XXI CHAPTER

Windows

Microsoft Windows is a graphical environment that runs atop MS-DOS on IBM-compatible PCs. Since the introduction of version 3.0 in 1990, Windows has boomed as a standard for PC programs.

As a C programmer writing applications under Windows, you will face many challenges because it is unlike any environment you have ever encountered. This chapter will present several questions you might have regarding programming in the Windows environment and will attempt to provide concise answers to these questions. However, you should not rely on this chapter alone for your technical information when working with Windows.

Instead, if you haven't done so already, you should rush out and get a copy of *Programming Windows 3.1* by Charles Petzold. This is the definitive reference and tutorial regarding Windows programming. Petzold starts the book with brief discussions regarding the background of Windows and then takes you step-by-step through the details of Windows programming. If you really want to learn how to write Microsoft Windows programs in C, you need this book. Another good book is *Teach Yourself Windows Programming in 21 Days*, from Sams Publishing.

Another reference that will come in handy when you're learning Windows programming is the *Microsoft Developer Network CD*. This is a CD-ROM that comes packed with samples from all sources. Many of the sample programs can be cut and pasted directly into your compiler.

If you have briefly studied Windows, this chapter should help answer some of your recurring questions.

XXI.1: Can *printf()* be used in a Windows program?

Answer:

The standard C function `printf()` can be used in a Microsoft Windows program; however, it has no usefulness to the Windows environment, and it does not produce the same result as in a DOS environment. The `printf()` function directs program output to `stdout`, the standard output device. Under DOS, the standard output device is the user's screen, and the output of a `printf()` statement under a DOS program is immediately displayed.

Conversely, Microsoft Windows is an *operating environment* that runs on top of DOS, and it has its own mechanism for displaying program output to the screen. This mechanism comes in the form of what is called a *device context*. A device context is simply a handle to a portion of the screen that is handled by your program. The only way to display output on-screen in the Microsoft Windows environment is for your program to obtain a handle to a device context. This task can be accomplished with several of the Windows SDK (Software Development Kit) functions. For instance, if you were to display the string `"Hello from Windows!"` in a windows program, you would need the following portion of code:

```
void print_output(void)
{
     ...
     hdcDeviceContext = BeginPaint(hwndWindow, psPaintStruct);

     DrawText(hdcDeviceContext, "Hello from Windows!", -1,
              &rectClientRect, DT_SINGLELINE);
     ...
}
```

Put simply, all output from a Windows program must be funnelled through functions provided by the Windows API. If you were to put the line

```
printf("Hello from Windows!");
```

into the preceding example, the output would simply be ignored by Windows, and your program would appear to print nothing. The output is ignored because `printf()` outputs to `stdout`, which is not defined under Windows. Therefore, any C function such as `printf()` (or any other function that outputs to `stdout`) under the Windows environment is rendered useless and should ultimately be avoided.

Note, however, that the standard C function `sprintf()`, which prints formatted output to a string, is permissible under the Windows environment. This is because all output from the `sprintf()` function goes directly to the string and not to `stdout`.

Cross Reference:

XXI.9: What is the difference between Windows functions and standard DOS functions?

XXI.2: How do you create a delay timer in a Windows program?

Answer:

You can create a delay timer in a Windows program by using the Windows API function `SetTimer()`. The `SetTimer()` function sets up a timer event in Windows to be triggered periodically at an interval that you specify. To create a timer, put the following code in your `WinMain()` function:

```
SetTimer(hwnd, 1, 1000, NULL);
```

This code sets up a timer in Windows. Now, every 1000 clock ticks (1 second), your program receives a `WM_TIMER` message. This message can be trapped in your `WndProc` (message loop) function as shown here:

```
switch (message)
{

          case WM_TIMER :

               /* this code is called in one-second intervals */

               return 0 ;

}
```

You can put whatever you like in the `WM_TIMER` section of your program. For instance, FAQ XXI.23 shows how you might display the date and time in a window's title bar every second to give your users a constant update on the current day and time. Or perhaps you would like your program to periodically remind you to save your work to a file. Whatever the case, a delay timer under Windows can be very handy.

To remove the timer, call the `KillTimer()` function. When you call this function, pass it the handle to the window the timer is attached to and your own timer identifier. In the preceding `SetTimer()` example, the number 1 is used as the timer identifier. To stop this particular timer, you would issue the following function call:

```
KillTimer(hwnd, 1);
```

Cross Reference:

None.

XXI.3: What is a handle?

Answer:

A handle under Windows is much like a handle used to refer to a file in C. It is simply a numeric representation of an object. Under Windows, a handle can refer to a brush (the object that paints the screen), a cursor, an icon, a window, a device context (the object that is used to output to your screen or printer), and many other objects. The handle assigned to an object is used to reference it when calling other Windows functions.

Handle numbers are assigned by Windows, usually when a Windows API function call is made in your program. Typically, variables that represent handles are prefixed with the letter `h` and a mnemonic representation of the object they refer to. For instance, to create a window, you might make the following Windows API call:

```
hwndSample =
     CreateWindow(szApplicationName,     /* Window class name */
                  "FAQ Sample Program", /* Caption for title bar */
                  WS_OVERLAPPEDWINDOW,  /* Style of window */
                  CW_USEDEFAULT,        /* Initial x position */
                  CW_USEDEFAULT,        /* Initial y position */
                  CW_USEDEFAULT,        /* Initial x size */
                  CW_USEDEFAULT,        /* Initial y size */
                  NULL,        /* Window handle of parent window */
                  NULL,        /* Menu handle for this window */
                  hInstance,   /* Instance handle */
                  NULL) ;      /* Other window parameters */
```

The Windows API function `CreateWindow()` is used to create an instance of a window on the screen. As you can see from the preceding example, it returns a handle to the window that is created. Whenever this window is referenced from this point on, the handle variable `hwndSample` is used. Windows keeps track of handle numbers internally, so it can dereference the handle number whenever you make a Windows API function call.

Cross Reference:

None.

XXI.4: How do you interrupt a Windows program?

Answer:

As a user of Microsoft Windows, you might already know that you can terminate a Windows program in many ways. Here are just a few methods:

- Choose **F**ile | E**x**it from the pull-down menu.
- Choose **C**lose from the control box menu (-) located to the left of the title bar.
- Double-click the mouse on the control box.
- Press Ctrl-Alt-Delete.
- Choose **E**nd Task from the Windows Task Manager.
- Exit Windows.

This list includes the more typical ways users exit their Windows programs. As a Windows developer, how can you provide a way for users to interrupt your program?

