Important Instructions to examiners:

1) The answers should be examined by key words and not as word-to-word as given in the model answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given more Importance (Not applicable for subject English and Communication Skills).
4) While assessing figures, examiner may give credit for principal components indicated in the figure. The figures drawn by candidate and model answer may vary. The examiner may give credit for any equivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constant values may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgement on part of examiner of relevant answer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalent concept.

1) Attempt any five : [Marks 20]

a) State the aspects of Windows.

(4 marks for any of the four points from below.)

1) Both Windows 98 and Windows NT are 32-bit preemptive multitasking and multithreading graphical operating systems. Windows possesses a graphical user interface (GUI), sometimes also called a "visual interface" or "graphical windowing environment."
2) All GUIs make use of graphics on a bitmapped video display. Graphics provides better utilization of screen real estate, a visually rich environment for conveying information, and the possibility of a WYSIWYG (what you see is what you get) video display of graphics and formatted text prepared for a printed document.
3) In earlier days, the video display was used solely to echo text that the user typed using the keyboard. In a graphical user interface, the video display itself becomes a source of user input. The video display shows various graphical objects in the form of icons and input devices such as buttons and scroll bars. Using the keyboard (or, more directly, a pointing device such as a mouse), the user can directly manipulate these objects on the screen. Graphics objects can be dragged, buttons can be pushed, and scroll bars can be scrolled.
4) Rather than the one-way cycle of information from the keyboard to the program to the video display, the user directly interacts with the objects on the display.
5) Users no longer expect to spend long periods of time learning how to use the computer or mastering a new program. Windows helps because all applications have the same fundamental look and feel. The program occupies a window—usually a rectangular area on the screen. Each window is identified by a caption bar. Most program functions are initiated through the program's menus. A

6) user can view the display of information too large to fit on a single screen by using scroll bars. Some menu items invoke dialog boxes, into which the user enters additional information. One dialog box in particular, that used to open a file, can be found in almost every large Windows program. This dialog box looks the same (or nearly the same) in all of these Windows programs, and it is almost always invoked from the same menu option.
7) Once you know how to use one Windows program, you're in a good position to easily learn another. The menus and dialog boxes allow a user to experiment with a new program and explore its features. Most Windows programs have both a keyboard interface and a mouse interface. Although most functions of Windows programs can be controlled through the keyboard, using the mouse is often easier for many chores.
8) From the programmer's perspective, the consistent user interface results from using the routines built into Windows for constructing menus and dialog boxes. All menus have the same keyboard and mouse interface because Windows—rather than the application program—handles this job.
9) To facilitate the use of multiple programs, and the exchange of information among them, Windows supports multitasking. Several Windows programs can be displayed and running at the same time. Each program occupies a window on the screen. The user can move the windows around on the screen, change their sizes, switch between different programs, and transfer data from one program to another. Because these windows look something like papers on a desktop (in the days before the desk became dominated by the computer itself, of course), Windows is sometimes said to use a "desktop metaphor" for the display of multiple programs.
10) Earlier versions of Windows used a system of multitasking called "nonpreemptive." This meant that Windows did not use the system timer to slice processing time between the various programs running under the system. The programs themselves had to voluntarily give up control so that other programs could run. Under Windows NT and Windows 98, multitasking is preemptive and programs themselves can split into multiple threads of execution that seem to run concurrently.
11) An operating system cannot implement multitasking without doing something about memory management. As new programs are started up and old ones terminate, memory can become fragmented. The system must be able to consolidate free memory space. This requires the system to move blocks of code and data in memory.

12) Programs running in Windows can share routines that are located in other files called "dynamic-link libraries." Windows includes a mechanism to link the program with the routines in the dynamic-link libraries at run time. Windows itself is basically a set of dynamic-link libraries.

13) Windows is a graphical interface, and Windows programs can make full use of graphics and formatted text on both the video display and the printer. A graphical interface not only is more attractive in appearance but also can impart a high level of information to the user.

14) Programs written for Windows do not directly access the hardware of graphics display devices such as the screen and printer. Instead, Windows includes a graphics programming language (called the Graphics Device Interface, or GDI) that allows the easy display of graphics and formatted text. Windows virtualizes display hardware. A program written for Windows will run with any video board or any printer for which a Windows device driver is available. The program does not need to determine what type of device is attached to the system.

15) Putting a device-independent graphics interface on the IBM PC was not an easy job for the developers of Windows. The PC design was based on the principle of open architecture. Third-party hardware manufacturers were encouraged to develop peripherals for the PC and have done so in great number. Although several standards have emerged, conventional MS-DOS programs for the PC had to individually support many different hardware configurations. It was fairly common for an MS-DOS word-processing program to be sold with one or two disks of small files, each one supporting a particular printer. Windows programs do not require these drivers because the support is part of Windows.

b) Explain MM-TEXT mapping mode.

(1 mark for details of MM_TEXT, 3 marks for Three points on description/explanation)

		Increasing Value	
Mapping Mode	***Logical Unit***	**x-*axis***	**y-*axis***
MM_TEXT	Pixel	Right	Down

The default mapping mode is MM_TEXT. In this mapping mode, logical units are the same as physical units, which allows us (or, depending on your perspective, forces us) to work directly in units of pixels. In a *TextOut* call that looks like this:

TextOut (hdc, 8, 16, TEXT ("Hello"), 5) ;

the text begins 8 pixels from the left of the client area and 16 pixels from the top.

If you feel comfortable working in units of pixels, you don't need to use any mapping modes except the default MM_TEXT mode.

c) Describe GDI primitives.

