute, indicating that the application frees memory for data when the clipboard is emptied. As a result, the system sends owner-free requests to the current clipboard owner.
- The application calling WinSetClipbrdData passes data with the CFI_OWNERDISPLAY attribute, indicating that the owner application draws the data in the clipboard viewer. As a result, the clipboard viewer sends drawing-related requests to the current clipboard owner.

The window specified in the call to the WinSetClipbrdOwner function responds to the following messages:

Table 22-3 (Page 1 of 2). Messages Handled by Clipboard Owner

Message	**Description**
WM_RENDERFMT	Sent by the system to the clipboard owner when a particular format with delayed rendering must be rendered. The receiver must render the data in the specified format and pass it to the clipboard by calling the WinSetClipbrdData function.

Table 22-3 (Page 2 of 2). Messages Handled by Clipboard Owner

Message	Description
WM_RENDERALLFMTS	Sent by the system to the clipboard owner just before the owner application terminates. The receiver must render the clipboard data in all formats on the clipboard with delayed rendering. It must pass the data for each format to the clipboard by calling the WinSetClipbrdData function.
WM_DESTROYCLIPBOARD	Sent by the system to the clipboard owner when the clipboard is cleared by another application calling the WinEmptyClipbrd function. The receiver must free the memory occupied by any clipboard formats using the CFI_OWNERFREE attribute.
WM_SIZECLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains the data handle with the CFI_OWNERDISPLAY attribute and when the clipboard-viewer changes size. When the clipboard viewer is being destroyed or reduced to an icon, this message is sent with the coordinates of the opposite corners set to (0,0), which permits the owner to free its display resources.
WM_VSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when an event occurs in the clipboard-viewer scroll bars. The receiver must respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the slider position.
WM_HSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when an event occurs in the scroll bars of the clipboard viewer. The receiver must respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the slider position.
WM_PAINTCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when the clipboard-viewer client area needs repainting. The receiver must respond to this message by painting the requested format (by calling WinGetPS for the window handle of the clipboard viewer).

An application automatically loses ownership of the clipboard when the clipboard data is cleared by the WinEmptyClipbrd function. Ownership is necessary only when data is present on the clipboard. Typically, an application loses ownership when another application places data on the clipboard.

Using the Clipboard

You can use the clipboard functions to perform the following tasks:

- Put data on the clipboard.
- Retrieve data from the clipboard.
- View data on the clipboard.

Putting Data on the Clipboard

The following code fragment shows how an application places text data on the clipboard, how it opens the clipboard, copies the text to a shared memory object, empties the clipboard, and passes the pointer to the clipboard:

```
#define MAXSTR   1024

PSZ  pszSrc, pszDest;
BOOL fSuccess;
CHAR szClipString[MAXSTR];
HAB  hab;

    .
    . /* Get character string (szClipString). */
    .

if (WinOpenClipbrd(hab)) {

    /* Allocate a shared memory object for the text data. */
    if (!(fSuccess = DosAllocSharedMem(
            (PVOID)&pszDest,       /* Pointer to shared memory object */
            NULL,                  /* Use unnamed shared memory       */
            strlen(szClipString)+1,/* Amount of memory to allocate    */
            PAG_WRITE  |           /* Allow write access              */
            PAG_COMMIT |           /* Commit the shared memory        */
            OBJ_GIVEABLE))) {      /* Make pointer giveable           */

        /* Set up the source pointer to point to text. */
        pszSrc = szClipString;

        /* Copy the string to the allocated memory. */
        while (*pszDest++ = *pszSrc++);

        /* Clear old data from the clipboard. */
        WinEmptyClipbrd(hab);

        /*
         * Pass the pointer to the clipboard in CF_TEXT format. Notice
         * that the pointer must be a ULONG value.
         */

        fSuccess = WinSetClipbrdData(hab, /* Anchor-block handle    */
            (ULONG) pszDest,              /* Pointer to text data   */
            CF_TEXT,                      /* Data is in text format */
            CFI_POINTER);                 /* Passing a pointer      */

        /* Close the clipboard. */
        WinCloseClipbrd(hab);
    }
}
```

Retrieving Data from the Clipboard

The following code fragment shows how to open the clipboard, retrieve data in the requested format, copy the data to local memory, and close the clipboard:

```
PSZ pszClipText, pszLocalText;

if (WinOpenClipbrd(hab)) {
    if (pszClipText = (PSZ) WinQueryClipbrdData(hab, CF_TEXT)) {

        /* Copy text from the shared memory object to local memory. */
        while (*pszLocalText++ = *pszClipText++);
    }
    WinCloseClipbrd(hab);
}
```

Viewing Data on the Clipboard

The following code fragment shows how a sample clipboard viewer responds to the WM_DRAWCLIPBOARD message, drawing text and bit map data in its window. Notice that the code uses the data retrieved from the clipboard before closing the clipboard. An alternative strategy is to copy the data and then close the clipboard. In any case, the original data from the clipboard cannot be used after the clipboard is closed.

```
PSZ     pszText;
HPS     hps;
RECTL   rcl;
HBITMAP hBitmap;
POINTL  ptlDest;

case WM_DRAWCLIPBOARD:
    if (!WinOpenClipbrd(hab))
        return 0;

    hps = WinGetPS(hwnd);  /* Get a presentation space for drawing */
    WinQueryWindowRect(hwnd, &rcl);/* Get dimensions of the window */

    if (pszText =(PSZ)WinQueryClipbrdData(hab, CF_TEXT)) {
        WinDrawText(hps,
            -1,                          /* Null-terminated string  */
            pszText,                     /* The string              */
            &rcl,                        /* Where to put the string */
            CLR_BLACK,                   /* Foreground color        */
            CLR_WHITE,                   /* Background color        */
            DT_CENTER | DT_VCENTER | DT_ERASERECT);
    }
    else if (hBitmap = (HBITMAP)WinQueryClipbrdData(hab, CF_BITMAP)) {
        ptlDest.x = ptlDest.y = 0;
        WinFillRect(hps, &rcl, CLR_WHITE);
        WinDrawBitmap(hps,
            hBitmap,
            NULL,                        /* Draws entire bit map    */
            &ptlDest,                    /* Destination             */
            CLR_BLACK,                   /* Foreground color        */
            CLR_WHITE,                   /* Background color        */
            DBM_NORMAL);                 /* Bit map flags           */
    }

    /* Remove rectangle from the update region */
    WinValidateRect(hwnd, &rcl, FALSE);
    WinReleasePS(hps);            /* Release the presentation space.*/
    WinCloseClipbrd(hab);         /* Close the clipboard.           */
    return 0;
```

Figure 22-3. Responding to WM_DRAWCLIPBOARD Message

Related Functions

This section covers the functions that are related to Clipboards.

WinCloseClipbrd

This function closes the clipboard, allowing other applications to open it with the WinOpenClipbrd function.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinCloseClipbrd  (HAB  hab)
```

Parameters

hab (HAB) – input

Anchor-block handle.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinEmptyClipbrd

This function empties the clipboard, removing and freeing all handles to data that is in the clipboard.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinEmptyClipbrd  (HAB  hab)
```

Parameters

hab (HAB) – input

Anchor-block handle.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinEnumClipbrdFmts

This function enumerates the list of clipboard data formats available in the clipboard.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG WinEnumClipbrdFmts  (HAB hab, ULONG fmt)
```

Parameters

hab (HAB) – input
Anchor-block handle.

fmt (ULONG) – input
Previous clipboard-data format index.

Returns

ulNext (ULONG) – returns
Next clipboard-data format index.

0 Enumeration is complete; that is, there are no more clipboard formats available.
Other Index of the next available clipboard-data format in the clipboard.

WinOpenClipbrd

This function opens the clipboard.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinOpenClipbrd (HAB hab)
```

Parameters

hab (HAB) – input
Anchor-block handle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Clipboard successfully opened
FALSE	Error occurred.

WinQueryClipbrdData

This function obtains a handle to the current clipboard data with a specified format.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
ULONG WinQueryClipbrdData  (HAB hab, ULONG fmt)
```

Parameters

hab (HAB) – input
Anchor-block handle.

fmt (ULONG) – input
Format of the data to be accessed.

CF_TEXT	Text format. Each line ends with a carriage-return/line-feed combination. Tab characters separate fields within a line. A NULL character signals the end of the data.
CF_DSPTEXT	Text display format associated with private format.
CF_BITMAP	Bit map.
CF_DSPBITMAP	Bit-map display format associated with private format.
CF_METAFILE	Metafile.
CF_DSPMETAFILE	Metafile display format associated with private format.
CF_PALETTE	Palette.

Returns

ulRet (ULONG) – returns
Handle to the clipboard data.

0	Format does not exist, or an error occurred
Other	Handle to the clipboard data.

WinQueryClipbrdFmtInfo

This function determines whether a particular format of data is present in the clipboard, and if so, provides information about that format.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL WinQueryClipbrdFmtInfo  (HAB hab, ULONG fmt, PULONG prgfFmtInfo)
```

Parameters

hab (HAB) – input
Anchor-block handle.

fmt (ULONG) – input
Format of the data to be queried.

CF_TEXT	Text format. Each line ends with a carriage-return/line-feed combination. Tab characters separate fields within a line. A NULL character signals the end of the data.
CF_DSPTEXT	Text display format associated with private format.
CF_BITMAP	Bit map.
CF_DSPBITMAP	Bit-map display format associated with private format.
CF_METAFILE	Metafile.
CF_DSPMETAFILE	Metafile display format associated with private format.
CF_PALETTE	Palette.

prgfFmtInfo (PULONG) – output
Memory model and usage flags.

Returns

rc (BOOL) – returns
Format-exists indicator.

TRUE	*fmt* exists in the clipboard and *prgfFmtInfo* is set
FALSE	*fmt* does not exist in the clipboard and *prgfFmtInfo* is not set.

WinQueryClipbrdOwner

This function obtains any current clipboard owner window.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HWND WinQueryClipbrdOwner  (HAB hab)
```

Parameters

hab (HAB) – input

Anchor-block handle.

Returns

hwndClipbrdOwner (HWND) – returns

Window handle of the current clipboard owner.

NULLHANDLE	If the clipboard is not owned by any window, or if an error occurred.
Other	Window handle of the current clipboard owner.

WinQueryClipbrdViewer

This function obtains any current clipboard viewer window.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

HWND WinQueryClipbrdViewer  (HAB hab)
```

Parameters

hab (HAB) – input

Anchor-block handle.

Returns

hwndClipbrdViewer (HWND) – returns

Current clipboard viewer window handle.

NULLHANDLE Clipboard does not have a current viewer window, or an error occurred

Other Current clipboard viewer window handle.

WinSetClipbrdData

This call puts data into the clipboard.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinSetClipbrdData  (HAB hab, ULONG ulh, ULONG ulfmt,
                         ULONG flFmtInfo)
```

Parameters

hab (HAB) – input
Anchor-block handle.

ulh (ULONG) – input
General handle to the data object being set into the clipboard.

ulfmt (ULONG) – input
Clipboard format of the data object referenced by *ulh*.

CF_TEXT	Text format. Each line ends with a carriage-return/line-feed combination. Tab characters separate fields within a line. A NULL character signals the end of the data.
CF_DSPTEXT	Text display format associated with private format.
CF_BITMAP	Bit map.
CF_DSPBITMAP	Bit-map display format associated with private format.
CF_METAFILE	Metafile.
CF_DSPMETAFILE	Metafile display format associated with private format.
CF_PALETTE	Palette.

flFmtInfo (ULONG) – input
Information.

Returns

rc (BOOL) – returns
Data-placed indicator.

TRUE	Data placed into clipboard.
FALSE	Data is not placed into clipboard, either an error occurred, or *ulh* is NULL.

WinSetClipbrdOwner

This function sets the current clipboard-owner window.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetClipbrdOwner  (HAB hab, HWND hwnd)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwnd (HWND) – input
Window handle of the new clipboard owner.

NULLHANDLE	Clipboard-owner window is released and no new clipboard-owner window is established.
Other	Window handle of the new clipboard owner.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

WinSetClipbrdViewer

This function sets the current clipboard-viewer window to a specified window.

Syntax