If you have used many DOS programs, you might remember the key combination Ctrl-Break. This combination was often used to break out of programs that were hung up or that you could not figure a way to get out of. Often, DOS programs would not trap the Ctrl-Break combination, and the program would be aborted. DOS programs could optionally check for this key combination and prevent users from breaking out of programs by pressing Ctrl-Break.

Under Windows, the Ctrl-Break sequence is translated into the virtual key VK_CANCEL (see FAQ XXI.17 for an explanation of virtual keys). One way you can trap for the Ctrl-Break sequence in your Windows program is to insert the following code into your event (message) loop:

```
...

switch (message)
{

     case WM_KEYDOWN:

          if (wParam == VK_CANCEL)
          {

              /* perform code to cancel or
                 interrupt the current process */

          }

}

...
```

In the preceding example program, if the wParam parameter is equal to the virtual key code VK_CANCEL, you know that the user has pressed Ctrl-Break. This way, you can query the user as to whether he wants to cancel the current operation. This feature comes in handy when you are doing long batch processes such as printing reports.

Cross Reference:

None.

XXI.5: What is the GDI and how is it accessed?

Answer:

GDI stands for Graphic Device Interface. The GDI is a set of functions located in a dynamic link library (named GDI.EXE, in your Windows system directory) that are used to support device-independent graphics output on your screen or printer. Through the GDI, your program can be run on any PC that supports Windows. The GDI is implemented at a high level, so your programs are sheltered from the complexities of dealing with different output devices. You simply make GDI calls, and Windows works with your graphics or printer driver to ensure that the output is what you intended.

The gateway to the GDI is through something called a Device Context. A device context handle is simply a numeric representation of a device context (that is, a GDI-supported object). By using the device context handle, you can instruct Windows to manipulate and draw objects on-screen. For instance, the following portion of code is used to obtain a device context handle from Windows and draw a rectangle on-screen:

```
long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
                                 UINT wParam, LONG lParam)
{

     HDC             hdcOutput;
```

```
    PAINTSTRUCT     psPaint;
    HPEN            hpenDraw;

    ...

    switch(message)
    {

        ...

        case WM_PAINT:

            hdcOutput = BeginPaint(hwndMyWindow, &psPaint);

            hpenDraw = CreatePen(PS_SOLID, 3, 0L);

            SelectObject(hdcOutput, hpenDraw);

            Rectangle(hdcOutput, 0, 0, 150, 150);

            TextOut(hdcOutput, 200, 200,
                    "Just the FAQ's, Ma'am...", 24);

        ...

    }

...

}
```

In the preceding program, the `BeginPaint()` function prepares the current window to be painted with graphics objects. The `CreatePen()` function creates a pen object of a specified style, width, and color. The pen is used to paint objects on-screen. The `SelectObject()` function selects the GDI object you want to work with. After these setup functions are called, the `Rectangle()` function is called to create a rectangle in the window, and the `TextOut()` function is called to print text to the window.

Cross Reference:

None.

XXI.6: Why is windows.h important?

Answer:

The windows.h header file contains all the definitions and declarations used within the Windows environment. For instance, all system color constants (see FAQ XXI.25) are defined in this header file. Additionally, all Windows-based structures are defined here. Each Windows API function is also declared in this header.

No Windows program can be created without the inclusion of the windows.h header file. This is because all Windows API functions have their declarations in this file, and without this file, your program will probably receive a warning or error message that there is no declaration for the Windows function you are

calling. All Windows-based structures, such as HDC and PAINTSTRUCT, are defined in the windows.h header file. You therefore will get compiler errors when you try to use any Windows-based structures in your program without including the windows.h file. Additionally, Windows contains numerous symbolic constants that are used throughout Windows programs. Each of these constants is defined in the windows.h header file.

Thus, the windows.h header file is extremely important, and no Windows program can exist without it. It is roughly equivalent (regarding the rules of inclusion) to the standard stdio.h file that you always include in any DOS-based C program. Not including the file can bring several compiler warnings and errors.

Cross Reference:

XXI.7: What is the Windows SDK?

XXI.8: Do you need Microsoft's Windows SDK to write Windows programs?

XXI.7: What is the Windows SDK?

Answer:

The Windows SDK (Software Development Kit) is a set of resources (software and manuals) that are available to the C programmer to construct Windows programs with. The Windows SDK includes all the Windows API function libraries, thus enabling you to link your C programs with the Windows API functions. It also includes a handful of useful utilities such as the Image Editor (for creating and modifying icons, bitmaps, and so on). It includes the executable file WINSTUB.EXE, which is linked with each Windows program to notify users that the executable is a Windows program. For instance, if you have ever tried running a Windows program from the DOS prompt, you probably have seen one of the following messages (or something similar):

```
This program requires Microsoft Windows.

This program must be run under Microsoft Windows.
```

The WINSTUB.EXE program automatically prints this message every time someone tries to run a Windows executable from a non-Windows environment. Note that the WINSTUB.EXE program is not separated, but rather is embedded into your Windows executable. It is transparent to you and the users of your programs.

The Windows SDK also includes extensive printed documentation of each Windows API function call. This documentation comes in handy when you are writing Windows programs. The *Programmer's Reference Guide* details how to use each Windows API function.

With all the utilities, libraries, and documentation included with the Windows SDK, you might be inclined to think that the Windows SDK is required in order to produce Windows-based programs. See the next FAQ for a response.

Cross Reference:

XXI.6: Why is windows.h important?

XXI.8: Do you need Microsoft's Windows SDK to write Windows programs?

XXI.8: Do you need Microsoft's Windows SDK to write Windows programs?

Answer:

No. You do not need to purchase the Windows SDK from Microsoft to produce Windows programs—instead, most of today's compilers include the Windows libraries, utilities, and online documentation that is replicated in the SDK.

When Windows was first introduced, Microsoft was the only vendor that had a C compiler capable of producing Windows programs. Simply purchasing the Microsoft C compiler did not enable you to create Windows programs, however. Instead, you were required to purchase the Windows SDK from Microsoft to allow your Microsoft C compiler to create your Windows programs.

With the advent of Borland C++ in 1990, the SDK was no longer required. This is because Borland licensed the Windows libraries from Microsoft so that the developers who used Borland C++ did not need to purchase the Windows SDK. Borland also included its own Windows utilities and documentation so that a developer would be fully equipped to write Windows applications. This, in effect, started a revolution. From this point on, compiler vendors typically licensed the Windows API libraries and included them in their compilers. Even Microsoft, pressured by competition, dropped the SDK "requirement" with the introduction of Microsoft C/C++ 7.0.

Today, you can purchase pretty much any compiler that is Windows-capable without having to spend hundreds of extra dollars to buy the Windows SDK. Instead, you will find that your Windows-based compiler comes with all the necessary libraries, utilities, and documentation. In most cases, the Windows API documentation is provided online and not in hard copy to save distribution expenses.