(4 marks for four GDI primitives)

The types of graphics you display on the screen or the printer can themselves be divided into several categories, which are called "primitives." These are:

- **Lines and curves** Lines are the foundation of any vector graphics drawing system. GDI supports straight lines, rectangles, ellipses (including that subset of ellipses known as circles), "arcs" that are partial curves on the circumference of an ellipse, and Bezier splines, all of which I'll discuss in this chapter. If you need to draw a different type of curve, you can draw it as a polyline, which is a series of very short lines that define a curve. GDI draws lines using the current pen selected in the device context.
- **Filled areas** Whenever a series of lines or curves encloses an area, you can cause that area to be filled with the current GDI brush object. This brush can be a solid color, a pattern (which can be a series of horizontal, vertical, or diagonal hatch marks), or a bitmapped image that is repeated vertically or horizontally within the area.
- **Bitmaps** A bitmap is a rectangular array of bits that correspond to the pixels of a display device. The bitmap is the fundamental tool of raster graphics. Bitmaps are generally used for displaying complex (often real-world) images on the video display or printer. Bitmaps are also used for displaying small images that must be drawn very quickly, such as icons, mouse cursors, and buttons that appear in application toolbars. GDI supports two types of bitmaps—the old (although still quite useful) "device-dependent" bitmap, which is a GDI object, and the newer (as of Windows 3.0) "device-independent" bitmap (or DIB), which can be stored in disk files.
- **Text** Text is not quite as mathematical as other aspects of computer graphics; instead it is bound to hundreds of years of traditional typography, which many typographers and other observers appreciate as an art. For this reason, text is often the most complex part of any computer graphics system, but it is also (assuming literacy remains the norm) the most important. Data structures used for defining GDI font objects and for obtaining font information are among the largest in Windows. Beginning with Windows 3.1, GDI began supporting TrueType fonts, which are based on filled outlines that can be manipulated with other GDI functions. Windows 98 continues to support the older bitmap-based fonts for compatibility and small memory requirements.

d) What is the purpose of virtual key code?

(4 marks for Four points on purpose from any of the following)

The virtual key code is stored in the wParam parameter of the WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, and WM_SYSKEYUP messages. This code identifies the key being pressed or released.

The virtual key codes you use most often have names beginning with VK_ defined in the WINUSER.H header file. The tables below show these names along with the numeric values (in both decimal and hexadecimal) and the IBM-compatible keyboard key that corresponds to the virtual key. The tables also indicate whether these keys are required for Windows to run properly. The tables show the virtual key codes in numeric order.

Three of the first four virtual key codes refer to mouse buttons:

Decimal	***Hex***	***WINUSER.H Identifier***	***Required?***	***IBM-Compatible Keyboard***
1	01	VK_LBUTTON		Mouse Left Button
2	02	VK_RBUTTON		Mouse Right Button
3	03	VK_CANCEL	X	Ctrl-Break
4	04	VK_MBUTTON		Mouse Middle Button

Several of the following keys—Backspace, Tab, Enter, Escape, and Spacebar—are commonly used by Windows programs. However, Windows programs generally use character messages (rather than keystroke messages) to process these keys.

Decimal	*Hex*	*WINUSER.H Identifier*	*Required?*	*IBM-Compatible Keyboard*
8	08	VK_BACK	X	Backspace
9	09	VK_TAB	X	Tab
12	0C	VK_CLEAR		Numeric keyboard 5 with Num Lock OFF
13	0D	VK_RETURN	X	Enter (either one)
16	10	VK_SHIFT	X	Shift (either one)
17	11	VK_CONTROL	X	Ctrl (either one)
18	12	VK_MENU	X	Alt (either one)
19	13	VK_PAUSE		Pause
20	14	VK_CAPITAL	X	Caps Lock
27	1B	VK_ESCAPE	X	Esc
32	20	VK_SPACE	X	Spacebar

Also, Windows programs usually do not need to monitor the status of the Shift, Ctrl, or Alt keys.

The first eight codes listed in the following table are perhaps the most commonly used virtual key codes along with VK_INSERT and VK_DELETE:

Decimal	*Hex*	*WINUSER.H Identifier*	*Required?*	*IBM-Compatible Keyboard*
33	21	VK_PRIOR	X	Page Up
34	22	VK_NEXT	X	Page Down
35	23	VK_END	X	End
36	24	VK_HOME	X	Home
37	25	VK_LEFT	X	Left Arrow
38	26	VK_UP	X	Up Arrow
39	27	VK_RIGHT	X	Right Arrow
40	28	VK_DOWN	X	Down Arrow
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT		Print Screen
45	2D	VK_INSERT	X	Insert
46	2E	VK_DELETE	X	Delete
47	2F	VK_HELP		

Notice that many of the names (such as VK_PRIOR and VK_NEXT) are unfortunately quite different from the labels on the keys and also not consistent with the identifiers used in scroll bars. The Print Screen key is largely ignored by Windows applications. Windows itself responds to the key by storing a

bitmap copy of the video display into the clipboard. VK_SELECT, VK_PRINT, VK_EXECUTE, and VK_HELP might be found on a hypothetical keyboard that few of us have ever seen.

Windows also includes virtual key codes for the letter keys and number keys on the main keyboard. (The number pad is handled separately.)

Decimal	***Hex***	***WINUSER.H Identifier***	***Required?***	***IBM-Compatible Keyboard***
48_57	30_3	None	X	0 through 9 on main keyboard
65_90	41_5A	None	X	A through Z

Notice that the virtual key codes are the ASCII codes for the numbers and letters. Windows programs almost never use these virtual key codes; instead, the programs rely on character messages for ASCII characters.

The following keys are generated from the Microsoft Natural Keyboard and compatibles:

Decimal	***Hex***	***WINUSER.H Identifier***	***Required?***	***IBM-Compatible Keyboard***
91	5B	VK_LWIN		Left Windows key
92	5C	VK_RWIN		Right Windows key
93	5D	VK_APPS		Applications key

The VK_LWIN and VK_RWIN keys are handled by Windows to open the Start menu or (in older versions) to launch the Task Manager. Together, they can log on or off Windows (in Microsoft Windows NT only), or log on or off a network (in Windows for Workgroups). Applications can process the application key by displaying help information or shortcuts.