```
#define INCL_WINCLIPBOARD /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL WinSetClipbrdViewer  (HAB hab, HWND hwndNewClipViewer)
```

Parameters

hab (HAB) – input

Anchor-block handle.

hwndNewClipViewer (HWND) – input

Window handle of the new clipboard viewer.

NULLHANDLE The clipboard-viewer window is released and no new clipboard-viewer window is established.

Other Window handle of the new clipboard viewer.

Returns

rc (BOOL) – returns

Success indicator.

TRUE Valid, new clipboard-viewer window established
FALSE There is no new clipboard-viewer window established.

Related Messages

This section covers the messages that are related to Clipboards.

WM_DESTROYCLIPBOARD

This message is sent to the clipboard owner when the clipboard is emptied through a call to WinEmptyClipbrd.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_DRAWCLIPBOARD

This message is sent to the clipboard viewer window whenever the contents of the clipboard change; that is, as a result of the WinCloseClipbrd function following a call to WinSetClipbrdData.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_HSCROLLCLIPBOARD

This message is sent to the clipboard-owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and there is an event in the clipboard viewer's horizontal scroll bar.

Parameters

param1

hwndViewer (HWND)
Handle.

This contains a handle to the clipboard application window.

param2

sposScroll (SHORT)
Scroll position.

The position is either:

0	*scodeScroll* is other than SB_SLIDERPOSITION
Other	The position of the slider when *scodeScroll* is SB_SLIDERPOSITION.

scodeScroll (SHORT)
Scroll-bar code.

This is one of the SB_* scroll-bar codes as defined in WM_HSCROLL (in Horizontal Scroll Bars).

SB_LINELEFT	Sent if the operator clicks the left arrow of the scroll bar, or presses the VK_LEFT key.
SB_LINERIGHT	Sent if the operator clicks the right arrow of the scroll bar, or presses the VK_RIGHT key.
SB_PAGELEFT	Sent if the operator clicks the area to the left of the slider, or presses the VK_PAGELEFT key.
SB_PAGERIGHT	Sent if the operator clicks the area to the right of the slider, or presses the VK_PAGERIGHT key.
SB_SLIDERPOSITION	Sent to indicate the final position of the slider. *sposScroll* contains the final position of the slider.
SB_SLIDERTRACK	Sent every time the slider position changes if the operator moves the scroll bar slider with the pointer device.
SB_ENDSCROLL	Sent when the operator has finished scrolling, but only if the operator has not been doing any absolute slider positioning.

Returns

ulReserved (ULONG)

Reserved value, should be 0.

WM_PAINTCLIPBOARD

This message is sent when the clipboard contains a data handle with the CFI_OWNERDISPLAY information flag set.

Parameters

param1

hwndViewer (HWND)
Handle.

This is a handle to the clipboard application window.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_RENDERALLFMTS

This message is sent to the application that owns the clipboard while the application is being destroyed.

Parameters

param1

ulReserved (ULONG)
Reserved value, should be 0.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_RENDERFMT

This message is a request to the clipboard owner to render the data of the format specified in *usfmt*.

Parameters

param1

usfmt (USHORT)
Data format.

This is the format of the data to be rendered.

CF_BITMAP	A bit map.
CF_DSPBITMAP	A bit-map representation of a private data format.
CF_DSPMETAFILE	A metafile representation of a private data format.
CF_DSPTEXT	A textual representation of a private data format.
CF_METAFILE	A metafile.
CF_TEXT	An array of text characters.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_SIZECLIPBOARD

This message is sent when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and the clipboard application window has changed size.

Parameters

param1

hwndViewer (HWND)
Handle of viewer window.

param2

ppaint (PRECTL)
Rectangle to be re-painted.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

WM_VSCROLLCLIPBOARD

This message is sent to the clipboard owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and there is an event in the clipboard viewer's vertical scroll bar.

Parameters

param1

hwndViewer (HWND)
Handle.

This contains a handle to the clipboard application window.

param2

sposScroll (SHORT)
Scroll position.

The position is either:

0	*scodeScroll* is other than SB_SLIDERPOSITION
Other	The position of the slider when *scodeScroll* is SB_SLIDERPOSITION.

scodeScroll (SHORT)
Scroll-bar code.

This is one of the SB_* scroll-bar codes as defined in WM_HSCROLL (in Horizontal Scroll Bars).

SB_LINELEFT	Sent if the operator clicks the left arrow of the scroll bar, or depresses the VK_LEFT key.
SB_LINERIGHT	Sent if the operator clicks the right arrow of the scroll bar, or depresses the VK_RIGHT key.
SB_PAGELEFT	Sent if the operator clicks the area to the left of the slider, or depresses the VK_PAGELEFT key.
SB_PAGERIGHT	Sent if the operator clicks the area to the right of the slider, or depresses the VK_PAGERIGHT key.
SB_SLIDERPOSITION	Sent to indicate the final position of the slider. *sposScroll* contains the final position of the slider.
SB_SLIDERTRACK	Sent every time the slider position changes if the operator moves the scroll bar slider with the pointer device.
SB_ENDSCROLL	Sent when the operator has finished scrolling, but only if the operator has not been doing any absolute slider positioning.

Returns

ulReserved (ULONG)

Reserved value, should be 0.

Summary

Following are the OS/2 functions and messages used with the clipboard:

Table 22-4. Clipboard Functions

Function name	Description
WinCloseClipbrd	Closes the clipboard, enabling other applications to open it by calling WinOpenClipbrd.
WinEmptyClipbrd	Empties the clipboard, removing and freeing all handles to data that is in the clipboard.
WinEnumClipbrdFmts	Enumerates the list of clipboard data formats available in the clipboard.
WinOpenClipbrd	Opens the clipboard.
WinQueryClipbrdData	Obtains a handle to the current clipboard data with a specified format.
WinQueryClipbrdFmtInfo	Determines whether a particular format of data is present in the clipboard; and, if so, provides information about that format.
WinQueryClipbrdOwner	Obtains any current clipboard owner window.
WinQueryClipbrdViewer	Obtains any current clipboard viewer window.
WinSetClipbrdData	Puts data into the clipboard.
WinSetClipbrdOwner	Sets the current clipboard owner window.
WinSetClipbrdViewer	Sets the current clipboard viewer window to a specified window.

Table 22-5 (Page 1 of 2). Clipboard Messages

Message	Description
WM_DESTROYCLIPBOARD	Sent to the clipboard owner when the clipboard is emptied through a call to WinEmptyClipbrd.
WM_DRAWCLIPBOARD	Sent to the clipboard viewer window whenever the contents of the clipboard change, that is, as a result of the WinCloseClipbrd call following a call to WinSetClipbrdData.
WM_HSCROLLCLIPBOARD	Sent to the clipboard owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format.
WM_PAINTCLIPBOARD	Sent when the clipboard contains a data handle with the CFI_OWNERDISPLAY information flag set.
WM_RENDERALLFMTS	Sent to the application that owns the clipboard while the application is being destroyed.
WM_RENDERFMT	A request to the clipboard owner to render the data of the format specified in *usfmt*.

Table 22-5 (Page 2 of 2). Clipboard Messages

Message	Description
WM_SIZECLIPBOARD	Sent when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and the clipboard application window has changed size.
WM_VSCROLLCLIPBOARD	Sent to the clipboard owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format.

Chapter 23. Window Timers

A *window timer* enables an application to post timer messages at specified intervals. This chapter describes how to use window timers in PM applications.

About Window Timers

A window timer causes the system to post WM_TIMER messages to a message queue at specified time intervals called *timeout values*. A timeout value is expressed in milliseconds.

An application starts the timer for a given window, specifying the timeout value. The system counts down approximately that number of milliseconds and posts a WM_TIMER message to the message queue for the corresponding window. The system repeats the countdown-post cycle continuously until the application stops the timer.

The timeout value can be any value in the range from 0 through 4,294,967,295 (full magnitude of *ULONG*) for OS/2 Version 3; for previous versions, the maximum value is 65535. However, the operating system cannot guarantee that all values are accurate. The actual timeout depends on how often the application retrieves messages from the queue and the system clock rate. In many computers, the operating system clock ticks about every 50 milliseconds, but this can vary widely from computer to computer. In general, a timer message cannot be posted more frequently than every system clock tick. To make the system post a timer message as often as possible, an application can set the timeout value to 0.

An application starts a timer by using the WinStartTimer function. If a window handle is given, the timer is created for that window. In such case, the WinDispatchMsg function dispatches the WM_TIMER message to the given window when the message is retrieved from the message queue. If a NULL window handle is given, it is up to the application to check the message queue for WM_TIMER messages and dispatch them to the appropriate window.

A new timer starts counting down as soon as it is created. An application can reset or change a timer's timeout value in subsequent calls to the WinStartTimer function. To stop a timer, an application can use the WinStopTimer function.

The system contains a limited number of timers that must be shared among all PM applications; each application should use as few timers as possible. An application can determine how many timers currently are available by checking the SV_CTIMERS system value.

Every timer has a unique timer identifier. An application can request that a timer be created with a particular identifier or have the system choose a unique value. When a WM_TIMER message is received, the timer identifier is contained in the first message parameter. Timer identifiers enable an application to determine the source of the WM_TIMER message.

Three timer identifiers are reserved by and for the system and cannot be used by applications; these system timer identifiers and their symbolic constants are shown in the following table:

Table 23-1. System Timers

Value	Meaning
TID_CURSOR	Identifies the timer that controls cursor blinking. Its timeout value is stored in the os2.ini file under the CursorBlinkRate keyname in the PM_ControlPanel section.
TID_FLASHWINDOW	Identifies the window-flashing timer.
TID_SCROLL	Identifies the scroll-bar repetition timer that controls scroll-bar response when the mouse button or a key is held down. Its timeout value is specified by the system value SV_SCROLLRATE.

WM_TIMER messages, like WM_PAINT and semaphore messages, are not actually posted to a message queue. Instead, when the time elapses, the system sets a record in the queue indicating which timer message was posted. The system builds the WM_TIMER message when the application retrieves the message from the queue.

Although a timer message may be in the queue, if there are any messages with higher priority in the queue, the application retrieves those messages first. If the time elapses again before the message is retrieved, the system does not create a separate record for this timer, meaning that the application should not depend on the timer messages being processed at precise intervals. To check the accuracy of the message, an application can retrieve the actual system time by using the WinGetCurrentTime function. Comparing the actual time with the time of the previous timer message is useful in determining what action to take for the timer.

Using Window Timers

There are two methods of using window timers. In the first method, you start the timer by using the WinStartTimer function, supplying the window handle and timer identifier. The function associates the timer with the specified window. The following code fragment starts two timers: the first timer is set for every half second (500 milliseconds); the second, for every two seconds (2000 milliseconds).

```
WinStartTimer(hab, /* Anchor-block handle */
    hwnd,          /* Window handle       */
    ID_TIMER1,     /* Timer identifier    */
    500);          /* 500 milliseconds    */

WinStartTimer(hab, /* Anchor-block handle */
    hwnd,          /* Window handle       */
    ID_TIMER2,     /* Timer identifier    */
    2000);         /* 2000 milliseconds   */
```

Once these timers are started, the WinDispatchMsg function dispatches WM_TIMER messages to the appropriate window. To process these messages, add a WM_TIMER case to the window procedure for the given window. By checking the first parameter of the WM_TIMER message, you can identify a particular timer, then carry out the actions related to it. The following code fragment shows how to process WM_TIMER messages:

```
case WM_TIMER:
    switch (SHORT1FROMMP(mp1)) { /* Obtains timer identifier */
        case ID_TIMER1:
            .
            . /* Carry out timer-related tasks.              */
            .
            return 0;

        case ID_TIMER2:
            .
            . /* Carry out timer-related tasks.              */
            .
            return 0;
    }
```

In the second method of using a timer, you specify NULL as the *hwnd* parameter of the WinStartTimer call. The system starts a timer that has no associated window and assigns an arbitrary timer identifier. The following code fragment starts two window timers using this method:

```
ULONG idTimer1, idTimer2;

idTimer1 = WinStartTimer(hab, (HWND) NULL, 0, 500);
idTimer2 = WinStartTimer(hab, (HWND) NULL, 0, 2000);
```

These timers have no associated window, so the application must check the message queue for WM_TIMER messages and dispatch them to the appropriate window procedure. The following code fragment shows a message loop that handles the window timers:

```
HWND hwndTimerHandler; /* Handle of window for timer messages */
QMSG qmsg;             /* Queue-message structure             */

while (WinGetMsg(hab, &qmsg, (HWND) NULL, 0, 0)) {
    if (qmsg.msg == WM_TIMER)
        qmsg.hwnd = hwndTimerHandler;
    WinDispatchMsg(hab, &qmsg);
}
```

You can use the WinStopTimer function at any time to stop a timer. The following code fragment demonstrates how to stop a timer:

```
WinStopTimer(hab, hwnd, ID_TIMER1);      /* Stops first timer */
```

Related Functions

This section covers the functions that are related to Windows Timers.

WinGetCurrentTime

This function returns the current time.

Syntax

```
#define INCL_WINTIMER /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
ULONG WinGetCurrentTime  (HAB hab)
```

Parameters

hab (HAB) – input
    Anchor-block handle.

Returns

ulTime (ULONG) – returns
    System-timer count.

WinStartTimer

This function starts a timer.

Syntax

```
#define INCL_WINTIMER /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG WinStartTimer  (HAB hab, HWND hwnd, ULONG idTimer,
                      ULONG dtTimeout)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwnd (HWND) – input
Window handle that is part of the timer identification.