Cross Reference:

XXI.6: Why is windows.h important?

XXI.7: What is the Windows SDK?

XXI.9: What is the difference between Windows functions and standard DOS functions?

Answer:

Unlike most DOS functions, Windows functions are always declared as `FAR PASCAL`. The `FAR` keyword signifies that the Windows API function is located in a different code segment than your program. All Windows API function calls are declared as `FAR`, because all the Windows functions are located in dynamic link libraries and must be loaded at runtime into a different code segment than the one you are running your program in.

The `PASCAL` keyword signifies that the pascal calling convention is used. The pascal calling convention is slightly more efficient than the default C calling convention. With regular non-pascal function calls, the

parameters are pushed on the stack from right to left beginning with the last parameter. The code calling the function is responsible for adjusting the stack pointer after the function returns. With the pascal calling sequence, the parameters are pushed on the stack from left to right, and the called function cleans up the stack. This method results in greater efficiency.

Note that the capitalized words FAR and PASCAL are really uppercase representations of their lowercase keywords, far and pascal. Windows simply #defines them as uppercase to comply with notation rules. Also note that DOS functions can optionally be declared as far pascal—this is perfectly legal. However, under Windows, all API functions are FAR PASCAL. This is not an option, but a mandatory requirement of the Windows environment.

Cross Reference:

XXI.1: Can printf() be used in a Windows program?

XXI.10: What is dynamic linking?

Answer:

All Windows programs communicate with the Windows kernel through a process known as dynamic linking. Normally, with DOS programs, your programs are linked statically. This means that your linker resolves all unresolved external function calls by pulling in the necessary object code modules (.OBJs) to form an executable file (.EXE) that contains the executable code for all functions called within your program.

The Windows environment, on the other hand, provides too many functions to be linked statically into one executable program. A statically linked program under Windows would probably be several megabytes in size and horribly inefficient. Instead, Windows makes extensive use of dynamic link libraries. Dynamic link libraries (.DLLs) are somewhat like the C libraries you create under DOS, with the exception that DLLs can be loaded dynamically at runtime and do not have to be linked in statically at link time. This method has several advantages. First, your Windows executables are typically small, relying on calls to DLLs to provide runtime support. Second, Windows can load and discard DLLs on demand—which allows Windows to fine-tune its environment at runtime. Windows can make room for more programs if it can dynamically discard functions that are not being used currently.

How does dynamic linking work? It is not an easy process by any means. First of all, when you link your Windows program, your compiler creates a table in your executable file that contains the name of the dynamic link library referenced and an ordinal number or name that represents that function in that dynamic link library. At runtime, when you reference a function call that is located in a dynamic link library, that DLL is loaded into memory, and the function's entry point is retrieved from your executable's DLL table. When the DLL is no longer needed, it is unloaded to make room for other programs.

Dynamic link libraries are typically used for large programs with many functions. You can create a DLL with your compiler—see your compiler's documentation for specific instructions on how to carry out this task.

Cross Reference:

None.

XXI.11: What are the differences among *HANDLE*, *HWND*, and *HDC*?

Answer:

Under Windows, the symbolic names HANDLE, HWND, and HDC have different meanings, as presented in Table XXI.11.

Table XXI.11. Symbolic names and their meanings.

Symbolic Name	*Meaning*
HANDLE	Generic symbolic name for a handle
HWND	Handle to a window
HDC	Handle to a device context

It is a Windows standard to make symbolic names uppercase. As FAQ XXI.3 explains, a handle under Windows is simply a numeric reference to an object. Windows keeps track of all objects through the use of handles. Because window objects and device context objects are used quite often under Windows, they have their own handle identifier names (HWND for window and HDC for device context). Many other standard handle names exist under Windows, such as HBRUSH (handle to a brush), HCURSOR (handle to a cursor), and HICON (handle to an icon).

Cross Reference:

None.

XXI.12: Are Windows programs compatible from one compiler to the next?

Answer:

All compilers available for development of Microsoft Windows programs *must* support the Microsoft Windows SDK (Software Development Kit), and therefore the Windows functions you use in your programs are compatible across compilers. A typical Windows program developed in standard C using only Windows API calls should compile cleanly for all Windows-compatible compilers. The functions provided in the Windows API are compiler-independent and easy to port between compilers such as Borland C++, Microsoft Visual C++, and Symantec C++.

Most of the Windows-based programs on the market today, however, use C++ class libraries to augment and simplify the complexity of using the Windows SDK. Some class libraries, such as Microsoft's Foundation Class Library (MFC) and Borland's ObjectWindows Library (OWL), are *compiler-specific*. This means that you cannot take a Windows program developed with MFC using Microsoft's Visual C++ and port it to

Borland C++, nor can you take a Windows program developed with OWL using Borland C++ and port it to Visual C++. Some class libraries, such as zApp and Zinc, are compiler-independent and are thus safer to use when multiple compilers must be supported.

Note that if you are using C++ for your Windows development, you should pay close attention to your compiler's adherence to ANSI-standard C++, because there are different levels of support for ANSI C++ between compilers. For instance, some compilers have full support for C++ *templates*, whereas others do not. If you were to write a Windows program using templates, you might have a hard time porting your code from one compiler to another.

Typically, though, if you are developing with ANSI-standard C and the Microsoft Windows API, your code should be 100 percent portable to any other Windows-compatible compiler.

Cross Reference:

None.

XXI.13: Will Windows always save and refresh your program's windows?

Answer:

No. Windows itself is not responsible for saving and restoring your program's windows. Instead, Windows sends a message to your program—the `WM_PAINT` message—that notifies your program that its client area needs repainting.

The client area refers to the portion of the screen that your program is in control of—that is, the portion of the screen occupied by your program's windows. Whenever another program overlays your program with another window, your client area is covered by that application's window. When that application's window is removed, Windows sends a `WM_PAINT` message to your program. Your Windows program should contain an event loop that looks for and responds to such messages. When your program receives the `WM_PAINT` message, you are responsible for initiating the appropriate code to repaint your application's window.

The `WM_PAINT` message is generated by Windows when one of the following events occurs:

- A previously hidden portion of your program's client area is uncovered.
- Your application's window is moved or resized.
- Your application's window is scrolled by the use of a scrollbar.
- A pull-down or pop-up menu is invoked.

Additionally, you can force your program to repaint the screen (thus generating a `WM_PAINT` message to your own program) by calling the Windows API function `InvalidateRect()`.