The following codes are for the keys on the numeric keypad (if present):

Decimal	***Hex***	***WINUSER.H Identifier***	***Required?***	***IBM-Compatible Keyboard***
96-105	60-69	VK_NUMPAD0 through VK_NUMPAD9		Numeric keypad 0 through 9 with Num Lock ON
106	6A	VK_MULTIPLY		Numeric keypad *
107	6B	VK_ADD		Numeric keypad +
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		Numeric keypad-
110	6E	VK_DECIMAL		Numeric keypad .
111	6F	VK_DIVIDE		Numeric keypad /

Decimal	*Hex*	*WINUSER.H Identifier*	*Required?*	*IBM-Compatible Keyboard*
112-121	70-79	VK_F1 through VK_F10	X	Function keys F1 through F10
122-135	7A-87	VK_F11 through VK_F24		Function keys F11 through F24
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

Finally, although most keyboards have 12 function keys, Windows requires only 10 but has numeric identifiers for 24. Again, programs generally use the function keys as keyboard accelerators so they usually don't process the keystrokes in this table:

e) Describe the process of emulating the mouse with the keyboard.

(4 marks for eight points from the following)

Adding a keyboard interface to a program that uses the mouse cursor for pointing purposes requires that we also must worry about displaying and moving the mouse cursor.

Even if a mouse device is not installed, Windows can still display a mouse cursor. Windows maintains something called a "display count" for this cursor. If a mouse is installed, the display count is initially 0; if not, the display count is initially -1. The mouse cursor is displayed only if the display count is non-negative. You can increment the display count by calling

ShowCursor (TRUE) ;

and decrement it by calling

ShowCursor (FALSE) ;

You do not need to determine if a mouse is installed before using *ShowCursor*. If you want to display the mouse cursor regardless of the presence of the mouse, simply increment the display count by calling ShowCursor. After you increment the display count once, decrementing it will hide the cursor if no mouse is installed but leave it displayed if a mouse is present.

Windows maintains a current mouse cursor position even if a mouse is not installed. If a mouse is not installed and you display the mouse cursor, it might appear in any part of the display and will remain in that position until you explicitly move it. You can obtain the cursor position by calling

GetCursorPos (&pt) ;

where *pt* is a POINT structure. The function fills in the POINT fields with the *x* and *y* coordinates of the mouse. You can set the cursor position by using

SetCursorPos (x, y) ;

In both cases, the *x* and *y* values are screen coordinates, not client-area coordinates. (This should be evident because the functions do not require a hwnd parameter.) As noted earlier, you can convert screen coordinates to client-area coordinates and vice versa by calling ScreenToClient and ClientToScreen.

If you call GetCursorPos while processing a mouse message and you convert to client-area coordinates, these coordinates might be slightly different from those encoded in the lParam parameter of the mouse message. The coordinates returned from GetCursorPos indicate the current position of the mouse. The coordinates in lParam are the coordinates of the mouse at the time the message was generated.

You'll probably want to write keyboard logic that moves the mouse cursor with the keyboard arrow keys and that simulates the mouse button with the Spacebar or Enter key. What you *don't* want to do is move the mouse cursor one pixel per keystroke. That forces a user to hold down an arrow key for too long a time to move it.

f) List the four types of common controls. Explain any one.

(2 marks for naming four types and 2 marks for explanation of any one type)

Any four of these will do

BS_PUSHBUTTON, "PUSHBUTTON",

BS_DEFPUSHBUTTON, "DEFPUSHBUTTON",

BS_CHECKBOX, "CHECKBOX",

BS_AUTOCHECKBOX, "AUTOCHECKBOX",

BS_RADIOBUTTON, "RADIOBUTTON",

BS_3STATE, "3STATE",

BS_AUTO3STATE, "AUTO3STATE",

BS_GROUPBOX, "GROUPBOX",

BS_AUTORADIOBUTTON, "AUTORADIO",

BS_OWNERDRAW, "OWNERDRAW",

Static class,

Scroll bar class,

Edit class,

List box class

Explanation of any one type

g) Describe the following terms:

i) Changing the Button Text

ii) Visible and Enabled Buttons.

(1 mark for function name and 1 mark for syntax)

You can change the text in a button (or in any other window) by calling SetWindowText:

SetWindowText (hwnd, pszString) ;

where hwnd is a handle to the window whose text is being changed and pszString is a pointer to a null-terminated string. For a normal window, this text is the text of the caption bar. For a button control, it's the text displayed with the button.

ii). Visible and enabled buttons.

(2 marks for 4 functions)

To receive mouse and keyboard input, a child window must be both visible (displayed) and enabled. When a child window is visible but not enabled, Windows displays the text in gray rather than black.

If you don't include WS_VISIBLE in the window class when creating the child window, the child window will not be displayed until you make a call to ShowWindow:

ShowWindow (hwndChild, SW_SHOWNORMAL) ;

But if you include WS_VISIBLE in the window class, you don't need to call ShowWindow. However, you can hide the child window by this call to ShowWindow:

ShowWindow (hwndChild, SW_HIDE) ;

You can determine if a child window is visible by a call to

IsWindowVisible (hwndChild) ;

You can also enable and disable a child window. By default, a window is enabled. You can disable it by calling

EnableWindow (hwndChild, FALSE) ;

For button controls, this call has the effect of graying the button text string. The button no longer responds to mouse or keyboard input. This is the best method for indicating that a button option is currently unavailable.

You can reenable a child window by calling

EnableWindow (hwndChild, TRUE) ;

You can determine whether a child window is enabled by calling

IsWindowEnabled (hwndChild) ;

2. Attempt any four: [Marks 16]

a) Explain the role played by Post Quit Message function.

(4 marks for four points about Post Quit Message function)

PostQuitMessage (0);

This function inserts a WM_QUIT message in the program's message queue. I mentioned earlier that GetMessage returns nonzero for any message other than WM_QUIT that it retrieves from the message queue. When GetMessage retrieves a WM_QUIT message, GetMessage returns 0. This causes WinMain to drop out of the message loop. The program then executes the following statement:

return msg.wParam ;

The wParam field of the structure is the value passed to the PostQuitMessage function (generally 0). The return statement exits from WinMain and terminates the program.

b) What is meant by Caret? Explain Caret functions.