NULLHANDLE — The *idTimer* parameter is ignored, and this function returns a unique, nonzero, identity which represents that timer. The timer message is posted in the queue associated with the current thread, with the *hwnd* parameter of the QMSG structure set to NULLHANDLE.

Other — Window handle.

idTimer (ULONG) – input
Timer identifier.

dtTimeout (ULONG) – input
Delay time in milliseconds.

Returns

idTimer (ULONG) – returns
Timer identity.

WinStopTimer

This function stops a timer.

Syntax

```
#define INCL_WINTIMER /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>

BOOL WinStopTimer  (HAB hab, HWND hwnd, ULONG ulTimer)
```

Parameters

hab (HAB) – input
Anchor-block handle.

hwnd (HWND) – input
Window handle.

ulTimer (ULONG) – input
Timer identifier.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred, or timer did not exist.

Related Messages

This section covers the messages that are related to Windows Timers.

WM_TIMER

This message is posted when a timer times out.

Parameters

param1

idTimer (USHORT)
Timer identity.

Any timer Ids that are not being used must be passed on the default window procedure.

param2

ulReserved (ULONG)
Reserved value, should be 0.

Returns

ulReserved (ULONG)
Reserved value, should be 0.

Summary

Following are the OS/2 functions and the message used with window timers:

Table 23-2. Window Timer Functions

Function Name	Description
WinGetCurrentTime	Returns the current time.
WinStartTimer	Starts a timer.
WinStopTimer	Stops a timer.

Table 23-3. Window Timer Message

Message	Description
WM_TIMER	Posted when a timer times out.

Chapter 24. Initialization Files

Initialization files enable an application to store and retrieve information that the application uses when it starts up. This chapter describes how to use the OS/2 Profile Manager to create, manage, and use the system's initialization files. The following topics are related to this chapter:

- File system
- Presentation Manager interface applications.

About Initialization Files

An initialization file is a convenient place to store information between sessions. Profile Manager enables applications to create their own initialization files and to access the OS/2 initialization files, *os2.ini* and *os2sys.ini*. Just as the system uses the os2.ini and os2sys.ini files to store configuration information for system startup, an application can create an initialization file that stores information it uses to initialize windows and data.

The system initialization files contain sections and settings used by the PM applications. Although applications can read settings from the initialization files, only rarely does an application need to change a setting. OS/2 initialization files are binary; the user cannot view or edit them directly.

An initialization file consists of one or more sections; each section contains one or more settings, or keys. Each key consists of two parts: a name and a value. Both section names and key names are null-terminated strings. The value assigned to a key can be a null-terminated string, a null-terminated string representing a signed integer, or individual bytes of data.

Once an initialization file is created, an application can rename, copy, move, or delete that file just as it does any other file. Although an application also could read directly to or write directly to the initialization file, the application should always use Profile Manager functions to access the contents of the file. Both character-based OS/2 applications and PM applications can use Profile Manager functions. Before calling Profile Manager, a thread must initialize an anchor block by using the WinInitialize function.

Using Initialization Files

This section explains how to use Profile Manager functions to perform the following tasks:

- Create, open, and close initialization files.
- Read and write settings.
- Identify the initialization files.

Creating, Opening, and Closing Initialization Files

You can create an initialization file or open an existing initialization file by using the PrfOpenProfile function. The function requires a handle to an anchor block and a pointer to the name of an initialization file. If the file does not exist in the given path, the function automatically creates an initialization file.

The following code fragment creates an initialization file named *pmtools.ini* in the current directory:

```
HAB hab;
HINI hini;

hab = WinInitialize(0);
if ((hini = PrfOpenProfile(hab, "pmtools.ini")) == NULL){
                      .
                      . /* File was not created */
                      .
   }
```

If the PrfOpenProfile function is successful, it returns a handle to the initialization file. Otherwise, it returns NULL, and the file is not created. Once you have an initialization-file handle, you can create new sections and settings in the file.

To close an initialization file, you use the PrfCloseProfile function.

Reading and Writing Settings

An application can store strings, integers, and binary data in an initialization file and retrieve them. To read from or write to an initialization file, your application must provide a section name and a key name that specify which setting to read or change. If the section or key name you specify in a writing operation does not exist in the file, it is added to the file and assigned the given value.

The following code fragment creates a section named “MyApp” and a key named “MainWindowColor” in a previously opened initialization file, and assigns the value of the RGB structure to the new setting:

```
HINI hini;
RGB rgb = { 0xff, 0x00, 0x00 };

PrfWriteProfileData(hini, "MyApp", "MainWindowColor", &rgb, sizeof(RGB));
```

To read a setting, your application can retrieve the size of the setting and then read the setting into an appropriate buffer by using the PrfQueryProfileSize and PrfQueryProfileData functions, as shown in the following example. This example reads the setting "MainWindowColor" from the "MyApp" section only if the size of the data is equal to the size of the RGB structure.

```
HINI hini;
ULONG cb;
RGB rgb;

PrfQueryProfileSize(hini, "MyApp", "MainWindowColor", &cb);
if (cb == sizeof(RGB))
    PrfQueryProfileData(hini, "MyApp", "MainWindowColor", &rgb, &cb);
```

An application can also read strings by using the PrfQueryProfileString function, write strings by using the PrfWriteProfileString function, and read integers (stored as strings) by using the PrfQueryProfileInt function.

Identifying the OS/2 Initialization Files

Your application can retrieve the names of the system initialization files by using the PrfQueryProfile function. Although the OS/2 initialization files are usually named os2.ini and os2sys.ini, you can use other files when starting the system.

The following example retrieves the names of the initialization files and copies their names to the strings szUserName and szSysName. Once you know the names of the OS/2 initialization files, you can use them to open the files and read settings.

```
CHAR szUserName[CCHMAXPATH];
CHAR szSysName[CCHMAXPATH];
HINI hini;

PRFPROFILE prfpro = { sizeof(szUserName), szUserName,
                      sizeof(szSysName), szSysName };

PrfQueryProfile(hini, &prfpro);
```

You can change the OS/2 initialization files to files of your choice by using the PrfReset function. This function requires the names of two initialization files and uses them as replacements for the os2.ini and os2sys.ini files. The system is then reset by using the settings in the new files.

Related Functions

This section covers the functions that are related to Initialization Files.

PrfCloseProfile

This function indicates that a profile is no longer available for use.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL PrfCloseProfile  (HINI hini)
```

Parameters

hini (HINI) – input
Initialization-file handle.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion.
FALSE	Error occurred.

PrfOpenProfile

This function indicates that a file is available for use as a profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
HINI PrfOpenProfile  (HAB hab, PSZ pszFileName)
```

Parameters

hab (HAB) – input
Anchor-block handle.

pszFileName (PSZ) – input
User-profile file name.

Returns

hini (HINI) – returns
Initialization-file handle.

NULLHANDLE	Error occurred
Other	Initialization-file handle.

PrfQueryProfile

This function returns a description of the current user and system profiles.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL PrfQueryProfile  (HAB hab, PPRFPROFILE pprfproProfile)
```

Parameters

hab (HAB) – input

Anchor-block handle.

pprfproProfile (PPRFPROFILE) – in/out

Profile names structure.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred, or there was insufficient space to record the names, which have been truncated.

PrfQueryProfileData

This function returns a string of binary data from the specified profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL PrfQueryProfileData  (HINI hini, PSZ pszApp, PSZ pszKey, PVOID pBuffer,
                           PULONG pulBufferMax)
```

Parameters

hini (HINI) – input
Initialization-file handle.

HINI_PROFILE	Both the user profile and system profile are searched
HINI_USERPROFILE	The user profile is searched
HINI_SYSTEMPROFILE	The system profile is searched
Other	Initialization-file handle.

pszApp (PSZ) – input
Application name.

pszKey (PSZ) – input
Key name.

pBuffer (PVOID) – output
Value data.

pulBufferMax (PULONG) – in/out
Size of value data.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

PrfQueryProfileInt

This function returns an integer value from the specified profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

LONG PrfQueryProfileInt  (HINI hini, PSZ pszApp, PSZ pszKey, LONG lDefault)
```

Parameters

hini (HINI) – input
Initialization-file handle.

HINI_PROFILE	Both the user profile and system profile are searched
HINI_USERPROFILE	The user profile is searched
HINI_SYSTEMPROFILE	The system profile is searched
Other	Initialization-file handle returned by PrfOpenProfile.

pszApp (PSZ) – input
Application name.

pszKey (PSZ) – input
Key name.

lDefault (LONG) – input
Default value.

Returns

lResult (LONG) – returns
Key value specified in the initialization file.

PrfQueryProfileSize

This function obtains the size in bytes of the value of a specified key for a specified application in the profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL PrfQueryProfileSize (HINI hini, PSZ pszApp, PSZ pszKey,
                          PULONG pDataLen)
```

Parameters

hini (HINI) – input
Initialization-file handle.

HINI_PROFILE	Both the user profile and system profile are searched
HINI_USERPROFILE	The user profile is searched
HINI_SYSTEMPROFILE	The system profile is searched
Other	Initialization-file handle returned by PrfOpenProfile.

pszApp (PSZ) – input
Application name.

pszKey (PSZ) – input
Key name.

pDataLen (PULONG) – output
Data length.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

PrfQueryProfileString

This function retrieves a string from the specified profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

ULONG PrfQueryProfileString  (HINI hini, PSZ pszApp, PSZ pszKey,
                              PSZ pszDefault, PVOID pBuffer,
                              ULONG cchBufferMax)
```

Parameters

hini (HINI) – input
Initialization-file handle.

HINI_PROFILE	Both the user profile and system profile are searched
HINI_USERPROFILE	The user profile is searched
HINI_SYSTEMPROFILE	The system profile is searched
Other	Initialization-file handle returned by the PrfOpenProfile function.

pszApp (PSZ) – input
Application name.

pszKey (PSZ) – input
Key name.

pszDefault (PSZ) – input
Default string.

pBuffer (PVOID) – output
Profile string.

cchBufferMax (ULONG) – input
Maximum string length.

Returns

pulLength (ULONG) – returns
String length returned.

PrfReset

This function defines which files are to be used as the user and system profiles.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */
#include <os2.h>
BOOL PrfReset  (HAB hab, PRFPROFILE prfproProfile)
```

Parameters

hab (HAB) – input
Anchor-block handle.

prfproProfile (PRFPROFILE) – input
Profile-names structure.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

PrfWriteProfileData

This function writes a string of binary data into the specified profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL PrfWriteProfileData  (HINI hini, PSZ pszApp, PSZ pszKey, PVOID pData,
                           ULONG  cchDataLen)
```

Parameters

hini (HINI) – input

Initialization-file handle.

HINI_PROFILE	User profile
HINI_USERPROFILE	User profile
HINI_SYSTEMPROFILE	System profile
Other	Initialization-file handle returned by PrfOpenProfile.

pszApp (PSZ) – input

Application name.

pszKey (PSZ) – input

Key name.

pData (PVOID) – input

Value data.

cchDataLen (ULONG) – input

Size of value data.

Returns

rc (BOOL) – returns

Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

PrfWriteProfileString

This function writes a string of character data into the specified profile.

Syntax

```
#define INCL_WINSHELLDATA /* Or use INCL_WIN, INCL_PM, */

#include <os2.h>

BOOL PrfWriteProfileString  (HINI hini, PSZ pszApp, PSZ pszKey, PSZ pszData)
```

Parameters

hini (HINI) – input
Initialization-file handle.

HINI_PROFILE	User profile
HINI_USERPROFILE	User profile
HINI_SYSTEMPROFILE	System profile
Other	Initialization-file handle returned by PrfOpenProfile.

pszApp (PSZ) – input
Application name.

pszKey (PSZ) – input
Key name.

pszData (PSZ) – input
Text string.

Returns

rc (BOOL) – returns
Success indicator.

TRUE	Successful completion
FALSE	Error occurred.

Related Data Structures

This section covers the data structures that are related to Initialization Files.

PRFPROFILE

Profile structure.

Syntax