Your program should contain an event loop that captures incoming messages and responds to them. Here is an example of a typical event loop that responds to a `WM_PAINT` message:

```
switch(message)
{
```

```
    ...
    case WM_PAINT:
        hdcOutput = BeginPaint(hwndMyWindow, &psPaint);
        hpenDraw = CreatePen(PS_SOLID, 3, 0L);
        SelectObject(hdcOutput, hpenDraw);
        Rectangle(hdcOutput, 0, 0, 150, 150);
        TextOut(hdcOutput, 200, 200, "Just the FAQ's, Ma'am...", 24);
    ...
}
```

When the preceding program is run, a `WM_PAINT` message is generated by Windows on program start-up and any time the client area is moved, resized, or scrolled.

It should be noted that actions such as cursor movement and drag-and-drop operations do not require a `WM_PAINT` message to be generated. In these cases, Windows saves and restores the portion of the screen that has been covered with the cursor or icon.

Cross Reference:

XXI.14: How do you determine a Windows program's client area size?

XXI.14: How do you determine a Windows program's client area size?

Answer:

Your program's client area size is defined as the height and width of your program's window that is displayed on-screen. The client area size can be determined by processing the `WM_SIZE` message in your program's event loop. The `WM_SIZE` message contains three parameters, two of which can be used to determine your client area size. Your program's event loop (window procedure) is passed a parameter named `lParam` that can be evaluated when a `WM_SIZE` message is received. The low-order word of the `lParam` variable contains your program's client area width, and the high-order word of the `lParam` variable contains your program's client area height. Here is an example of determining client area size:

```
switch (message)
{
    ...
    case WM_SIZE:
        nProgramWidth = LOWORD(lParam);
```

```
            nProgramHeight = HIWORD(lParam);
        ...
}
```

`LOWORD` and `HIWORD` are actually macros defined in windows.h that extract the low-order and high-order words, respectively.

Cross Reference:

XXI.20: Can a mouse click be captured in an area outside your program's client area?

XXI.15: What are OEM key codes?

Answer:

The OEM (Original Equipment Manufacturer) key codes refer to the original 255 characters preprogrammed into all IBM-compatible ROMs—everything from hex 00 to hex FF. These characters not only represent the uppercase and lowercase characters of the alphabet, but also contain several nonprintable characters (tab, bell, carriage return, linefeed, and such) and several "graphical" characters used for line drawing. This character set also contains some symbols for representing fractions, pi, infinity, and others. Many DOS-based programs use this character set to print graphics on-screen, because these 255 characters are the only ones available for DOS to use.

Cross Reference:

XXI.16: Should a Windows program care about the OEM key codes?

XXI.17: What are virtual key codes?

XXI.18: What is a dead key?

XXI.16: Should a Windows program care about the OEM key codes?

Answer:

No. As FAQ XXI.15 explains, OEM key codes refer to the original 255 characters of the IBM character set that comes preprogrammed into every 80x86 ROM.

Many of these characters were used in older DOS-based programs to represent characters that normally would have required graphics. Because Windows is a graphical environment that contains hundreds of functions for creating graphical objects, these characters are no longer needed. Instead of writing Windows functions to use the OEM character set to draw a rectangle, for instance, you can simply call the Windows API function `Rectangle()`. Thus, the OEM character codes are not needed in Windows, and you can effectively ignore them when writing your Windows programs.

Note that although you can ignore these key codes, Windows cannot. For instance, you probably already know that many of your DOS programs can be run in a window under Windows. When this is the case, Windows must "interpret" the DOS program's use of the OEM character set and map it accordingly on-screen.

Cross Reference:

XXI.15: What are OEM key codes?

XXI.17: What are virtual key codes?

XXI.18: What is a dead key?

XXI.17: What are virtual key codes?

Answer:

When your program receives a `WM_KEYUP`, `WM_KEYDOWN`, `WM_SYSKEYUP`, or `WM_SYSKEYDOWN` message, the `wParam` parameter will contain the keystroke's virtual key code. This virtual key code can be used to reference what key on the keyboard was pressed. The key code does not map to any physical character set (such as the OEM key codes—see FAQ XXI.16), but rather it originates from a "virtual" table (set forth in windows.h) of key codes. Table XXI.17 lists some available virtual key codes.

Table XXI.17. Some of the virtual key codes available in Windows programs.

Hex	*Symbolic Name*	*Key*
01	`VK_LBUTTON`	N/A
02	`VK_RBUTTON`	N/A
03	`VK_CANCEL`	Ctrl-Break
04	`VK_MBUTTON`	N/A
08	`VK_BACK`	Backspace
09	`VK_TAB`	Tab
0C	`VK_CLEAR`	Numeric keypad 5 (Num Lock off)
0D	`VK_RETURN`	Enter
10	`VK_SHIFT`	Shift
11	`VK_CONTROL`	Ctrl
12	`VK_MENU`	Alt
13	`VK_PAUSE`	Pause
14	`VK_CAPITAL`	Caps Lock
1B	`VK_ESCAPE`	Esc
20	`VK_SPACE`	Spacebar
21	`VK_PRIOR`	Page Up
22	`VK_NEXT`	Page Down

Hex	*Symbolic Name*	*Key*
23	VK_END	End
24	VK_HOME	Home
25	VK_LEFT	Left arrow
26	VK_UP	Up arrow
27	VK_RIGHT	Right arrow
28	VK_DOWN	Down arrow
29	VK_SELECT	N/A
2A	VK_PRINT	N/A
2B	VK_EXECUTE	N/A
2C	VK_SNAPSHOT	Print Screen
2D	VK_INSERT	Insert
2E	VK_DELETE	Delete
2F	VK_HELP	N/A
30–39		0 through 9 on main keyboard
41–5A		A through Z
60	VK_NUMPAD0	Numeric keypad 0
61	VK_NUMPAD1	Numeric keypad 1
62	VK_NUMPAD2	Numeric keypad 2
63	VK_NUMPAD3	Numeric keypad 3
64	VK_NUMPAD4	Numeric keypad 4
65	VK_NUMPAD5	Numeric keypad 5
66	VK_NUMPAD6	Numeric keypad 6
67	VK_NUMPAD7	Numeric keypad 7
68	VK_NUMPAD8	Numeric keypad 8
69	VK_NUMPAD9	Numeric keypad 9
6A	VK_MULTIPLY	Numeric keypad *
6B	VK_ADD	Numeric keypad +
6C	VK_SEPARATOR	N/A
6D	VK_SUBTRACT	Numeric keypad –
6E	VK_DECIMAL	Numeric keypad
6F	VK_DIVIDE	Numeric keypad /
70	VK_F1	F1
71	VK_F2	F2
72	VK_F3	F3
73	VK_F4	F4

continues

Table XXI.17. continued

Hex	*Symbolic Name*	*Key*
74	VK_F5	F5
75	VK_F6	F6
76	VK_F7	F7
77	VK_F8	F8
78	VK_F9	F9
79	VK_F10	F10
7A	VK_F11	F11
7B	VK_F12	F12
7C	VK_F13	N/A
7D	VK_F14	N/A
7E	VK_F15	N/A
7F	VK_F16	N/A
90	VK_NUMLOCK	Num Lock
91	VK_SCROLL	Scroll Lock

Many more virtual keys are available, but most of them depend on which international settings you have set up for your Windows configuration.