(4 marks for eight important points from the following)

When you type text into a program, generally a little underline, vertical bar, or box shows you where the next character you type will appear on the screen. You may know this as a "cursor," but you'll have to get out of that habit when programming for Windows. In Windows, it's called the "caret." The word "cursor" is reserved for the little bitmap image that represents the mouse position.

There are five essential caret functions:

- *CreateCaret* Creates a caret associated with a window.
- *SetCaretPos* Sets the position of the caret within the window.
- *ShowCaret* Shows the caret.
- *HideCaret* Hides the caret.
- *DestroyCaret* Destroys the caret.

There are also functions to get the current caret position (GetCaretPos) and to get and set the caret blink time (GetCaretBlinkTime and SetCaretBlinkTime).

In Windows, the caret is customarily a horizontal line or box that is the size of a character, or a vertical line that is the height of a character.

The main rule for using the caret is simple: a window procedure calls CreateCaret during the WM_SETFOCUS message and DestroyCaret during the WM_KILLFOCUS message.

There are a few other rules: The caret is created hidden. After calling CreateCaret, the window procedure must call ShowCaret for the caret to be visible. In addition, the window procedure must hide the caret by calling HideCaret whenever it draws something on its window during a message other than

WM_PAINT. After it finishes drawing on the window, the program calls ShowCaret to display the caret again. The effect of HideCaret is additive: if you call HideCaret several times without calling ShowCaret, you must call ShowCaret the same number of times before the caret becomes visible again.

c) Difference between Window and Dialog Box.

(4 marks for eight points from the following.)

Dialog boxes are most often used for obtaining additional input from the user beyond what can be easily managed through a menu. The programmer indicates that a menu item invokes a dialog box by adding an ellipsis (...) to the menu item.

A dialog box generally takes the form of a popup window containing various child window controls. The size and placement of these controls are specified in a "dialog box template" in the program's resource script file. Although a programmer can define a dialog box template "manually," these days dialog boxes are usually interactively designed in the Visual C++ Developer Studio. Developer Studio then generates the dialog template.

When a program invokes a dialog box based on a template, Microsoft Windows 98 is responsible for creating the dialog box popup window and the child window controls, and for providing a window procedure to process dialog box messages, including all keyboard and mouse input. The code within Windows that does all this is sometimes referred to as the "dialog box manager."

Many of the messages that are processed by that dialog box window procedure located within Windows are also passed to a function within your own program, called a "dialog box procedure" or "dialog procedure." The dialog procedure is similar to a normal window procedure, but with some important differences. Generally, you will not be doing much within the dialog procedure beyond initializing the child window controls when the dialog box is created, processing messages from the child window controls, and ending the dialog box. Dialog procedures generally do not process WM_PAINT messages, nor do they directly process keyboard and mouse input.

d) Explain functions required to draw pixel and line.

(4 marks for eight functions with syntaxes and argument descriptions)

The SetPixel function sets the pixel at a specified x- and y-coordinate to a particular color:

SetPixel (hdc, x, y, crColor) ;

As in any drawing function, the first argument is a handle to a device context. The second and third arguments indicate the coordinate position. Mostly you'll obtain a device context for the client area of

your window, and x and y will be relative to the upper left corner of that client area. The final argument is of type COLORREF to specify the color. If the color you specify in the function cannot be realized on the video display, the function sets the pixel to the nearest pure nondithered color and returns that value from the function.

Windows can draw straight lines, elliptical lines (curved lines on the circumference of an ellipse), and Bezier splines. Windows 98 supports seven functions that draw lines:

- LineTo Draws a straight line.
- Polyline and PolylineTo Draw a series of connected straight lines.
- PolyPolyline Draws multiple polylines.
- Arc Draws elliptical lines.
- PolyBezier and PolyBezierTo Draw Bezier splines.

In addition, Windows NT supports three more line-drawing functions:

- ArcTo and AngleArc Draw elliptical lines.
- PolyDraw Draws a series of connected straight lines and Bezier splines.

These three functions are not supported under Windows 98.

Later in this chapter I'll also be discussing some functions that draw lines but that also fill the enclosed area within the figure they draw. These functions are

- Rectangle Draws a rectangle.
- Ellipse Draws an ellipse.
- RoundRect Draws a rectangle with rounded corners.
- Pie Draws a part of an ellipse that looks like a pie slice.
- Chord Draws part of an ellipse formed by a chord.

Five attributes of the device context affect the appearance of lines that you draw using these functions: current pen position (for LineTo, PolylineTo, PolyBezierTo, and ArcTo only), pen, background mode, background color, and drawing mode.

To draw a straight line, you must call two functions. The first function specifies the point at which the line begins, and the second function specifies the end point of the line:

```
MoveToEx (hdc, xBeg, yBeg, NULL) ;
LineTo (hdc, xEnd, yEnd) ;
```

MoveToEx doesn't actually draw anything; instead, it sets the attribute of the device context known as the "current position." The LineTo function then draws a straight line from the current position to the point specified in the LineTo function. The current position is simply a starting point for several other GDI functions. In the default device context, the current position is initially set to the point (0, 0). If you

call LineTo without first setting the current position, it draws a line starting at the upper left corner of the client area.

A brief historical note: In the 16-bit versions of Windows, the function to set the current position was MoveTo. This function had just three arguments—the device context handle and x- and y-coordinates. The function returned the previous current position packed as two 16-bit values in a 32-bit unsigned long. However, in the 32-bit versions of Windows, coordinates are 32-bit values. Because the 32-bit versions of C do not define a 64-bit integral data type, this change meant that MoveTo could no longer indicate the previous current position in its return value. Although the return value from MoveTo was almost never used in real-life programming, a new function was required, and this was MoveToEx.

The last argument to MoveToEx is a pointer to a POINT structure. On return from the function, the x and y fields of the POINT structure will indicate the previous current position. If you don't need this information (which is almost always the case), you can simply set the last argument to NULL as in the example shown above.