```
typedef struct _PRFPROFILE {
ULONG      cchUserName;
PSZ        pszUserName;
ULONG      cchSysLen;
PSZ        pszSysName;
 } PRFPROFILE;

typedef PRFPROFILE *PPRFPROFILE;
```

Fields

cchUserName (ULONG)
Length of user profile name.

pszUserName (PSZ)
User profile name.

cchSysLen (ULONG)
Length of system profile name.

pszSysName (PSZ)
System profile name.

Summary

Following are the functions and data structure used with initialization files:

Table 24-1. Initialization File Functions

Function name	Description
PrfCloseProfile	Indicates that a profile is no longer available for use.
PrfOpenProfile	Indicates that a file is available for use as a profile
PrfQueryProfile	Returns a description of the current user and system profiles.
PrfQueryProfileData	Returns a string of binary data from the specified profile.
PrfQueryProfileInt	Returns an integer value from the specified profile.
PrfQueryProfileSize	Obtains the size, in bytes, of the value of a specified key for a specified application in the profile.
PrfQueryProfileString	Retrieves a string from the specified profile.
PrfReset	Defines which files are to be used as the user and system profiles.
PrfWriteProfileData	Writes a string of binary data into the specified profile.
PrfWriteProfileString	Writes a string of character data into the specified profile.

Table 24-2. Initialization File Structures

Structure name	Description
PRFPROFILE	Profile Structure.

Appendix A. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood NY 10594, U.S.A.

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries:

Common User Access
CUA
IBM
Operating System/2
OS/2
Presentation Manager
SAA
Systems Application Architecture
Workplace Shell

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies as follows. Other trademarks are trademarks of their respective companies.

Helvetica	Trademark of Linotype Company

Double-Byte Character Set (DBCS)

Throughout this publication, you will see reference to specific values for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), note that one DBCS equals two SBCS characters.

Glossary

This glossary defines many of the terms used in this book. It includes terms and definitions from the *IBM Dictionary of Computing*, as well as terms specific to the OS/2 operating system and the Presentation Manager. It is not a complete glossary for the entire OS/2 operating system; nor is it a complete dictionary of computer terms.

Other primary sources for these definitions are:

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyrighted 1990 by the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. These definitions are identified by the symbol (A) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

Glossary Listing

A

accelerator. In SAA Common User Access architecture, a key or combination of keys that invokes an application-defined function.

accelerator table. A table used to define which key strokes are treated as *accelerators* and the commands they are translated into.

access mode. The manner in which an application gains access to a file it has opened. Examples of access modes are read-only, write-only, and read/write.

access permission. All access rights that a user has regarding an object. (I)

action. One of a set of defined tasks that a computer performs. Users request the application to perform an action in several ways, such as typing a command, pressing a function key, or selecting the action name from an action bar or menu.

action bar. In SAA Common User Access architecture, the area at the top of a window that contains choices that give a user access to actions available in that window.

action point. The current position on the screen at which the pointer is pointing. Contrast with *hot spot* and *input focus*.

active program. A program currently running on the computer. An active program can be interactive (running and receiving input from the user) or noninteractive (running but not receiving input from the user). See also *interactive program* and *noninteractive program*.

active window. The window with which the user is currently interacting.

address space. (1) The range of addresses available to a program. (A) (2) The area of virtual storage available for a particular job.

alphanumeric video output. Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

Note: IBM has defined an extension to ASCII code (characters 128-255).

anchor. A window procedure that handles Presentation Manager* message conversions between an icon procedure and an application.

anchor block. An area of Presentation-Manager-internal resources to allocated process or thread that calls WinInitialize.

anchor point. A point in a window used by a program designer or by a window manager to position a subsequently appearing window.

ANSI. American National Standards Institute.

APA. All points addressable.

API. Application programming interface.

application. A collection of software components used to perform specific types of work on a computer; for example, a payroll application, an airline reservation application, a network application.

application object. In SAA Advanced Common User Access architecture, a form that an application provides for a user; for example, a spreadsheet form. Contrast with *user object*.

application programming interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application-modal. Pertaining to a message box or dialog box for which processing must be completed before further interaction with any other window owned by the same application may take place.

area. In computer graphics, a filled shape such as a solid rectangle.

ASCII. American National Standard Code for Information Interchange.

ASCIIZ. A string of ASCII characters that is terminated with a byte containing the value 0.

aspect ratio. In computer graphics, the width-to-height ratio of an area, symbol, or shape.

asynchronous (ASYNC). (1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as common timing signals. (T) (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions. See also *synchronous*.

atom. A constant that represents a string. As soon as a string has been defined as an atom, the atom can be used in place of the string to save space. Strings are associated with their respective atoms in an *atom table*. See also *integer atom*.

atom table. A table used to relate *atoms* with the strings that they represent. Also in the table is the mechanism by which the presence of a string can be checked.

atomic operation. An operation that completes its work on an object before another operation can be performed on the same object.

attribute. A characteristic or property that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *graphics attributes* and *segment attributes*.

automatic link. In Information Presentation Facility (IPF), a link that begins a chain reaction at the primary window. When the user selects the primary window, an automatic link is activated to display secondary windows.

AVIO. Advanced Video Input/Output.

B

Bézier curve. (1) A mathematical technique of specifying smooth continous lines and surfaces, which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bézier. (2) (D of C) In the AIX Graphics Library, a cubic spline approximation to a set of four control points that passes through the first and fourth control points and that has a continuous slope where two spline segments meet. Named after Dr. P. Bézier.

background. (1) In multiprogramming, the conditions under which low-priority programs are executed. Contrast with *foreground*. (2) An active session that is not currently displayed on the screen.

background color. The color in which the background of a graphic primitive is drawn.

background mix. An attribute that determines how the background of a graphic primitive is combined with the existing color of the graphics presentation space. Contrast with *mix*.

background program. In multiprogramming, a program that executes with a low priority. Contrast with *foreground program*.

bit map. A representation in memory of the data displayed on an APA device, usually the screen.

block. (1) A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or logical records. (T) (2) To record data in a block. (3) A collection of contiguous records recorded as a unit. Blocks are separated by interblock gaps and each block may contain one or more records. (A)

block device. A storage device that performs I/O operations on blocks of data called *sectors*. Data on block devices can be randomly accessed. Block devices are designated by a drive letter (for example, **C:**).

blocking mode. A condition set by an application that determines when its threads might block. For example, an application might set the Pipemode parameter for the DosCreateNPipe function so that its threads perform I/O operations to the named pipe block when no data is available.

border. A visual indication (for example, a separator line or a background color) of the boundaries of a window.

boundary determination. An operation used to compute the size of the smallest rectangle that encloses a graphics object on the screen.

breakpoint. (1) A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. (T) (2) A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broken pipe. When all of the handles that access one end of a pipe have been closed.

bucket. One or more fields in which the result of an operation is kept.

buffer. (1) A portion of storage used to hold input or output data temporarily. (2) To allocate and schedule the use of buffers. (A)

button. A mechanism used to request or initiate an action. See also *barrel buttons*, *bezel buttons*, *mouse button*, *push button*, and *radio button*.

byte pipe. Pipes that handle data as byte streams. All unnamed pipes are byte pipes. Named pipes can be byte pipes or message pipes. See *byte stream*.

byte stream. Data that consists of an unbroken stream of bytes.

C

cache. A high-speed buffer storage that contains frequently accessed instructions and data; it is used to reduce access time.

cached micro presentation space. A presentation space from a Presentation-Manager-owned store of micro presentation spaces. It can be used for drawing to a window only, and must be returned to the store when the task is complete.

CAD. Computer-Aided Design.

call. (1) The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A) (2) To transfer control to a procedure, program, routine, or subroutine.

calling sequence. A sequence of instructions together with any associated data necessary to execute a call. (T)

Cancel. An action that removes the current window or menu without processing it, and returns the previous window.

cascaded menu. In the OS/2 operating system, a menu that appears when the arrow to the right of a cascading choice is selected. It contains a set of choices that are related to the cascading choice. Cascaded menus are used to reduce the length of a menu. See also *cascading choice*.

cascading choice. In SAA Common User Access architecture, a choice in a menu that, when selected, produces a cascaded menu containing other choices. An arrow (→) appears to the right of the cascading choice.

CASE statement. In PM programming, provides the body of a window procedure. There is usually one CASE statement for each message type supported by an application.

CGA. Color graphics adapter.

chained list. A list in which the data elements may be dispersed but in which each data element contains information for locating the next. (T) Synonymous with *linked list*.

character. A letter, digit, or other symbol.

character box. In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single character from a character set. See also *character mode*. Contrast with *character cell*.

character cell. The physical, rectangular space in which any single character is displayed on a screen or printer device. Position is addressed by row and column coordinates. Contrast with *character box*.

character code. The means of addressing a character in a character set, sometimes called *code point*.

character device. A device that performs I/O operations on one character at a time. Because character devices view data as a stream of bytes, character-device data cannot be randomly accessed. Character devices include the keyboard, mouse, and printer, and are referred to by name.

character mode. A mode that, in conjunction with the font type, determines the extent to which graphics characters are affected by the character box, shear, and angle attributes.

character set. (1) An ordered set of unique representations called characters; for example, the 26 letters of English alphabet, Boolean 0 and 1, the set of symbols in the Morse code, and the 128 ASCII characters. (A) (2) All the valid characters for a programming language or for a computer system. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print.

check box. In SAA Advanced Common User Access architecture, a square box with associated text that represents a choice. When a user selects a choice, an X appears in the check box to indicate that the choice is in effect. The user can clear the check box by selecting the choice again. Contrast with *radio button*.

check mark. (1) (D of C) In SAA Advanced Common User Access architecture, a (√) symbol that shows that a choice is currently in effect. (2) The symbol that is used to indicate a selected item on a pull-down menu.

child process. In the OS/2 operating system, a process started by another process, which is called the parent process. Contrast with *parent process*.

child window. A window that appears within the border of its parent window (either a primary window or another child window). When the parent window is resized, moved, or destroyed, the child window also is resized, moved, or destroyed; however, the child window can be moved or resized independently from the parent window, within the boundaries of the parent window. Contrast with *parent window*.

choice. (1) An option that can be selected. The choice can be presented as text, as a symbol (number or letter), or as an icon (a pictorial symbol). (2) (D of C) In SAA Common User Access architecture, an item that a user can select.

chord. (1) To press more than one button on a pointing device while the pointer is within the limits that the user has specified for the operating environment. (2) (D of C) In graphics, a short line segment whose end points lie on a circle. Chords are a means for producing a circular image from straight lines. The higher the number of chords per circle, the smoother the circular image.

class. In object-oriented design or programming, a group of objects that share a common definition and that therefore share common properties, operations, and behavior. Members of the group are called instances of the class.

class method. In System Object Model, an action that can be performed on a class object. Synonymous with factory method.

class object. In System Object Model, the run-time implementation of a class.

class style. The set of properties that apply to every window in a window class.

client. (1) A functional unit that receives shared services from a server. (T) (2) A user, as in a client process that uses a named pipe or queue that is created and owned by a server process.

client area. The part of the window, inside the border, that is below the menu bar. It is the user's work space, where a user types information and selects choices from selection fields. In primary windows, it is where an application programmer presents the objects that a user works on.

client program. An application that creates and manipulates instances of classes.

client window. The window in which the application displays output and receives input. This window is located inside the frame window, under the window title bar and any menu bar, and within any scroll bars.

clip limits. The area of the paper that can be reached by a printer or plotter.

clipboard. In SAA Common User Access architecture, an area of computer memory, or storage, that temporarily holds data. Data in the clipboard is available to other applications.

clipping. In computer graphics, removing those parts of a display image that lie outside a given boundary. (I) (A)

clipping area. The area in which the window can paint.

clipping path. A clipping boundary in world-coordinate space.

clock tick. The minimum unit of time that the system tracks. If the system timer currently counts at a rate of X Hz, the system tracks the time every 1/X of a second. Also known as *time tick*.

CLOCK$. Character-device name reserved for the system clock.

code page. An assignment of graphic characters and control-function meanings to all code points.

code point. (1) Synonym for *character code*. (2) (D of C) A 1-byte code representing one of 256 potential characters.

code segment. An executable section of programming code within a load module.

color dithering. See *dithering*.

color graphics adapter (CGA). An adapter that simultaneously provides four colors and is supported by all IBM Personal Computer and Personal System/2 models.

command. The name and parameters associated with an action that a program can perform.

command area. An area composed of a command field prompt and a command entry field.

command entry field. An entry field in which users type commands.

command line. On a display screen, a display line, sometimes at the bottom of the screen, in which only commands can be entered.

command mode. A state of a system or device in which the user can enter commands.

command prompt. A field prompt showing the location of the command entry field in a panel.

Common Programming Interface (CPI). Definitions of those application development languages and services that have, or are intended to have, implementations on and a high degree of commonality across the SAA environments. One of the three SAA architectural areas. See also *Common User Access architecture*.

Common User Access (CUA) architecture. Guidelines for the dialog between a human and a workstation or terminal. One of the three SAA architectural areas. See also *Common Programming Interface*.

compile. To translate a program written in a higher-level programming language into a machine language program.