Note that besides being able to obtain the keystroke from Windows, you can also obtain the state of the Shift, Ctrl (Control), and Alt keys. You can do so by using the function `GetKeyState()`. For instance, the function call

```
GetKeyState(VK_SHIFT);
```

returns a negative value if the Shift key is down (pressed). If the Shift key is not pressed, the return value is positive.

Cross Reference:

XXI.15: What are OEM key codes?

XXI.16: Should a Windows program care about the OEM key codes?

XXI.18: What is a dead key?

XXI.18: What is a dead key?

Answer:

A dead key is a keystroke that is not recognizable by Windows. On some international keyboards, it is impossible to translate certain characters into keystrokes. In this case, Windows sends either a `WM_DEADCHAR` or a `WM_SYSDEADCHAR` message to your program, indicating that Windows cannot interpret the character code of the incoming keystroke.

You can, however, manually reference the actual ASCII character code of the incoming character. When your program receives one of these two messages, you can inspect the value of the `wParam` parameter and determine which key was pressed. You therefore can manually customize your programs for internationalization by determining ahead of time which foreign characters your program needs to handle.

Cross Reference:

XXI.15: What are OEM key codes?

XXI.16: Should a Windows program care about the OEM key codes?

XXI.17: What are virtual key codes?

XXI.19: What is the difference between the caret and the cursor?

Answer:

In Windows, the cursor represents the mouse position on the screen. The caret represents the current editing position. If you look at the Windows program Notepad, for example, you'll notice that as you move the mouse around, you see the familiar arrow move. This arrow is the cursor; it represents the current position of the mouse.

If you type some text into the Notepad program, you'll notice that the next available edit position in the Notepad window has a blinking vertical bar in it. This is the caret; it represents the current editing position. You can control the caret's blink rate by invoking the Windows control panel.

In Windows programs, five functions are available to control the caret. These functions are listed in Table XXI.19.

Table XXI.19. Functions to control the caret.

Function Name	*Purpose*
`CreateCaret`	Creates a caret
`SetCaretPos`	Sets the position of the caret
`ShowCaret`	Shows the caret
`HideCaret`	Hides the caret
`DestroyCaret`	Destroys the caret

If you're a die-hard DOS programmer moving into Windows programming, you might think it odd that the "cursor" position actually represents the mouse position and not the editing position. This is just one little caveat you must get used to when joining the ranks of Windows programmers who now have to refer to the "cursor" position as the "caret" position.

Cross Reference:

None.

XXI.20: Can a mouse click be captured in an area outside your program's client area?

Answer:

In Windows, the client area of your program includes all the space within the border of your window, with the exception of the following areas:

- The title bar
- The scrollbars
- The pull-down menu

Can a mouse click be captured within any of these three regions? Yes. When the mouse is clicked in these regions, Windows sends a "nonclient area" message to your program. This way, you can trap for these events when they occur.

Trapping for these events is unusual, however. This is because Windows has prebuilt functionality to handle mouse clicks in these regions. For instance, if you double-click on a window's title bar, the window resizes itself (maximized or restored). If you click on a scrollbar, the window scrolls. If you click on a pull-down menu, the menu is shown on-screen. None of these events requires any code to be written—they are automatically handled by Windows.

Most of the time, you will want to pass these messages to what is called the `DefWindowProc()` function. The `DefWindowProc()` calls the default window procedure (that is, it implements the window's built-in functionality). You very rarely would need to trap for a nonclient mouse hit. Nonetheless, Table XXI.20 presents some of the messages you can trap for.

Table XXI.20. Nonclient area mouse events.

Nonclient Message	*Meaning*
`WM_NCLBUTTONDOWN`	Nonclient left mouse button down
`WM_NCMBUTTONDOWN`	Nonclient middle mouse button down
`WM_NCRBUTTONDOWN`	Nonclient right mouse button down
`WM_NCLBUTTONUP`	Nonclient left mouse button up
`WM_NCMBUTTONUP`	Nonclient middle mouse button up
`WM_NCRBUTTONUP`	Nonclient right mouse button up
`WM_NCLBUTTONDBLCLK`	Nonclient left mouse button double-click
`WM_NCMBUTTONDBLCLK`	Nonclient middle mouse button double-click
`WM_NCRBUTTONDBLCLK`	Nonclient right mouse button double-click

Cross Reference:

XXI.14: How do you determine a Windows program's client area size?

XXI.21: How do you create an animated bitmap?

Answer:

Sometimes you will run across a Windows program that entertains you with an animated bitmap. How is this task accomplished? One method is to set up a timer event that switches the bitmap every second or two, thus making the bitmap "appear" to be animated. In fact, it is not animated, but rather several versions of the same bitmap are switched fast enough to make it appear as though the bitmap is moving.

The first step is to insert the following code into your `WinMain()` function:

```
SetTimer(hwnd, 1, 1000, NULL);
```

This code sets up a timer event that will be invoked every 1000 clock ticks (1 second). In your event (message) loop, you can then trap the timer event, as shown here:

```
switch(message)
{

     case WM_TIMER:

          /* trapped timer event; perform something here */

}
```

Now, when the `WM_CREATE` message comes through, you can load the original bitmap:

```
case WM_CREATE:

     hBitmap = LoadBitmap(hInstance, BMP_ButterflyWingsDown);
```

In this case, `BMP_ButterflyWingsDown` is a bitmap resource bound to the executable through the use of a resource editor. Every time a `WM_TIMER` event is triggered, the following code is performed:

```
case WM_TIMER:

     if (bWingsUp)
          hBitmap = LoadBitmap(hInstance, BMP_ButterflyWingsDown);
     else
          hBitmap = LoadBitmap(hInstance, BMP_ButterflyWingsUp);
```

This way, by using the boolean flag `bWingsUp`, you can determine whether the butterfly bitmap's wings are up or down. If they are up, you display the bitmap with the wings down. If they are down, you display the bitmap with the wings up. This technique gives the illusion that the butterfly is flying.

Cross Reference:

None.

XXI.22: How do you get the date and time in a Windows program?