And now the caveat: Although coordinate values in Windows 98 appear to be 32-bit values, only the lower 16 bits are used. Coordinate values are effectively restricted to -32,768 to 32,767. In Windows NT, the full 32-bit values are used.

If you ever need the current position, you can obtain it by calling

GetCurrentPositionEx (hdc, &pt) ;

where pt is a POINT structure.

The following code draws a grid in the client area of a window, spacing the lines 100 pixels apart starting from the upper left corner. The variable hwnd is assumed to be a handle to the window, hdc is a handle to the device context, and x and y are integers:

```
GetClientRect (hwnd, &rect) ;
for (x = 0 ; x < rect.right ; x+= 100)
{
    MoveToEx (hdc, x, 0, NULL) ;
    LineTo (hdc, x, rect.bottom) ;
}
for (y = 0 ; y < rect.bottom ; y += 100)
{
    MoveToEx (hdc, 0, y, NULL) ;
    LineTo (hdc, rect.right, y) ;
}
```

Although it seems like a nuisance to be forced to use two functions to draw a single line, the current position comes in handy when you want to draw a series of connected lines. For instance, you might want to define an array of 5 points (10 values) that define the outline of a rectangle:

```
POINT apt[5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 } ;
```

Notice that the last point is the same as the first. Now you need only use MoveToEx for the first point and LineTo for the successive points:

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
for (i = 1 ; i < 5 ; i++)
    LineTo (hdc, apt[i].x, apt[i].y) ;
```

Because LineTo draws from the current position up to (but not including) the point in the LineTo function, no coordinate gets written twice by this code. While overwriting points is not a problem with a video display, it might not look good on a plotter or with some drawing modes that I'll discuss later in this chapter.

When you have an array of points that you want connected with lines, you can draw the lines more easily using the Polyline function. This statement draws the same rectangle as in the code shown above:

```
Polyline (hdc, apt, 5) ;
```

The last argument is the number of points. We could also have represented this value by sizeof (apt) / sizeof (POINT). Polyline has the same effect on drawing as an initial MoveToEx followed by multiple LineTo functions. However, Polyline doesn't use or change the current position. PolylineTo is a little different. This function uses the current position for the starting point and sets the current position to the end of the last line drawn. The code below draws the same rectangle as that last shown above:

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
PolylineTo (hdc, apt + 1, 4) ;
```

Although you can use Polyline and PolylineTo to draw just a few lines, the functions are most useful when you need to draw a complex curve. You do this by using hundreds or even thousands of very short lines.

e) Explain Bounding Box Functions.

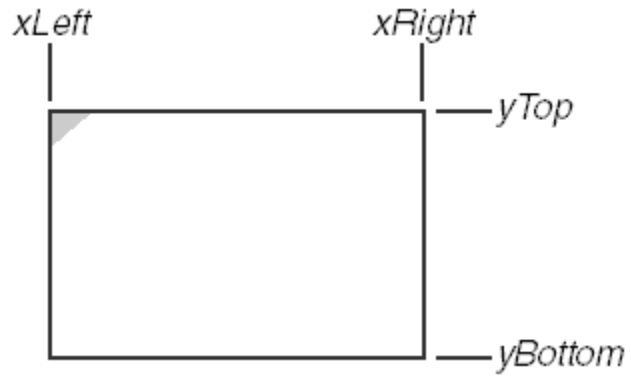
(4 marks for any four functions with syntax and argument descriptions from the following)

The Rectangle, Ellipse, RoundRect, Chord, and Pie functions are not strictly line-drawing functions. Yes, the functions draw lines, but they also fill an enclosed area with the current area-filling brush. This brush is solid white by default, so it may not be obvious that these functions do more than draw lines when you first begin experimenting with them.

The functions listed above are all similar in that they are built up from a rectangular "bounding box." You define the coordinates of a box that encloses the object—the bounding box—and Windows draws the object within this box.

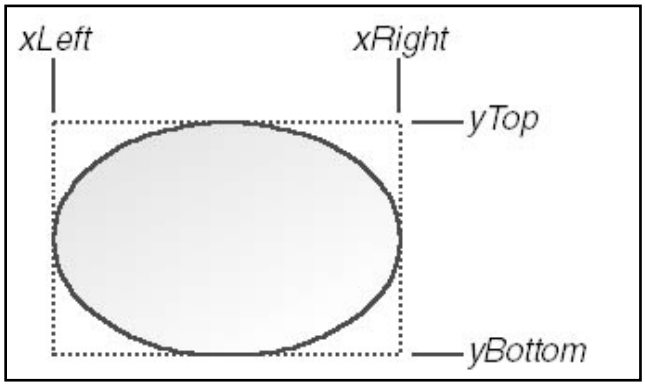
The simplest of these functions draws a rectangle:

Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;



The point (xLeft, yTop) is the upper left corner of the rectangle, and (xRight, yBottom) is the lower right corner. A figure drawn using the Rectangle function is shown in Figure 5-8. The sides of the rectangle are always parallel to the horizontal and vertical sides of the display.

Ellipse (hdc, xLeft, yTop, xRight, yBottom) ;



The function to draw rectangles with rounded corners uses the same bounding box as the Rectangle and Ellipse functions but includes two more arguments:

RoundRect (hdc, xLeft, yTop, xRight, yBottom,
xCornerEllipse, yCornerEllipse) ;

The Arc, Chord, and Pie functions all take identical arguments:

Arc (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;

Chord (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;

Pie (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;