composite window. A window composed of other windows (such as a frame window, frame-control windows, and a client window) that are kept together as a unit and that interact with each other.

computer-aided design (CAD). The use of a computer to design or change a product, tool, or machine, such as using a computer for drafting or illustrating.

COM1, COM2, COM3. Character-device names reserved for serial ports 1 through 3.

CON. Character-device name reserved for the console keyboard and screen.

container. In SAA Common User Access architecture, an object that holds other objects. A folder is an example of a container object. See also *folder* and *object*.

contextual help. In SAA Common User Access Architecture, help that gives specific information about the item the cursor is on. The help is contextual because it provides information about a specific item as it is currently being used. Contrast with *extended help*.

contiguous. Touching or joining at a common edge or boundary, for example, an unbroken consecutive series of storage locations.

control. In SAA Advanced Common User Access architecture, a component of the user interface that allows a user to select choices or type information; for example, a check box, an entry field, a radio button.

control area. A storage area used by a computer program to hold control information. (I) (A)

Control Panel. In the Presentation Manager, a program used to set up user preferences that act globally across the system.

Control Program. (1) The basic functions of the operating system, including DOS emulation and the support for keyboard, mouse, and video input/output. (2) A computer program designed to schedule and to supervise the execution of programs of a computer system. (I) (A)

control window. A window that is used as part of a composite window to perform simple input and output tasks. Radio buttons and check boxes are examples.

control word. An instruction within a document that identifies its parts or indicates how to format the document.

coordinate space. A two-dimensional set of points used to generate output on a video display of printer.

Copy. A choice that places onto the clipboard, a copy of what the user has selected. See also *Cut* and *Paste*.

correlation. The action of determining which element or object within a picture is at a given position on the display. This follows a *pick* operation.

coverpage window. A window in which the application's help information is displayed.

CPI. Common Programming Interface.

critical extended attribute. An extended attribute that is necessary for the correct operation of the system or a particular application.

critical section. (1) In programming languages, a part of an asynchronous procedure that cannot be executed simultaneously with a certain part of another asynchronous procedure. (I)

Note: Part of the other asynchronous procedure also is a critical section. (2) A section of code that is not reentrant; that is, code that can be executed by only one thread at a time.

CUA architecture. Common User Access architecture.

current position. In computer graphics, the position, in user coordinates, that becomes the starting point for the next graphics routine, if that routine does not explicitly specify a starting point.

cursor. A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors, and selection cursors. Contrast with *pointer* and *input focus*.

Cut. In SAA Common User Access architecture, a choice that removes a selected object, or a part of an object, to the clipboard, usually compressing the space it occupied in a window. See also *Copy* and *Paste*.

D

daisy chain. A method of device interconnection for determining interrupt priority by connecting the interrupt sources serially.

data segment. A nonexecutable section of a program module; that is, a section of a program that contains data definitions.

data structure. The syntactic structure of symbolic expressions and their storage-allocation characteristics. (T)

data transfer. The movement of data from one object to another by way of the clipboard or by direct manipulation.

DBCS. Double-byte character set.

DDE. Dynamic data exchange.

deadlock. (1) Unresolved contention for the use of a resource. (2) An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by, or a response from, the other. (3) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state.

debug. To detect, diagnose, and eliminate errors in programs. (T)

decipoint. In printing, one tenth of a point. There are 72 points in an inch.

default procedure. A function provided by the Presentation Manager Interface that may be used to process standard messages from dialogs or windows.

default value. A value assumed when no value has been specified. Synonymous with assumed value. For example, in the graphics programming interface, the default line-type is 'solid'.

definition list. A type of list that pairs a term and its description.

delta. An application-defined threshold, or number of container items, from either end of the list.

descendant. See *child process*.

descriptive text. Text used in addition to a field prompt to give more information about a field.

Deselect all. A choice that cancels the selection of all of the objects that have been selected in that window.

Desktop Manager. In the Presentation Manager, a window that displays a list of groups of programs, each of which can be started or stopped.

desktop window. The window, corresponding to the physical device, against which all other types of windows are established.

detached process. A background process that runs independent of the parent process.

detent. A point on a slider that represents an exact value to which a user can move the slider arm.

device context. A logical description of a data destination such as memory, metafile, display, printer, or plotter. See also *direct device context*, *information device context*, *memory device context*, *metafile device context*, *queued device context*, and *screen device context*.

device driver. A file that contains the code needed to attach and use a device such as a display, printer, or plotter.

device space. (1) Coordinate space in which graphics are assembled after all GPI transformations have been applied. Device space is defined in device-specific units. (2) (D of C) In computer graphics, a space defined by the complete set of addressable points of a display device. (A)

dialog. The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box. In SAA Advanced Common User Access architecture, a movable window, fixed in size, containing controls that a user uses to provide information required by an application so that it can continue to process a user request. See also *message box, primary window, secondary window*. Also known as a *pop-up window*.

Dialog Box Editor. A *WYSIWYG* editor that creates dialog boxes for communicating with the application user.

dialog item. A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog procedure. A dialog window that is controlled by a window procedure. It is responsible for responding to all messages sent to the dialog window.

dialog tag language. A markup language used by the DTL compiler to create dialog objects.

dialog template. The definition of a dialog box, which contains details of its position, appearance, and window ID, and the window ID of each of its child windows.

direct device context. A logical description of a data destination that is a device other than the screen (for example, a printer or plotter), and where the output is not to go through the spooler. Its purpose is to satisfy queries. See also *device context*.

direct manipulation. The action of using the mouse to move objects around the screen. For example, moving files and directories around in the *Workplace Shell*.

direct memory access (DMA). A technique for moving data directly between main storage and peripheral equipment without requiring processing of the data by the processing unit.(T)

directory. A type of file containing the names and controlling information for other files or other directories.

display point. Synonym for *pel*.

dithering. (1) The process used in color displays whereby every other pel is set to one color, and the intermediate pels are set to another. Together they produce the effect of a third color at normal viewing distances. This process can only be used on solid areas of color; it does not work, for example, on narrow lines. (2) (D of C) In computer graphics, a technique of interleaving dark and light pixels so that the resulting image looks smoothly shaded when viewed from a distance.

DMA. Direct memory access.

DOS Protect Mode Interface (DPMI). An interface between protect mode and real mode programs.

double-byte character set (DBCS). A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more characters than can be represented by 256 code points, require double-byte character sets. Since each character requires two bytes, the entering, displaying, and printing of DBCS characters requires hardware and software that can support DBCS.

doubleword. A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. (A)

DPMI. DOS Protect Mode Interface.

drag. In SAA Common User Access, to use a pointing device to move an object; for example, clicking on a window border, and dragging it to make the window larger.

dragging. (1) In computer graphics, moving an object on the display screen as if it were attached to the pointer. (2) (D of C) In computer graphics, moving one or more segments on a display surface by translating. (I) (A)

drawing chain. See *segment chain*.

drop. To fix the position of an object that is being dragged, by releasing the select button of the pointing device.

drop. To fix the position of an object that is being dragged, by releasing the select button of the pointing device. See also *drag*.

DTL. Dialog tag language.

dual-boot function. A feature of the OS/2 operating system that allows the user to start DOS from within the operating system, or an OS/2 session from within DOS.

duplex. Pertaining to communication in which data can be sent and received at the same time. Synonymous with *full duplex*.

dynamic data exchange (DDE). A message protocol used to communicate between applications that share data. The protocol uses shared memory as the means of exchanging data between applications.

dynamic data formatting. A formatting procedure that enables you to incorporate text, bit maps or metafiles in an IPF window at execution time.

dynamic link library. A collection of executable programming code and data that is bound to an application at load time or run time, rather than during linking. The programming code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic linking. The process of resolving external references in a program module at load time or run time rather than during linking.

dynamic segments. Graphics segments drawn in exclusive-OR mix mode so that they can be moved from one screen position to another without affecting the rest of the displayed picture.

dynamic storage. (1) A device that stores data in a manner that permits the data to move or vary with time such that the specified data is not always available for recovery. (A) (2) A storage in which the cells require repetitive application of control signals in order to retain stored data. Such repetitive application of the control signals is called a refresh operation. A dynamic storage may use static addressing or sensing circuits. (A) (3) See also *static storage*.

dynamic time slicing. Varies the size of the time slice depending on system load and paging activity.

dynamic-link module. A module that is linked at load time or run time.

E

EBCDIC. Extended binary-coded decimal interchange code. A coded character set consisting of 8-bit coded characters (9 bits including parity check), used for information interchange among data processing systems, data communications systems, and associated equipment.

edge-triggered. Pertaining to an event semaphore that is posted then reset before a waiting thread gets a chance to run. The semaphore is considered to be posted for the rest of that thread's waiting period; the thread does not have to wait for the semaphore to be posted again.

EGA. Extended graphics adapter.

element. An entry in a graphics segment that comprises one or more graphics orders and that is addressed by the element pointer.

EMS. Expanded Memory Specification.

encapsulation. Hiding an object's implementation, that is, its private, internal data and methods. Private variables and methods are accessible only to the object that contains them.

entry field. In SAA Common User Access architecture, an area where a user types information. Its boundaries are usually indicated. See also *selection field*.

entry panel. A defined panel type containing one or more entry fields and protected information such as headings, prompts, and explanatory text.

entry-field control. The component of a user interface that provides the means by which the application receives data entered by the user in an entry field. When it has the input focus, the entry field displays a flashing pointer at the position where the next typed character will go.

environment segment. The list of environment variables and their values for a process.

environment strings. ASCII text strings that define the value of environment variables.

environment variables. Variables that describe the execution environment of a process. These variables are named by the operating system or by the application. Environment variables named by the operating system are PATH, DPATH, INCLUDE, INIT, LIB, PROMPT, and TEMP. The values of environment variables are defined by the user in the CONFIG.SYS file, or by using the SET command at the OS/2 command prompt.

error message. An indication that an error has been detected. (A)

event semaphore. A semaphore that enables a thread to signal a waiting thread or threads that an event has occurred or that a task has been completed. The waiting threads can then perform an action that is dependent on the completion of the signaled event.

exception. An abnormal condition such as an I/O error encountered in processing a data set or a file.

exclusive system semaphore. A system semaphore that can be modified only by threads within the same process.

executable file. (1) A file that contains programs or commands that perform operations or actions to be taken. (2) A collection of related data records that execute programs.

exit. To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T) Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel*.

expanded memory specification (EMS). Enables DOS applications to access memory above the 1MB real mode addressing limit.

extended attribute. An additional piece of information about a file object, such as its data format or category. It consists of a name and a value. A file object may have more than one extended attribute associated with it.

extended help. In SAA Common User Access architecture, a help action that provides information about the contents of the application window from which a user requested help. Contrast with *contextual help*.

extended-choice selection. A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection*.

extent. Continuous space on a disk or diskette that is occupied by or reserved for a particular data set, data space, or file.

external link. In Information Presentation Facility, a link that connects external online document files.

F

family-mode application. An application program that can run in the OS/2 environment and in the DOS environment; however, it cannot take advantage of many of the OS/2-mode facilities, such as multitasking, interprocess communication, and dynamic linking.

FAT. File allocation table.

FEA. Full extended attribute.

field-level help. Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used; the information is dependent upon the context within the work session.

FIFO. First-in-first-out. (A)

file. A named set of records stored or processed as a unit. (T)

file allocation table (FAT). In IBM personal computers, a table used by the operating system to allocate space on a disk for a file, and to locate and chain together parts of the file that may be scattered on different sectors so that the file can be used in a random or sequential manner.

file attribute. Any of the attributes that describe the characteristics of a file.

File Manager. In the Presentation Manager, a program that displays directories and files, and allows various actions on them.

file specification. The full identifier for a file, which includes its drive designation, path, file name, and extension.

file system. The combination of software and hardware that supports storing information on a storage device.

file system driver (FSD). A program that manages file I\O and controls the format of information on the storage media.

fillet. A curve that is tangential to the end points of two adjoining lines. See also *polyfillet*.

filtering. An application process that changes the order of data in a queue.

first-in-first-out (FIFO). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

flag. (1) An indicator or parameter that shows the setting of a switch. (2) A character that signals the occurrence of some condition, such as the end of a word. (A) (3) (D of C) A characteristic of a file or directory that enables it to be used in certain ways. See also *archive flag*, *hidden flag*, and *read-only flag*.

focus. See *input focus*.

folder. A container used to organize objects.

font. A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

Font Editor. A utility program provided with the IBM Developers Toolkit that enables the design and creation of new fonts.

foreground program. (1) The program with which the user is currently interacting. Also known as *interactive program*. Contrast with *background program*. (2) (D of C) In multiprogramming, a high-priority program.

frame. The part of a window that can contain several different visual elements specified by the application, but drawn and controlled by the Presentation Manager. The frame encloses the client area.

frame styles. Standard window layouts provided by the Presentation Manager.

FSD. File system driver.

full-duplex. Synonym for *duplex*.

full-screen application. An application that has complete control of the screen.

function. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a call. (2) A set of related control statements that cause one or more programs to be performed.

function key. A key that causes a specified sequence of operations to be performed when it is pressed, for example, F1 and Alt-K.

function key area. The area at the bottom of a window that contains function key assignments such as F1=Help.

G

GDT. Global Descriptor Table.

general protection fault. An exception condition that occurs when a process attempts to use storage or a module that has some level of protection assigned to it, such as I/O privilege level. See also *IOPL code segment*.

Global Descriptor Table (GDT). A table that defines code and data segments available to all tasks in an application.

global dynamic-link module. A dynamic-link module that can be shared by all processes in the system that refer to the module name.

global file-name character. Either a question mark (?) or an asterisk (*) used as a variable in a file name or file name extension when referring to a particular file or group of files.

glyph. A graphic symbol whose appearance conveys information.

GPI. Graphics programming interface.

graphic primitive. In computer graphics, a basic element, such as an arc or a line, that is not made up of smaller parts and that is used to create diagrams and pictures. See also *graphics segment*.

graphics. (1) A picture defined in terms of graphic primitives and graphics attributes. (2) (D of C) The making of charts and pictures. (3) Pertaining to charts, tables, and their creation. (4) See *computer graphics, coordinate graphics, fixed-image graphics, interactive graphics, passive graphics, raster graphics*.

graphics attributes. Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition. See also *segment attributes*.

graphics field. The clipping boundary that defines the visible part of the presentation-page contents.

graphics mode. One of several states of a display. The mode determines the resolution and color content of the screen.

graphics model space. The conceptual coordinate space in which a picture is constructed after any model transforms have been applied. Also known as *model space*.

Graphics programming interface. The formally defined programming language that is between an IBM graphics program and the user of the program.

graphics segment. A sequence of related graphic primitives and graphics attributes. See also *graphic primitive*.

graying. The indication that a choice on a pull-down is unavailable.

group. A collection of logically connected controls. For example, the buttons controlling paper size for a printer could be called a group. See also *program group*.

H

handle. (1) An identifier that represents an object, such as a device or window, to the Presentation Interface. (2) (D of C) In the Advanced DOS and OS/2 operating systems, a binary value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

hard error. An error condition on a network that requires either that the system be reconfigured or that the source of the error be removed before the system can resume reliable operation.

header. (1) System-defined control information that precedes user data. (2) The portion of a message that contains control information for the message, such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

heading tags. A document element that enables information to be displayed in windows, and that controls entries in the contents window controls placement of push buttons in a window, and defines the shape and size of windows.

heap. An area of free storage available for dynamic allocation by an application. Its size varies according to the storage requirements of the application.

help function. (1) A function that provides information about a specific field, an application panel, or information about the help facility. (2) (D of C) One or more display images that describe how to use application software or how to do a system operation.

Help index. In SAA Common User Access architecture, a help action that provides an index of the help information available for an application.

help panel. A panel with information to assist users that is displayed in response to a help request from the user.

help window. A Common-User-Access-defined secondary window that displays information when the user requests help.

hidden file. An operating system file that is not displayed by a directory listing.

hide button. In the OS/2 operating system, a small, square button located in the right-hand corner of the title bar of a window that, when selected, removes from the screen all the windows associated with that window. Contrast with *maximize button*. See also *restore button*.

hierarchical inheritance. The relationship between parent and child classes. An object that is lower in the inheritance hierarchy than another object, inherits all the characteristics and behaviors of the objects above it in the hierarchy.

hierarchy. A tree of segments beginning with the root segment and proceeding downward to dependent segment types.

high-performance file system (HPFS). In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the coexistence of multiple, active file systems on a single personal computer, with the capability of multiple and different storage devices. File names used with the HPFS can have as many as 254 characters.

hit testing. The means of identifying which window is associated with which input device event.

hook. A point in a system-defined function where an application can supply additional code that the system processes as though it were part of the function.

hook chain. A sequence of hook procedures that are "chained" together so that each event is passed, in turn, to each procedure in the chain.

hot spot. The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point*.

HPFS. high-performance file system.

hypergraphic link. A connection between one piece of information and another through the use of graphics.

hypertext. A way of presenting information online with connections between one piece of information and another, called *hypertext links*. See also *hypertext link*.

hypertext link. A connection between one piece of information and another.

I

I/O operation. An input operation to, or output operation from a device attached to a computer.

I-beam pointer. A pointer that indicates an area, such as an entry field in which text can be edited.

icon. In SAA Advanced Common User Access architecture, a graphical representation of an object, consisting of an image, image background, and a label. Icons can represent items (such as a document file) that the user wants to work on, and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions, and minimized programs.

icon area. In the Presentation Manager, the area at the bottom of the screen that is normally used to display the icons for minimized programs.

Icon Editor. The Presentation Manager-provided tool for creating icons.

image font. A set of symbols, each of which is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of one of the symbols. Contrast with *outline font*.

indirect manipulation. Interaction with an object through choices and controls.

information device context. A logical description of a data destination other than the screen (for example, a printer or plotter), but where no output will occur. Its purpose is to satisfy queries. See also *device context*.

information panel. A defined panel type characterized by a body containing only protected information.

Information Presentation Facility (IPF). A facility provided by the OS/2 operating system, by which application developers can produce online documentation and context-sensitive online help panels for their applications.

input focus. (1) The area of a window where user interaction is possible using an input device, such as a mouse or the keyboard. (2) The position in the *active window* where a user's normal interaction with the keyboard will appear.

input router. An internal OS/2 process that removes messages from the system queue.

input/output control. A device-specific command that requests a function of a device driver.

installable file system (IFS). A file system in which software is installed when the operating system is started.

instance. A single occurrence of an object class that has a particular behavior.

instruction pointer. In system/38, a pointer that provides addressability for a machine interface instruction in a program.

integer atom. An *atom* that represents a predefined system constant and carries no storage overhead. For example, names of window classes provided by Presentation Manager are expressed as integer atoms.

interactive graphics. Graphics that can be moved or manipulated by a user at a terminal.

interactive program. (1) A program that is running (active) and is ready to receive (or is receiving) input from a user. (2) A running program that can receive input from the keyboard or another input device. Compare with *active program* and contrast with *noninteractive program*.

Also known as a *foreground program*.

interchange file. A file containing data that can be sent from one Presentation Manager interface application to another.

interpreter. A program that translates and executes each instruction of a high-level programming language before it translates and executes.

interprocess communication (IPC). In the OS/2 operating system, the exchange of information between processes or threads through semaphores, pipes, queues, and shared memory.

interval timer. (1) A timer that provides program interruptions on a program-controlled basis. (2) An electronic counter that counts intervals of time under program control.

IOCtl. Input/output control.

IOPL. Input/output privilege level.

IOPL code segment. An IOPL executable section of programming code that enables an application to directly manipulate hardware interrupts and ports without replacing the device driver. See also *privilege level*.

IPC. Interprocess communication.

IPF. Information Presentation Facility.

IPF compiler. A text compiler that interpret tags in a source file and converts the information into the specified format.

IPF tag language. A markup language that provides the instructions for displaying online information.

item. A data object that can be passed in a DDE transaction.

J

journal. A special-purpose file that is used to record changes made in the system.

K

Kanji. A graphic character set used in Japanese ideographic alphabets.

KBD$. Character-device name reserved for the keyboard.

kernel. The part of an operating system that performs basic functions, such as allocating hardware resources.

kerning. The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

keyboard accelerator. A keystroke that generates a command message for an application.

keyboard augmentation. A function that enables a user to press a keyboard key while pressing a mouse button.

keyboard focus. A temporary attribute of a window. The window that has a keyboard focus receives all keyboard input until the focus changes to a different window.

Keys help. In SAA Common User Access architecture, a help action that provides a listing of the application keys and their assigned functions.

L

label. In a graphics segment, an identifier of one or more elements that is used when editing the segment.

LAN. local area network.

language support procedure. A function provided by the Presentation Manager Interface for applications that do not, or cannot (as in the case of COBOL and FORTRAN programs), provide their own dialog or window procedures.

lazy drag. See *pickup and drop*.

lazy drag set. See *pickup set*.

LDT. In the OS/2 operating system, Local Descriptor Table.

LIFO stack. A stack from which data is retrieved in last-in, first-out order.

linear address. A unique value that identifies the memory object.

linked list. Synonym for *chained list*.

list box. In SAA Advanced Common User Access architecture, a control that contains scrollable choices from which a user can select one choice.

Note: In CUA architecture, this is a programmer term. The end user term is selection list.

list button. A button labeled with an underlined down-arrow that presents a list of valid objects or choices that can be selected for that field.

list panel. A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

load time. The point in time at which a program module is loaded into main storage for execution.

load-on-call. A function of a linkage editor that allows selected segments of the module to be disk resident while other segments are executing. Disk resident segments are loaded for execution and given control when any entry point that they contain is called.

local area network (LAN). (1) A computer network located on a user's premises within a limited geographical area. Communication within a local area network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation. (T)

Note: A LAN does not use store and forward techniques. (2) A network inwhich a set of devices are connected to one another for communication and that can be connected to a larger network.

Local Descriptor Table (LDT). Defines code and data segments specific to a single task.

lock. A serialization mechanism by means of which a resource is restricted for use by the holder of the lock.

logical storage device. A device that the user can map to a physical (actual) device.

LPT1, LPT2, LPT3. Character-device names reserved for parallel printers 1 through 3.

M

main window. The window that is positioned relative to the *desktop window*.

manipulation button. The button on a pointing device a user presses to directly manipulate an object.

map. (1) A set of values having a defined correspondence with the quantities or values of another set. (I) (A) (2) To establish a set of values having a defined correspondence with the quantities or values of another set. (I)

marker box. In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single marker from a marker set.

marker symbol. A symbol centered on a point. Graphs and charts can use marker symbols to indicate the plotted points.

marquee box. The rectangle that appears during a selection technique in which a user selects objects by drawing a box around them with a pointing device.

Master Help Index. In the OS/2 operating system, an alphabetic list of help topics related to using the operating system.

maximize. To enlarge a window to its largest possible size.

media window. The part of the physical device (display, printer, or plotter) on which a picture is presented.

memory block. Part memory within a heap.

memory device context. A logical description of a data destination that is a memory bit map. See also *device context*.

memory management. A feature of the operating system for allocating, sharing, and freeing main storage.

memory object. Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint.

menu. In SAA Advanced Common User Access architecture, an extension of the menu bar that displays a list of choices available for a selected choice in the menu bar. After a user selects a choice in menu bar, the corresponding menu appears. Additional pop-up windows can appear from menu choices.

menu bar. In SAA Advanced Common User Access architecture, the area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus.

menu button. The button on a pointing device that a user presses to view a pop-up menu associated with an object.

message. (1) In the Presentation Manager, a packet of data used for communication between the Presentation Manager interface and Presentation Manager applications (2) In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process.

message box. (1) A dialog window predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. (2) (D of C) In SAA Advanced Common User Access architecture, a type of window that shows messages to users. See also *dialog box, primary window, secondary window*.

message filter. The means of selecting which messages from a specific window will be handled by the application.

message queue. A sequenced collection of messages to be read by the application.

message stream mode. A method of operation in which data is treated as a stream of messages. Contrast with *byte stream*.

metacharacter. See *global file-name character*.

metaclass. The conjunction of an object and its class information; that is, the information pertaining to the class as a whole, rather than to a single instance of the class. Each class is itself an object, which is an instance of the metaclass.

metafile. A file containing a series of attributes that set color, shape and size, usually of a picture or a drawing. Using a program that can interpret these attributes, a user can view the assembled image.

metafile device context. A logical description of a data destination that is a metafile, which is used for graphics interchange. See also *device context*.

metalanguage. A language used to specify another language. For example, data types can be described using a metalanguage so as to make the descriptions independent of any one computer language.

mickey. A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

micro presentation space. A graphics presentation space in which a restricted set of the GPI function calls is available.

minimize. To remove from the screen all windows associated with an application and replace them with an icon that represents the application.

mix. An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as *foreground mix*. Contrast with *background mix*.

mixed character string. A string containing a mixture of one-byte and *Kanji* or Hangeul (two-byte) characters.

mnemonic. (1) A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item. (2) (D of C) In SAA Advanced Common User Access architecture, usually a single character, within the text of a choice, identified by an underscore beneath the character. If all characters in a choice already serve as mnemonics for other choices, another character, placed in parentheses immediately following the choice, can be used. When a user types the mnemonic for a choice, the choice is either selected or the cursor is moved to that choice.