Answer:

To get the date and time in a Windows program, you should call the standard C library functions `time()` and `localtime()` or some derivative (`asctime()`, `ctime()`, `_ftime()`, `gmttime()`). These functions are compatible with both DOS and Windows. You should never attempt to call a DOS-only or a ROM BIOS function directly. You should always use either Windows API function calls or standard C library routines. Here is an example of code that can be used to print the current date and time in a Windows program:

```
char*          szAmPm = "PM";
char           szCurrTime[128];
char           szCurrDate[128];
struct tm*     tmToday;
time_t         lTime;

time(&lTime);

tmToday = localtime(lTime);

wsprintf(szCurrDate, "Current Date: %02d/%02d/%02d",
                      tmToday->tm_month, tmToday->tm_mday,
                      tmToday->tm_year);

if (tmToday->tm_hour < 12 )
     strcpy(szAmPm, "AM" );

if (tmToday->tm_hour > 12 )
     tmToday->tm_hour -= 12;

wsprintf(szCurrTime, "Current Time: %02d:%02d:%02d %s",
                      tmToday->tm_hour, tmToday->tm_min,
                      tmToday->tm_sec, szAmPm);

TextOut(50, 50, szCurrDate, strlen(szCurrDate));

TextOut(200, 50, szCurrTime, strlen(szCurrTime));

}
```

The `time()` and `localtime()` functions are used to get the current local time (according to the Windows timer, which gets its time from MS-DOS). The `time()` function returns a `time_t` variable, and the `localtime()` function returns a tm structure. The tm structure can easily be used to put the current date and time into a readable format. After this task is completed, the `wsprintf()` function is used to format the date and time into two strings, `szCurrDate` and `szCurrTime`, which are then printed in the current window via the `TextOut()` Windows API function call.

Cross Reference:

None.

XXI.23: How do you update the title bar in a Windows program?

Answer:

The title bar (or caption bar, as it is often called) can be updated in a Windows program by using the Windows API function `SetWindowText()`. The `SetWindowText()` function takes two parameters. The first parameter is the handle to the window, and the second parameter is the new title you want to display on the window.

One reason you might want to take this action is to provide your users with the current date and time on the title bar. This task can be accomplished with the following code:

```
char*           szAmPm = "PM";
char            szNewCaption[200];
struct tm*      tmToday;
time_t          lTime;

time(&lTime);

tmToday = localtime(lTime);

wsprintf(szNewCaption,
         "My Application - %02d/%02d/%02d   %02d:%02d:%02d %s",
         tmToday->tm_month, tmToday->tm_mday,  tmToday->tm_year,
         tmToday->tm_hour, tmToday->tm_min,
         tmToday->tm_sec, szAmPm);

SetWindowText(hwnd, szNewCaption);
```

Of course, you probably will want to set up this code in some sort of timer event loop so that the title is updated every second (or minute).

Cross Reference:

None.

XXI.24: How do you access the system colors in a Windows program?

Answer:

You can obtain the system colors by calling the Windows API function `GetSysColor()`. The `GetSysColor()` function takes one parameter, which signifies which color element you want to obtain. The color elements are represented by color constants defined in the windows.h header file. The Windows system color constants are listed in the following FAQ (XXI.25).

For instance, to obtain the color for the window's active border, you might make the following function call:

```
rgbColor = GetSysColor(COLOR_ACTIVEBORDER);
```

The `GetSysColor()` function returns an RGB value. The RGB value represents the intensity of the colors red, green, and blue that are present in the returned color. An RGB value of 0 signifies black, and an RGB value of 255 signifies white. You can extract the individual red, green, and blue values from the RGB value by calling the `GetRValue()`, `GetGValue()`, and `GetBValue()` Windows API functions.

The Windows API function `SetSysColors()` can be used to set system colors. Here is an example of some code that sets the color of the active border to red:

```
int         aiColorElements[1];
DWORD       argbColor[1];

aiColorElements[0] = COLOR_ACTIVEBORDER;

argbColor[0] = RGB(0xFF, 0x00, 0x00);

SetSysColors(1, aiColorElements, argbColor);
```

The `SetSysColors()` function takes three arguments. The first argument is the number of elements to set color for, the second is an array of integers holding the system color constants to set color for, and the third is an array of RGB values that correspond to the colors you want to invoke for the elements represented by the second argument.

Cross Reference:

XXI.25: What are the system color constants?

XXI.25: What are the system color constants?

Answer:

The system color constants are used by Windows to control the colors of various objects included in the Windows environment. Table XXI.25 lists the system color constants (as defined in windows.h).

Table XXI.25. The system color constants.

Color Constant	*Target Object*
`COLOR_SCROLLBAR`	Scrollbar
`COLOR_BACKGROUND`	Windows desktop
`COLOR_ACTIVECAPTION`	Active title
`COLOR_INACTIVECAPTION`	Inactive title
`COLOR_MENU`	Menu bar
`COLOR_WINDOW`	Window
`COLOR_WINDOWFRAME`	Window frame
`COLOR_MENUTEXT`	Menu text
`COLOR_WINDOWTEXT`	Window text

Color Constant	*Target Object*
COLOR_CAPTIONTEXT	Title text
COLOR_ACTIVEBORDER	Active border
COLOR_INACTIVEBORDER	Inactive border
COLOR_APPWORKSPACE	Application workspace
COLOR_HIGHLIGHT	Highlight
COLOR_HIGHLIGHTTEXT	Highlight text
COLOR_BTNFACE	Button face
COLOR_BTNSHADOW	Button shadow
COLOR_GRAYTEXT	Grayed-out text
COLOR_BTNTEXT	Button text

You can change the system colors from within your Windows programs by calling the `GetSysColor()` and `SetSysColor()` functions. You can also set these colors by altering the `[colors]` section of your WIN.INI (Windows initialization) file, or you can interactively set them by using the Windows control panel.

Cross Reference:

XXI.24: How do you access the system colors in a Windows program?

XXI.26: How do you create your own buttons or controls?

Answer:

Controls such as buttons are typically created with a resource editor. With a resource editor, you can interactively design your windows and place pushbuttons, check boxes, radio buttons, and other controls in your window. You can then access them from within your Windows program by referring to each resource's unique resource id (which you define).