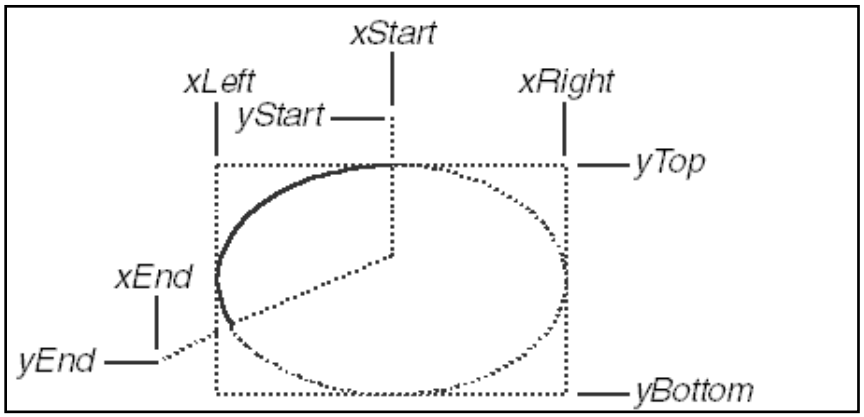


Fig. Arc

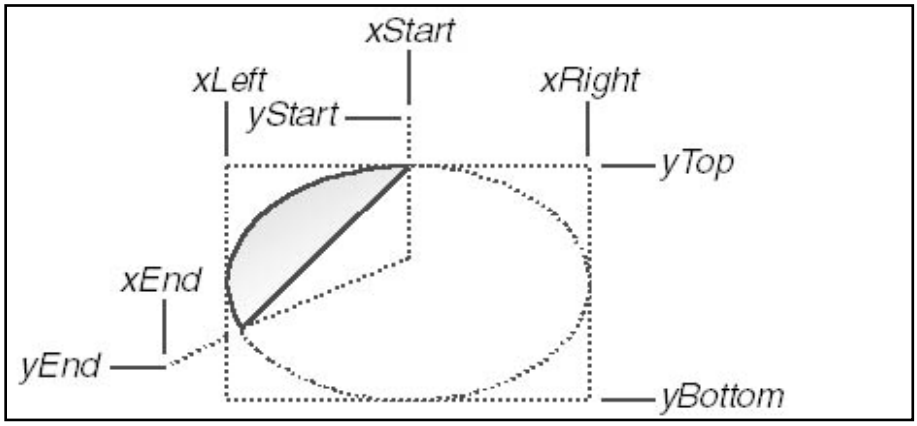


Fig. Chord

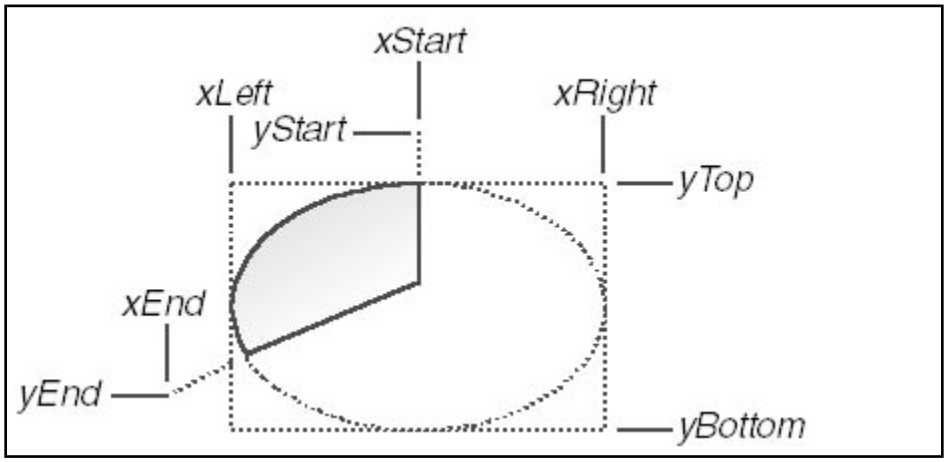


Fig. Pie

f) Explain Windows string functions.

(1 mark for explanation and 3 marks for six functions with description of arguments)

String functions defined in Windows that calculate string lengths, copy strings, concatenate strings, and compare strings:

ILength = lstrlen (pString) ;

pString = lstrcpy (pString1, pString2) ;

pString = lstrcpyn (pString1, pString2, iCount) ;

pString = lstrcat (pString1, pString2) ;

iComp = lstrcmp (pString1, pString2) ;

iComp = lstrcmpi (pString1, pString2) ;

These work much the same as their C library equivalents. They accept wide-character strings if the UNICODE identifier is defined and regular strings if not. The wide-character version of the *lstrlenW* function is implemented in Windows 98.

3. Attempt any four : [Marks 16]

a) State the foreign language keyboard problem.

(4 marks for the following answer)

The Foreign-Language Keyboard Problem refers to the problem where keyboard layout changing is not supported by GDI leading to incorrect symbols being generated when keys are pressed.

Example: KEYLOOK1 will display incorrect characters—when running the English version of Windows with the Russian or Greek keyboard layouts installed, when running the Greek version of Windows with the Russian or German keyboard layouts installed, and when running the Russian version of Windows with the German, Russian, or Greek keyboards installed. Also when entering characters from the Input Method Editor in the Japanese version of Windows.

b) How GDI function calls can be classified?

(4 marks for four classes)

The several hundred function calls that comprise GDI can be classified in several broad groups:

- Functions that get (or create) and release (or destroy) a device context As we saw in earlier chapters, you need a handle to a device context in order to draw. The BeginPaint and EndPaint functions (although technically a part of the USER module rather than the GDI module) let you do this during the

WM_PAINT message, and GetDC and ReleaseDC functions let you do this during other messages. We'll examine some other functions regarding device contexts shortly.

- Functions that obtain information about the device context The GetTextMetrics function to obtain information about the dimensions of the font currently selected in the device context.
- Functions that draw something we use the TextOut function to display some text in the client area of the window. Other GDI functions let us draw lines and filled areas.
- Functions that set and get attributes of the device context An "attribute" of the device context determines various details regarding how the drawing functions work. For example, you can use SetTextColor to specify the color of any text you draw using TextOut or other text output functions. We used SetTextAlign to tell GDI that the starting position of the text string in the TextOut function should be the right side of the string rather than the left, which is the default. All attributes of the device context have default values that are set when the device context is obtained. For all Set functions, there are Get functions that let you obtain the current device context attributes.
- Functions that work with GDI "objects" Here's where GDI gets a bit messy. First an example: By default, any lines you draw using GDI are solid and of a standard width. You may wish to draw thicker lines or use lines composed of a series of dots or dashes. The line width and this line style are not attributes of the device context. Instead, they are characteristics of a "logical pen." You can think of a pen as a collection of bundled attributes. You create a logical pen by specifying these characteristics in the CreatePen, CreatePenIndirect, or ExtCreatePen function. Although these functions are considered to be part of GDI, unlike most GDI functions they do not require a handle to a device context. The functions return a handle to a logical pen. To use this pen, you "select" the pen handle into the device context. The current pen selected in the device context is considered an attribute of the device context. From then on, whatever lines you draw use this pen. Later on, you deselect the pen object from the device context and destroy the object. Destroying the pen is necessary because the pen definition occupies allocated memory space. Besides pens, you also use GDI objects for creating brushes that fill enclosed areas, for fonts, for bitmaps, and for other aspects of GDI.

c) Describe the term Dynamic Link Library.