modal dialog box. In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that requires a user to enter information before continuing to work in the application window from which it was displayed. Contrast with *modeless dialog box*. Also known as a *serial dialog box*. Contrast with *parallel dialog box*.

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

model space. See *graphics model space*.

modeless dialog box. In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that allows users to continue their dialog with the application without entering information in the dialog box. Also known as a *parallel dialog box*. Contrast with *modal dialog box*.

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

module definition file. A file that describes the code segments within a load module. For example, it indicates whether a code segment is loadable before module execution begins (preload), or loadable only when referred to at run time (load-on-call).

mouse. In SAA usage, a device that a user moves on a flat surface to position a pointer on the screen. It allows a user to select a choice o function to be performed or to perform operations on the screen, such as dragging or drawing lines from one position to another.

MOUSE$. Character-device name reserved for a mouse.

multiple-choice selection. In SAA Basic Common User Access architecture, a type of field from which a user can select one or more choices or select none. See also *check box*. Contrast with *extended-choice selection*.

multiple-line entry field. In SAA Advanced Common User Access architecture, a control into which a user types more than one line of information. See also *single-line entry field*.

multitasking. The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

mutex semaphore. (Mutual exclusion semaphore). A semaphore that enables threads to serialize their access to resources. Only the thread that currently owns the mutex semaphore can gain access to the resource, thus preventing one thread from interrupting operations being performed by another.

muxwait semaphore. (Multiple wait semaphore). A semaphore that enables a thread to wait either for multiple event semaphores to be posted or for multiple mutex semaphores to be released. Alternatively, a muxwait semaphore can be set to enable a thread to wait for any ONE of the event or mutex semaphores in the muxwait semaphore's list to be posted or released.

N

named pipe. A named buffer that provides client-to-server, server-to-client, or full duplex communication between unrelated processes. Contrast with *unnamed pipe*.

national language support (NLS). The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

nested list. A list that is contained within another list.

NLS. national language support.

non-8.3 file-name format. A file-naming convention in which file names can consist of up to 255 characters. See also *8.3 file-name format*.

noncritical extended attribute. An extended attribute that is not necessary for the function of an application.

nondestructive read. Reading that does not erase the data in the source location. (T)

noninteractive program. A running program that cannot receive input from the keyboard or other input device. Compare with *active program*, and contrast with *interactive program*.

nonretained graphics. Graphic primitives that are not remembered by the Presentation Manager interface when they have been drawn. Contrast with *retained graphics*.

null character (NUL). (1) Character-device name reserved for a nonexistent (dummy) device. (2) (D of C) A control character that is used to accomplish media-fill or time-fill and that may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence; however, the control of equipment or the format may be affected by this character. (I) (A)

null-terminated string. A string of (n+1) characters where the (n+1)th character is the 'null' character (0x00) Also known as 'zero-terminated' string and 'ASCIIZ' string.

O

object. A set of data and actions that can be performed on that data.

Object Interface Definition Language (OIDL). Specification language for SOM class definitions.

object window. A window that does not have a parent but which might have child windows. An object window cannot be presented on a device.

OIDL. Object Interface Definition Language.

open. To start working with a file, directory, or other object.

ordered list. Vertical arrangements of items, with each item in the list preceded by a number or letter.

outline font. A set of symbols, each of which is created as a series of lines and curves. Synonymous with *vector font*. Contrast with *image font*.

output area. An area of storage reserved for output. (A)

owner window. A window into which specific events that occur in another (owned) window are reported.

ownership. The determination of how windows communicate using messages.

owning process. The process that owns the resources that might be shared with other processes.

P

page. (1) A 4KB segment of contiguous physical memory. (2) (D of C) A defined unit of space on a storage medium.

page viewport. A boundary in device coordinates that defines the area of the output device in which graphics are to be displayed. The presentation-page contents are transformed automatically to the page viewport in device space.

paint. (1) The action of drawing or redrawing the contents of a window. (2) In computer graphics, to shade an area of a display image; for example, with crosshatching or color.

panel. In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

panel area. An area within a panel that contains related information. The three major Common User Access-defined panel areas are the action bar, the function key area, and the panel body.

panel area separator. In SAA Basic Common User Access architecture, a solid, dashed, or blank line that provides a visual distinction between two adjacent areas of a panel.

panel body. The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body can contain protected information, selection fields, and entry fields. The layout and content of the panel body determine the panel type.

panel body area. See *client area*.

panel definition. A description of the contents and characteristics of a panel. A panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID. In SAA Basic Common User Access architecture, a panel identifier, located in the upper-left corner of a panel. A user can choose whether to display the panel ID.

panel title. In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

paper size. The size of paper, defined in either standard U.S. or European names (for example, A, B, A4), and measured in inches or millimeters respectively.

parallel dialog box. See *modeless dialog box*.

parameter list. A list of values that provides a means of associating addressability of data defined in a called program with data in the calling program. It contains parameter names and the order in which they are to be associated in the calling and called program.

parent process. In the OS/2 operating system, a process that creates other processes. Contrast with *child process*.

parent window. In the OS/2 operating system, a window that creates a child window. The child window is drawn within the parent window. If the parent window is moved, resized, or destroyed, the child window also will be moved, resized, or destroyed. However, the child window can be moved and resized independently from the parent window, within the boundaries of the parent window. Contrast with *child window*.

partition. (1) A fixed-size division of storage. (2) On an IBM personal computer fixed disk, one of four possible storage areas of variable size; one may be accessed by DOS, and each of the others may be assigned to another operating system.

Paste. A choice in the Edit pull-down that a user selects to move the contents of the clipboard into a preselected location. See also *Copy* and *Cut*.

path. The route used to locate files; the storage location of a file. A fully qualified path lists the drive identifier, directory name, subdirectory name (if any), and file name with the associated extension.

PDD. Physical device driver.

peeking. An action taken by any thread in the process that owns the queue to examine queue elements without removing them.

pel. (1) The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonym for *display point*, *pixel*, and *picture element*. (2) (D of C) Picture element.

physical device driver (PDD). A system interface that handles hardware interrupts and supports a set of input and output functions.

pick. To select part of a displayed object using the pointer.

pickup. To add an object or set of objects to the pickup set.

pickup and drop. A drag operation that does not require the direct manipulation button to be pressed for the duration of the drag.

pickup set. The set of objects that have been picked up as part of a pickup and drop operation.

picture chain. See *segment chain*.

picture element. (1) Synonym for *pel*. (2) (D of C) In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. (T) . (3) The area of the finest detail that can be reproduced effectively on the recording medium.

PID. Process identification.

pipe. (1) A named or unnamed buffer used to pass data between processes. A process reads from or writes to a pipe as if the pipe were a standard-input or standard-output file. See also *named pipe* and *unnamed pipe*. (2) (D of C) To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|).

pixel. (1) Synonym for *pel*. (2) (D of C) Picture element.

plotter. An output unit that directly produces a hardcopy record of data on a removable medium, in the form of a two-dimensional graphic representation. (T)

PM. Presentation Manager.

pointer. (1) The symbol displayed on the screen that is moved by a pointing device, such as a *mouse*. The pointer is used to point at items that users can select. Contrast with *cursor*. (2) A data element that indicates the location of another data element. (T)

POINTER$. Character-device name reserved for a pointer device (mouse screen support).

pointing device. In SAA Advanced Common User Access architecture, an instrument, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

pointings. Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device, such as a *mouse*.

polyfillet. A curve based on a sequence of lines. The curve is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines. See also *fillet*.

polygon. One or more closed figures that can be drawn filled, outlined, or filled and outlined.

polyline. A sequence of adjoining lines.

polymorphism. A concept whereby the behavior of an application object is dependent solely upon the class and contents of the messages received by that object, and is not affected by any other external factor.

pop. To retrieve an item from a last-in-first-out stack of items. Contrast with *push*.

pop-up window. (1) A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window. (2) (D of C) In SAA Advanced Common User Access architecture, a movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request.

presentation drivers. Special purpose I/O routines that handle field device-independent I/O requests from the PM and its applications.

Presentation Manager (PM). The interface of the OS/2 operating system that presents, in windows a graphics-based interface to applications and files installed and running under the OS/2 operating system.

presentation page. The coordinate space in which a picture is assembled for display.

presentation space (PS). (1) Contains the device-independent definition of a picture. (2) (D of C) The display space on a display device.

primary window. In SAA Common User Access architecture, the window in which the main interaction between the user and the application takes place. In a multiprogramming environment, each application starts in its own primary window. The primary window remains for the duration of the application, although the panel displayed will change as the user's dialog moves forward. See also *secondary window*.

primitive. In computer graphics, one of several simple functions for drawing on the screen, including, for example, the rectangle, line, ellipse, polygon, and so on.

primitive attribute. A specifiable characteristic of a graphic primitive. See *graphics attributes*.

print job. The result of sending a document or picture to be printed.

Print Manager. In the Presentation Manager, the part of the spooler that manages the spooling process. It also allows users to view print queues and to manipulate print jobs.

privilege level. A protection level imposed by the hardware architecture of the IBM personal computer. There are four privilege levels (number 0 through 3). Only certain types of programs are allowed to execute at each privilege level. See also *IOPL code segment*.

procedure call. In programming languages, a language construct for invoking execution of a procedure.

process. An instance of an executing application and the resources it is using.

program. A sequence of instructions that a computer can interpret and execute.

program details. Information about a program that is specified in the *Program Manager* window and is used when the program is started.

program group. In the Presentation Manager, several programs that can be acted upon as a single entity.

program name. The full file specification of a program. Contrast with *program title*.

program title. The name of a program as it is listed in the *Program Manager* window. Contrast with *program name*.

prompt. A displayed symbol or message that requests input from the user or gives operational information; for example, on the display screen of an IBM personal computer, the DOS A> prompt. The user must respond to the prompt in order to proceed.

protect mode. A method of program operation that limits or prevents access to certain instructions or areas of storage. Contrast with *real mode*.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I)

pseudocode. An artificial language used to describe computer program algorithms without using the syntax of any particular programming language. (A)

pull-down. (1) An *action bar* extension that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users. (2) (D of C) In SAA Common User Access architecture, pertaining to a choice in an action bar pull-down.

push. To add an item to a last-in-first-out stack of items. Contrast with *pop*.

push button. In SAA Advanced Common User Access architecture, a rectangle with text inside. Push buttons are used in windows for actions that occur immediately when the push button is selected.

putback. To remove an object or set of objects from the lazy drag set. This has the effect of undoing the pickup operation for those objects

putdown. To drop the objects in the lazy drag set on the target object.

Q

queue. (1) A linked list of elements waiting to be processed in FIFO order. For example, a queue may be a list of print jobs waiting to be printed. (2) (D of C) A line or list of items waiting to be processed; for example, work to be performed or messages to be displayed.

queued device context. A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context*.

R

radio button. (1) A control window, shaped like a round button on the screen, that can be in a checked or unchecked state. It is used to select a single item from a list. Contrast with *check box*. (2) In SAA Advanced Common User Access architecture, a circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected.

RAS. Reliability, availability, and serviceability.

raster. (1) In computer graphics, a predetermined pattern of lines that provides uniform coverage of a display space. (T) (2) The coordinate grid that divides the display area of a display device. (A)

read-only file. A file that can be read from but not written to.

real mode. A method of program operation that does not limit or prevent access to any instructions or areas of storage. The operating system loads the entire program into storage and gives the program access to all system resources. Contrast with *protect mode*.

realize. To cause the system to ensure, wherever possible, that the physical color table of a device is set to the closest possible match in the logical color table.

recursive routine. A routine that can call itself, or be called by another routine that was called by the recursive routine.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference phrase. (1) A word or phrase that is emphasized in a device-dependent manner to inform the user that additional information for the word or phrase is available. (2) (D of C) In hypertext, text that is highlighted and preceded by a single-character input field used to signify the existence of a hypertext link.

reference phrase help. In SAA Common User Access architecture, highlighted words or phrases within help information that a user selects to get additional information.

refresh. To update a window, with changed information, to its current status.

region. A clipping boundary in device space.

register. A part of internal storage having a specified storage capacity and usually intended for a specific purpose. (T)

remote file system. A file-system driver that gains access to a remote system without a block device driver.

resource. The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators, and mnemonics; the definitions are held in a resource file.

resource file. A file containing information used in the definition of a window. Definitions can be of fonts, templates, accelerators, and mnemonics.

restore. To return a window to its original size or position following a sizing or moving action.

retained graphics. Graphic primitives that are remembered by the Presentation Manager interface after they have been drawn. Contrast with *nonretained graphics*.

return code. (1) A value returned to a program to indicate the results of an operation requested by that program. (2) A code used to influence the execution of succeeding instructions.(A)

reverse video. (1) A form of highlighting a character, field, or cursor by reversing the color of the character, field, or cursor with its background; for example, changing a red character on a black background to a black character on a red background. (2) In SAA Basic Common User Access architecture, a screen emphasis feature that interchanges the foreground and background colors of an item.