This is not the only way, however, to create controls such as buttons. Buttons and other controls are called "child window controls." Each child window control has the capability to trap incoming messages (such as the `WM_COMMAND` message) and pass them on to the parent window control. A child window control such as a pushbutton can be created by using the Windows API function `CreateWindow()`. It might seem odd to call the function `CreateWindow()` to create a pushbutton, but a control is, in effect, its own "virtual" window, and thus it needs to have its own handle. Here is some sample code that shows how this task is performed:

```
...

switch (message)
{

   ...

   case WM_CREATE:
```

```
        hwndCloseButton =
          CreateWindow("button",   /* Windows registered class name */
                       "Close",    /* Window text (title) */
                       WS_CHILD | WS_VISIBLE | PUSHBUTTON,    /* Style */
                       50,         /* Horizontal position */
                       50,         /* Vertical position */
                       100,        /* Width */
                       100,        /* Height */
                       hwndParent, /* Handle of parent window */
                       0,          /* Child-window identifier */
                       ((LPCREATESTRUCT) lParam)->hInstance,
                       NULL);  /* Window creation options */

    ...

}
```

Cross Reference:

None.

XXI.27: What is a static child window?

Answer:

A static child window is a window control that does not accept mouse or keyboard input. Typical examples of static child windows are rectangles, frames, a window's background or border, and static text (labels) that appear on-screen. Typically, it makes no sense to process mouse or keyboard events when dealing with static controls.

A static control can be created by specifying the "static" class in the Windows API `CreateWindow()` function call. Here is an example of a field label that is created by invoking the `CreateWindow()` function:

```
hwndNameLabel = CreateWindow ("static", "Customer Name:",
                              WS_CHILD | WS_VISIBLE | SS_LEFT,
                              0, 0, 0, 0,
                              hwnd,
                              50,
                              hInstance, NULL) ;
```

This example creates a field label in the window with the caption "Customer Name:." This field label would probably coincide with a window of the edit class that would accept the user's input of the customer's name.

Cross Reference:

XXI.28: What is window subclassing?

XXI.28: What is window subclassing?

Answer:

Windows subclassing refers to a technique whereby you can "tap" into a built-in Windows function and add your own functionality without disturbing or abandoning the original Windows functionality. For example, the Windows procedure for a check box control is coded deep within the system internals of Windows, and the source code is not readily available. However, through the use of two Windows API functions, you can tap into this code and build your own functionality into it.

The two Windows API functions that accomplish this task are called `GetWindowLong()` and `SetWindowLong()`. The `GetWindowLong()` function returns the address of the Windows procedure you want to subclass. The `SetWindowLong()` function can be used to override the default Windows procedure and point to your own custom-made version. Note that you do not have to replicate all functionality by doing this—when you need to reference the original procedure's functionality, you can pass the messages through to it.

You can save the old procedure's address by including the following code in your program:

```
lpfnOldCheckBoxProc = (FARPROC) GetWindowLong(hwndCheckBox, GWL_WNDPROC);
```

Your new custom check box procedure can replace the old procedure by including the following code in your program:

```
SetWindowLong(hwndCheckBox, GWL_WNDPROC, (LONG) lpfnCustomCheckBoxProc);
```

In this example, the `GetWindowLong()` function is used to save the old procedure's address. The `GWL_WNDPROC` identifier tells the `GetWindowLong()` function to return a pointer to the check box's procedure. After this is saved, a new procedure (named `lpfnCustomCheckBoxProc`) is invoked by a call to the `SetWindowLong()` function. Now, whenever Windows would normally call `hwndCheckBox`'s default procedure, your custom check box procedure will be called instead.

In your custom check box procedure, you can always pass messages through to the original procedure. This is done by using the Windows API function `CallWindowProc()` as shown here:

```
CallWindowProc(lpfnOldCheckBoxProc, hwnd, message, wParam, lParam);
```

This way, you do not have to replicate all functionality that was in the original procedure. Instead, you can trap for only the messages you want to customize. For instance, if you have ever seen a variation of a check box (such as Borland's custom check box), you know that it does not look like the default check box you would normally see in Windows programs. This variation is accomplished through Windows subclassing. The `WM_PAINT` message is simply subclassed out, and a new customized version of the check box rather than the original version is painted.

Cross Reference:

XXI.27: What is a static child window?

XXI.29: What is the edit class?

Answer:

Windows contains many classes of windows that can be created by using the `CreateWindow()` function. One of these classes is the edit class, which creates a rectangular region in the window that is editable. When this editable region receives the focus, you can edit the text in any way you want—select it; cut, copy, or paste it to the clipboard; or delete it.

A window created with the edit class is called an edit control. Edit controls can be single-line or multiline. Here is an example of a portion of code that creates a multiline scrollable edit control in a window:

```
...

switch (message)
{

   ...

    case WM_CREATE:

       hwndEdit = CreateWindow (“edit”,
                                NULL,
                                WS_CHILD | WS_VISIBLE |
                                WS_HSCROLL | WS_VSCROLL |
                                WS_BORDER | ES_LEFT | ES_MULTILINE |
                                ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                                0, 0, 0, 0,
                                hwnd, 1,
                                ((LPCREATESTRUCT) lParam) -> hInstance,
                                NULL) ;

   ...

}
```

The edit class is very powerful. It has a lot of built-in functionality that does not need to be programmed into it. For instance, all the clipboard commands (cut, copy, paste, delete) are automatically active when focus is switched to an edit control. You do not need to write any code for this functionality to be included.

As you can see from the preceding example, an edit control has many options that can make it totally customizable to your needs. An edit control also carries with it several associated events (messages) that you can trap for. Table XXI.29 lists these events.

Table XXI.29. Events associated with an edit control.

Message	*Meaning*
`EN_SETFOCUS`	Received input focus
`EN_KILLFOCUS`	Lost input focus
`EN_CHANGE`	Contents are about to change
`EN_UPDATE`	Contents have changed
`EN_HSCROLL`	Horizontal scrollbar clicked

Message	*Meaning*
EN_VSCROLL	Vertical scrollbar clicked
EN_ERRSPACE	No space left in buffer
EN_MAXTEXT	No space left while in insert mode

Cross Reference:

XXI.30: What is the listbox class?

XXI.30: What is the listbox class?

Answer:

One of the predefined classes available when you are calling the `CreateWindow()` function is the listbox class. This class provides a vertically scrollable list of items enclosed in a rectangular region. This list of items can be modified—you can add items to and delete items from the list at runtime. A listbox control can be a single-selection listbox (only one item at a time can be selected) or a multiselection listbox (more than one item at a time can be selected).

As with the edit class, the listbox class comes with a tremendous amount of predefined functionality. You do not need to program in many of the listbox class's functions. For instance, in a single-selection listbox, you can move the arrow keys up and down, and the selection changes with the movement. If the listbox is scrollable, the list automatically scrolls. The Page Up and Page Down keys scroll the listbox region one page up or down. You can even perform an "auto-search" within the listbox: when you press a letter on the keyboard, the listbox "jumps" to the first item that begins with that letter. When you press the spacebar, the current item is selected. The multiselectable listbox control has all of this functionality and more. Plus, each of these listbox styles is automatically mouse-enabled.