(Any Four relevant point – 1 mark each)

1. Dynamic-link library (also written unhyphenated), or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems.

2. These libraries usually have the file extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers).
3. The file formats for DLLs are the same as for Windows EXE files — that is, Portable Executable (PE) for 32-bit and 64-bit Windows, and New Executable (NE) for 16-bit Windows. As with EXEs, DLLs can contain code, data, and resources, in any combination.
4. Data files with the same file format as a DLL, but with different file extensions and possibly containing only resource sections, can be called *resource DLLs*.
5. Examples of such DLLs include *icon libraries*, sometimes having the extension `ICL`, and font files, having the extensions FON and FOT.

d) Write the program to display shortest message.

(Any other program can also be written)

```
/*--------------------------------------------------------
HELLOWIN.C -- Displays "Hello, Windows!" in client area
--------------------------------------------------------*/
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
static char szAppName[] = "HelloWin" ;
HWND hwnd ;
MSG msg ;
WNDCLASSEX wndclass ;
wndclass.cbSize = sizeof (wndclass) ;
wndclass.style = CS_HREDRAW | CS_VREDRAW ;
wndclass.lpfnWndProc = WndProc ;
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = szAppName ;
wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;
RegisterClassEx (&wndclass) ;

hwnd = CreateWindow (szAppName, // window class name
"The Hello Program", // window caption
WS_OVERLAPPEDWINDOW, // window style
CW_USEDEFAULT, // initial x position
CW_USEDEFAULT, // initial y position
CW_USEDEFAULT, // initial x size
```

```
CW_USEDEFAULT, // initial y size
NULL, // parent window handle
NULL, // window menu handle
hInstance, // program instance handle
NULL) ; // creation parameters
ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;
while (GetMessage (&msg, NULL, 0, 0))
{
TranslateMessage (&msg) ;
DispatchMessage (&msg) ;
}
return msg.wParam ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
HDC hdc ;
PAINTSTRUCT ps ;
        RECT rect ;

switch (iMsg)
{
case WM_CREATE :
return 0 ;
case WM_PAINT :
hdc = BeginPaint (hwnd, &ps) ;
GetClientRect (hwnd, &rect) ;
DrawText (hdc, "Hello, Windows!", -1, &rect,
DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
EndPaint (hwnd, &ps) ;
return 0 ;
case WM_DESTROY :
PostQuitMessage (0) ;
return 0 ;
}
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
        }
```

e) Explain the term – the child talks to its parent.

(Any four relevant point – 1 mark each)

[Note: The question should have been "Explain the term –The Parent Talks to Its Child"]

A window procedure can also send messages to the child window control. These messages include many of the window messages beginning with the prefix WM. In addition, eight button-specific messages are

defined in WINUSER.H; each begins with the letters BM, which stand for "button message." These button messages are shown in the following table:

Button Message Value

BM_GETCHECK 0x00F0

BM_SETCHECK 0x00F1

BM_GETSTATE 0x00F2

BM_SETSTATE 0x00F3

BM_SETSTYLE 0x00F4

BM_CLICK 0x00F5

BM_GETIMAGE 0x00F6

BM_SETIMAGE 0x00F7

The BM_GETCHECK and BM_SETCHECK messages are sent by a parent window to a child window control to get and set the check mark of check boxes and radio buttons. The BM_GETSTATE and BM_SETSTATE messages refer to the normal, or pushed, state of a window when you click it with the mouse or press it with the Spacebar. The BM_SETSTYLE message lets you change the button style after the button is created.

Each child window has a window handle and an ID that is unique among its siblings. Knowing one of these items allows you to get the other. If you know the window handle of the child, you can obtain the ID using

id = GetWindowLong (hwndChild, GWL_ID) ;

This function (along with SetWindowLong) was used in the CHECKER3 program in Chapter 7 to maintain data in a special area reserved when the window class was registered. The area accessed with the GWL_ID identifier is reserved by Windows when the child window is created. You can also use

id = GetDlgCtrlID (hwndChild) ;

Even though the "Dlg" part of the function name refers to a dialog box, this is really a general-purpose function.

Knowing the ID and the parent window handle, you can get the child window handle:

hwndChild = GetDlgItem (hwndParent, id) ;

f) What are GDI object? Write procedure to create Green Brush.

(2 marks for GDI and 2 marks for procedure)

A logical pen is a GDI Object one of six GDI Objects a program can create. The other five are brushes, bitmaps, regions, fonts, and palettes. Except for palettes, all of these objects are selected into the device context using *SelectObject*.

The procedure to create **Green Brush**

```
hBrush = CreateSolidBrush (RGB(0,255,0)) ;
```

4. Attempt any two: [Marks 16]

a) What is Device Context? Write the methods to get Device Context Handle.

(2 marks for description, 6 marks for the method)

When you want to draw on a graphics output device such as the screen or printer, you must first obtain a handle to a device context (or DC). In giving your program this handle, Windows is giving you permission to use the device. You then include the handle as an argument to the GDI functions to identify to Windows the device on which you wish to draw.