REXX Language. Restructured Extended Executor. A procedural language that provides batch language functions along with structured programming constructs such as loops; conditional testing and subroutines.

RGB. (1) Color coding in which the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light. (2) Pertaining to a color display that accepts signals representing red, green, and blue.

roman. Relating to a type style with upright characters.

root segment. In a hierarchical database, the highest segment in the tree structure.

round-robin scheduling. A process that allows each thread to run for a specified amount of time.

run time. (1) Any instant at which the execution of a particular computer program takes place. (T) (2) The amount of time needed for the execution of a particular computer program. (T) (3) The time during which an instruction in an instruction register is decoded and performed. Synonym for *execution time*.

S

SAA. Systems Application Architecture.

SBCS. Single-byte character set.

scheduler. A computer program designed to perform functions such as scheduling, initiation, and termination of jobs.

screen. In SAA Basic Common User Access architecture, the physical surface of a display device upon which information is shown to a user.

screen device context. A logical description of a data destination that is a particular window on the screen. See also *device context*.

SCREEN$. Character-device name reserved for the display screen.

scroll bar. In SAA Advanced Common User Access architecture, a part of a window, associated with a scrollable area, that a user interacts with to see information that is not currently allows visible.

scrollable entry field. An entry field larger than the visible field.

scrollable selection field. A selection field that contains more choices than are visible.

scrolling. Moving a display image vertically or horizontally in a manner such that new data appears at one edge, as existing data disappears at the opposite edge.

secondary window. A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

sector. On disk or diskette storage, an addressable subdivision of a track used to record one block of a program or data.

segment. See *graphics segment*.

segment attributes. Attributes that apply to the segment as an entity, as opposed to the individual primitives within the segment. For example, the visibility or detectability of a segment.

segment chain. All segments in a graphics presentation space that are defined with the 'chained' attribute. Synonym for *picture chain*.

segment priority. The order in which segments are drawn.

segment store. An area in a normal graphics presentation space where retained graphics segments are stored.

select. To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing.

select button. The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor. In SAA Advanced Common User Access architecture, a visual indication that a user has selected a choice. It is represented by outlining the choice with a dotted box. See also *text cursor*.

selection field. (1) In SAA Advanced Common User Access architecture, a set of related choices. See also *entry field*. (2) In SAA Basic Common User Access architecture, an area of a panel that cannot be scrolled and contains a fixed number of choices.

semantics. The relationships between symbols and their meanings.

semaphore. An object used by applications for signalling purposes and for controlling access to serially reusable resources.

separator. In SAA Advanced Common User Access architecture, a line or color boundary that provides a visual distinction between two adjacent areas.

serial dialog box. See *modal dialog box*.

serialization. The consecutive ordering of items.

serialize. To ensure that one or more events occur in a specified sequence.

serially reusable resource (SRR). A logical resource or object that can be accessed by only one task at a time.

session. (1) A routing mechanism for user interaction via the console; a complete environment that determines how an application runs and how users interact with the application. OS/2 can manage more than one session at a time, and more than one process can run in a session. Each session has its own set of environment variables that determine where OS/2 looks for dynamic-link libraries and other important files. (2) (D of C) In the OS/2 operating system, one instance of a started program or command prompt. Each session is separate from all other sessions that might be running on the computer. The operating system is responsible for coordinating the resources that each session uses, such as computer memory, allocation of processor time, and windows on the screen.

Settings Notebook. A control window that is used to display the settings for an object and to enable the user to change them.

shadow box. The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shared data. Data that is used by two or more programs.

shared memory. In the OS/2 operating system, a segment that can be used by more than one program.

shear. In computer graphics, the forward or backward slant of a graphics symbol or string of such symbols relative to a line perpendicular to the baseline of the symbol.

shell. (1) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens, and communicate them to the operating system. (2) Software that allows a kernel program to run under different operating-system environments.

shutdown. The process of ending operation of a system or a subsystem, following a defined procedure.

sibling processes. Child processes that have the same parent process.

sibling windows. Child windows that have the same parent window.

simple list. A list of like values; for example, a list of user names. Contrast with *mixed list*.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set*.

slider box. In SAA Advanced Common User Access architecture: a part of the scroll bar that shows the position and size of the visible information in a window relative to the total amount of information available. Also known as *thumb mark*.

SOM. System Object Model.

source file. A file that contains source statements for items such as high-level language programs and data description specifications.

source statement. A statement written in a programming language.

specific dynamic-link module. A dynamic-link module created for the exclusive use of an application.

spin button. In SAA Advanced Common User Access architecture, a type of entry field that shows a scrollable ring of choices from which a user can select a choice. After the last choice is displayed, the first choice is displayed again. A user can also type a choice from the scrollable ring into the entry field without interacting with the spin button.

spline. A sequence of one or more Bézier curves.

spooler. A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete and the required device is available. The spooler prevents output from different sources from being intermixed.

stack. A list constructed and maintained so that the next data element to be retrieved is the most recently stored. This method is characterized as last-in-first-out (LIFO).

standard window. A collection of window elements that form a panel. The standard window can include one or more of the following window elements: sizing borders, system menu icon, title bar, maximize/minimize/restore icons, action bar and pull-downs, scroll bars, and client area.

static control. The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

static storage. (1) A read/write storage unit in which data is retained in the absence of control signals. (A) Static storage may use dynamic addressing or sensing circuits. (2) Storage other than *dynamic storage*. (A)

style. See *window style*.

subdirectory. In an IBM personal computer, a file referred to in a root directory that contains the names of other files stored on the diskette or fixed disk.

swapping. (1) A process that interchanges the contents of an area of real storage with the contents of an area in auxiliary storage. (I) (A) (2) In a system with virtual storage, a paging technique that writes the active pages of a job to auxiliary storage and reads pages of another job from auxiliary storage into real storage. (3) The process of temporarily removing an active job from main storage, saving it on disk, and processing another job in the area of main storage formerly occupied by the first job.

switch. (1) In SAA usage, to move the cursor from one point of interest to another; for example, to move from one screen or window to another or from a place within a displayed image to another place on the same displayed image. (2) In a computer program, a conditional instruction and an indicator to be interrogated by that instruction. (3) A device or programming technique for making a selection, for example, a toggle, a conditional jump.

switch list. See *Task List*.

symbolic identifier. A text string that equates to an integer value in an include file, which is used to identify a programming object.

symbols. In Information Presentation Facility, a document element used to produce characters that cannot be entered from the keyboard.

synchronous. Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals. (T) See also *asynchronous*.

System Menu. In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

System Object Model (SOM). A mechanism for language-neutral, object-oriented programming in the OS/2 environment.

system queue. The master queue for all pointer device or keyboard events.

system-defined messages. Messages that control the operations of applications and provides input an other information for applications to process.

Systems Application Architecture (SAA). A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

T

table tags. In Information Presentation Facility, a document element that formats text in an arrangement of rows and columns.

tag. (1) One or more characters attached to a set of data that contain information about the set, including its identification. (I) (A) (2) In Generalized Markup Language markup, a name for a type of document or document element that is entered in the source document to identify it.

target object. An object to which the user is transferring information.

Task List. In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

template. An ASCII-text definition of an action bar and pull-down menu, held in a resource file, or as a data structure in program memory.

terminate-and-stay-resident (TSR). Pertaining to an application that modifies an operating system interrupt vector to point to its own location (known as hooking an interrupt).

text. Characters or symbols.

text cursor. A symbol displayed in an entry field that indicates where typed input will appear.

text window. Also known as the VIO window.

text-windowed application. The environment in which the operating system performs advanced-video input and output operations.

thread. A unit of execution within a process. It uses the resources of the process.

thumb mark. The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box*.

thunk. Term used to describe the process of address conversion, stack and structure realignment, etc., necessary when passing control between 16-bit and 32-bit modules.

tilde. A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

time slice. (1) An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing-unit time is allocated to another task, so a task cannot monopolize processing-unit time beyond a fixed limit. (2) In systems with time sharing, a segment of time allocated to a terminal job.

time-critical process. A process that must be performed within a specified time after an event has occurred.

timer. A facility provided under the Presentation Manager, whereby Presentation Manager will dispatch a message of class WM_TIMER to a particular window at specified intervals. This capability may be used by an application to perform a specific processing task at predetermined intervals, without the necessity for the application to explicitly keep track of the passage of time.

timer tick. See *clock tick*.

title bar. In SAA Advanced Common User Access architecture, the area at the top of each window that contains the window title and system menu icon. When appropriate, it also contains the minimize, maximize, and restore icons. Contrast with *panel title*.

TLB. Translation lookaside buffer.

transaction. An exchange between a workstation and another device that accomplishes a particular action or result.

transform. (1) The action of modifying a picture by scaling, shearing, reflecting, rotating, or translating. (2) The object that performs or defines such a modification; also referred to as a *transformation*.

Translation lookaside buffer (TLB). A hardware-based address caching mechanism for paging information.

Tree. In the Presentation Manager, the window in the *File Manager* that shows the organization of drives and directories.

truncate. (1) To terminate a computational process in accordance with some rule (A) (2) To remove the beginning or ending elements of a string. (3) To drop data that cannot be printed or displayed in the line width specified or available. (4) To shorten a field or statement to a specified length.

TSR. Terminate-and-stay-resident.

unnamed pipe. A circular buffer, created in memory, used by related processes to communicate with one another. Contrast with *named pipe*.

unordered list. In Information Presentation Facility, a vertical arrangement of items in a list, with each item in the list preceded by a special character or bullet.

update region. A system-provided area of dynamic storage containing one or more (not necessarily contiguous) rectangular areas of a window that are visually invalid or incorrect, and therefore are in need of repainting.

user interface. Hardware, software, or both that allows a user to interact with and perform operations on a system, program, or device.

User Shell. A component of OS/2 that uses a graphics-based, windowed interface to allow the user to manage applications and files installed and running under OS/2.

utility program. (1) A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program. (T) (2) A program designed to perform an everyday task such as copying data from one storage device to another. (A)

U

There are no glossary terms for this starting letter.

V

value set control. A visual component that enables a user to select one choice from a group of mutually exclusive choices.

vector font. A set of symbols, each of which is created as a series of lines and curves. Synonymous with *outline font*. Contrast with *image font*.

VGA. Video graphics array.

viewing pipeline. The series of transformations applied to a graphic object to map the object to the device on which it is to be presented.

viewing window. A clipping boundary that defines the visible part of model space.

VIO. Video Input/Output.

virtual memory (VM). Synonymous with *virtual storage*.

virtual storage. (1) The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of auxiliary storage available, not by the actual number of main storage locations. (I) (A) (2) Addressable space that is apparent to the user as the processor storage space, from which the instructions and the data are mapped into the processor storage locations. (3) Synonymous with *virtual memory*.

visible region. A window's presentation space, clipped to the boundary of the window and the boundaries of any overlying window.

volume. (1) A file-system driver that uses a block device driver for input and output operations to a local or remote device. (I) (2) A portion of data, together with its data carrier, that can be handled conveniently as a unit.

W

wildcard character. Synonymous with *global file-name character*.

window. (1) A portion of a display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A) (2) An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen.

window class. The grouping of windows whose processing needs conform to the services provided by one window procedure.

window coordinates. A set of coordinates by which a window position or size is defined; measured in device units, or *pels*.

window handle. Unique identifier of a window, generated by Presentation Manager when the window is created, and used by applications to direct messages to the window.

window procedure. Code that is activated in response to a message. The procedure controls the appearance and behavior of its associated windows.

window rectangle. The means by which the size and position of a window is described in relation to the desktop window.

window resource. A read-only data segment stored in the .EXE file of an application o the .DLL file of a dynamic link library.

window style. The set of properties that influence how events related to a particular window will be processed.

window title. In SAA Advanced Common User Access architecture, the area in the title bar that contains the name of the application and the OS/2 operating system file name, if applicable.

workstation. (1) A display screen together with attachments such as a keyboard, a local copy device, or a tablet. (2) (D of C) One or more programmable or nonprogrammable devices that allow a user to do work.

world coordinates. A device-independent Cartesian coordinate system used by the application program for specifying graphical input and output. (I) (A)

world-coordinate space. Coordinate space in which graphics are defined before transformations are applied.

WYSIWYG. What-You-See-Is-What-You-Get. A capability of a text editor to continually display pages exactly as they will be printed.

X

There are no glossary terms for this starting letter.

Y

There are no glossary terms for this starting letter.

Z

z-order. The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

zooming. The progressive scaling of an entire display image in order to give the visual impression of movement of all or part of a display group toward or away from an observer. (I) (A)

8.3 file-name format. A file-naming convention in which file names are limited to eight characters before and three characters after a single dot. Usually pronounced “eight-dot-three.” See also *non-8.3 file-name format*.

Index

25H7103
G25H-7103-00

Printed on recycled paper.

Printed in U.S.A.

G25H-7103-00

P25H7103