You can create a new listbox control in a window by using the `CreateWindow()` function like this:

```
...

switch (message)
{

   ...

   case WM_CREATE :

        hwndList = CreateWindow ("listbox", NULL,
                                 WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                                 100,
                                 200 + GetSystemMetrics (SM_CXVSCROLL),
                                 200,
                                 hwnd, 1,
                                 GetWindowWord (hwnd, GWW_HINSTANCE),
                                 NULL) ;

   ...
```

```
}

...
```

Like the edit class, the listbox class comes with many associated attributes and events (messages). Table XXI.30 presents some messages available for the listbox class.

Table XXI.30. Some of the available messages for the listbox class.

Message	*Meaning*
LBN_SETFOCUS	The listbox received input focus
LBN_KILLFOCUS	The listbox lost input focus
LBN_SELCHANGE	The current selection has changed
LBN_DBLCLK	The user double-clicked the mouse on a selection
LBN_SELCANCEL	The user deselected an item
LBN_ERRSPACE	The listbox control has run out of space

Cross Reference:

XXI.29: What is the edit class?

XXI.31: How is memory organized in Windows?

Answer:

Windows organizes its memory into two "heaps": the local heap and the global heap. The local heap is much like the local heap of a DOS program. It contains room for static data, some stack space, and any free memory up to 64K. You can access memory in this area in the same manner as you would access a DOS program's memory (by using the `malloc()` or `calloc()` functions).

The global heap, however, is totally different from anything available under DOS. The global heap is controlled by Windows, and you cannot use standard C function calls to allocate memory from the global heap. Instead, you must use Windows API function calls to allocate global memory. Global memory segments can be classified as either fixed or movable. A fixed segment in the global heap cannot be moved in memory. Its address remains the same throughout the life of the program. On the other hand, a movable segment in the global heap can be "moved" to another location in memory to make more contiguous room for global allocation. Windows is smart enough to move memory when it needs to allocate more for Windows programs. This action is roughly equivalent to an "automatic" `realloc()` function call.

Generally, you should make every effort to ensure that the code and data segments of your Windows programs are marked as movable. You should do so because movable segments can be handled at runtime by Windows. Fixed segments, on the other hand, always reside in memory, and Windows can never reclaim this space. Heavy use of fixed segments might make Windows perform acrobatic "disk thrashing" with your hard drive, because Windows attempts to shuffle things in memory to make more room.

Using movable segments allows Windows to have more control at runtime, and your programs will generally be better liked by your users. Movable segments can also be marked as discardable. This means that when Windows needs additional memory space, it can free up the area occupied by the segment. Windows uses a "least recently used" (LRU) algorithm to determine which segments to discard when attempting to free up memory. Code, data, and resources can be discarded and later read back in from your program's .EXE file.

Cross Reference:

XXI.32: How is memory allocated in a Windows program?

XXI.32: How is memory allocated in a Windows program?

Answer:

FAQ XXI.31 explained how memory was organized under Windows. For allocating memory from the local heap, typical C functions such as `malloc()` or `calloc()` can be used. However, to allocate memory from the global heap, you must use one of the Windows API functions, such as `GlobalAlloc()`. The `GlobalAlloc()` function allocates memory from the global heap and marks it as fixed, movable, or discardable (see FAQ XXI.31 for an explanation of these three terms).

Here is an example of a Windows program code snippet that allocates 32K of global memory:

```
GLOBALHANDLE      hGlobalBlock;
LPSTR             lpGlobalBlock;

hGlobalBlock = GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, 0x8000L);

... /* various program statements */

lpGlobalBlock = GlobalLock(hGlobalBlock);

... /* various program statements */

GlobalUnlock(hGlobalBlock);

... /* various program statements */

GlobalFree(hGlobalBlock);
```

Take a look at how the preceding portion of code works. First, two variables are declared. The first variable is `hGlobalBlock`, which is of type `GLOBALHANDLE`. This is simply defined as a 16-bit integer to hold the handle of the block returned by Windows. Notice that, unlike DOS, Windows returns a handle to an allocated portion of memory. It does so because portions of memory can be moved or discarded, thus invalidating pointer usage. If you use pointers in your program, you can't be sure that Windows has not moved the portion of memory you were working with—that is, unless you call the `GlobalLock()` function (which is explained in the next paragraph).

The first function call, `GlobalAlloc()`, is used to allocate a 32K portion of global heap memory that is flagged as movable and is automatically zero-initialized. After calling this function, `hGlobalBlock` will contain a handle to this portion of memory. Using this handle, you can reference the memory as much as you want in your program. It makes no difference to you if the memory is moved by Windows—the handle will remain

the same. However, if you need to work with a pointer to this memory rather than a handle (for accessing memory directly), you need to call the `GlobalLock()` function. This function is used to "lock" (that is, mark as nonmovable) the portion of memory that you have allocated. When memory is locked like this, Windows will not move the memory, and thus the pointer returned from the `GlobalLock()` function will remain valid until you explicitly unlock the memory by using the `GlobalUnlock()` function.

When you are finished working with the allocated global heap memory, you can free it up and return its space to Windows by calling the `GlobalFree()` Windows API function.

Cross Reference:

XXI.31: How is memory organized in Windows?

XXI.33: What is the difference between modal and modeless dialog boxes?

Answer:

Windows dialog boxes can be either "modal" or "modeless." When your program displays a modal dialog box, the user cannot switch between the dialog box and another window in your program. The user must explicitly end the dialog box, usually by clicking a pushbutton marked OK or Cancel. The user can, however, generally switch to another program while the dialog box is still displayed. Some dialog boxes (called "system modal") do not allow even this movement. System modal dialog boxes must be ended before the user does anything else in Windows. A common use of modal dialog boxes is to create an "About" box that displays information regarding your program.

Modeless dialog boxes enable the user to switch between the dialog box and the window that created it as well as between the dialog box and other programs. Modeless dialog boxes are preferred when the user would find it convenient to keep the dialog box displayed awhile. Many word processing programs such as Microsoft Word use modeless dialog boxes for utilities such as the spell checker—the dialog box remains displayed on-screen, enabling you to make changes to your text and return immediately to your spell checker.

Modeless dialog boxes are created by using the `CreateDialog()` function:

```
hdlgModeless = CreateDialog(hInstance, lpszTemplate,
                            hwndParent, lpfnDialogProc) ;
```

The `CreateDialog()` function returns immediately with the window handle of the dialog box. Modal dialog boxes are created by using the `DialogBox()` function:

```
DialogBox(hInstance, "About My Application", hwnd, lpfnAboutProc);
```

The `DialogBox()` function returns when the user closes the dialog box.

Cross Reference:

None.