The device context contains many "attributes" that determine how the GDI functions work on the device. These attributes allow GDI functions to have just a few arguments, such as starting coordinates. The GDI functions do not need arguments for everything else that Windows needs to display the object on the device. For example, when you call TextOut, you need specify in the function only the device context handle, the starting coordinates, the text, and the length of the text. You don't need to specify the font, the color of the text, the color of the background behind the text, or the intercharacter spacing. These are all attributes that are part of the device context. When you want to change one of these attributes, you call a function that does so. Subsequent TextOut calls to that device context use the new attribute.

The most common method for obtaining a device context handle and then releasing it involves using the BeginPaint and EndPaint calls when processing the WM_PAINT message:

```
hdc = BeginPaint (hwnd, &ps) ;
[other program lines]
EndPaint (hwnd, &ps) ;
```

The variable ps is a structure of type PAINTSTRUCT. The hdc field of this structure is the same handle to the device context that BeginPaint returns. The PAINSTRUCT structure also contains a RECT (rectangle) structure named rcPaint that defines a rectangle encompassing the invalid region of the

window's client area. With the device context handle obtained from BeginPaint you can draw only within this region. The BeginPaint call also validates this region.

Windows programs can also obtain a handle to a device context while processing messages other than WM_PAINT:

```
hdc = GetDC (hwnd) ;
[other program lines]
ReleaseDC (hwnd, hdc) ;
```

This device context applies to the client area of the window whose handle is hwnd. The primary difference between the use of these calls and the use of the BeginPaint and EndPaint combination is that you can draw on your entire client area with the handle returned from GetDC. However, GetDC and ReleaseDC don't validate any possibly invalid regions of the client area.

A Windows program can also obtain a handle to a device context that applies to the entire window and not only to the window's client area:

```
hdc = GetWindowDC (hwnd) ;
[other program lines]
ReleaseDC (hwnd, hdc) ;
```

This device context includes the window title bar, menu, scroll bars, and frame in addition to the client area. Applications programs rarely use the GetWindowDC function. If you want to experiment with it, you should also trap the WM_NCPAINT ("nonclient paint") message, which is the message Windows uses to draw on the nonclient areas of the window.

The BeginPaint, GetDC, and GetWindowDC calls obtain a device context associated with a particular window on the video display. A much more general function for obtaining a handle to a device context is CreateDC:

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData) ;
[other program lines]
DeleteDC (hdc) ;
```

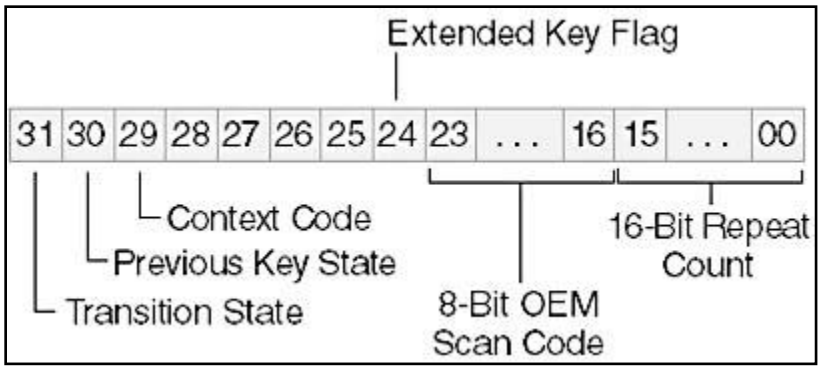
b) Describe the six keystroke message fields of the lParam variable.

(Diagram 2 marks, six keystroke 1 mark each)

For the four keystroke messages the 32 bit lParam variable is set and it contains six fields (coded in the bits):

- Repeat count (bit 0-15): number of keystrokes
- OEM scan code (bit 16-23): the hardware generated keystroke code

- Extended key flag (bit 24): if the keystroke is one of the IBM Extended Keyboard
- Context code (bit 29): 1 if Alt pressed
- Previous key state (bit 30): 0 if the key was previously up and 1 if it was down
- Transition state (bit 31): 0 for KEYDOWN and 1 for KEYUP.



c) Write a Windows Program to put and extract the string in the list box.

(8 marks for the program like below with step marks).

```
/*----------------------------------------
  ENVIRON.C -- Environment List Box
           (c) Charles Petzold, 1998
 ----------------------------------------*/

#include <windows.h>

#define ID_LIST     1
#define ID_TEXT     2

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("Environ") ;
    HWND         hwnd ;
    MSG          msg ;
    WNDCLASS     wndclass ;

    wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc   = WndProc ;

   wndclass.cbClsExtra    = 0 ;
```

```
    wndclass.cbWndExtra    = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Environment List Box"),
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

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

void FillListBox (HWND hwndList)
{
    int     iLength ;
    TCHAR * pVarBlock, * pVarBeg, * pVarEnd, * pVarName ;

    pVarBlock = GetEnvironmentStrings () ;  // Get pointer to environment block

    while (*pVarBlock)
    {
        if (*pVarBlock != `=`)   // Skip variable names beginning with `=`
        {
            pVarBeg = pVarBlock ;              // Beginning of variable name
            while (*pVarBlock++ != `=`) ;      // Scan until `=`
            pVarEnd = pVarBlock - 1 ;          // Points to `=` sign
            iLength = pVarEnd - pVarBeg ;      // Length of variable name
```

```
                // Allocate memory for the variable name and terminating
                // zero. Copy the variable name and append a zero.

            pVarName = calloc (iLength + 1, sizeof (TCHAR)) ;
            CopyMemory (pVarName, pVarBeg, iLength * sizeof (TCHAR)) ;
            pVarName[iLength] = `\0' ;

                // Put the variable name in the list box and free memory.
            SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) pVarName) ;
            free (pVarName) ;
        }
        while (*pVarBlock++ != `\0') ;     // Scan until terminating zero
    }
    FreeEnvironmentStrings (pVarBlock) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    static HWND  hwndList, hwndText ;
    int          iIndex, iLength, cxChar, cyChar ;
    TCHAR      * pVarName, * pVarValue ;

    switch (message)
    {
    case WM_CREATE :
          cxChar = LOWORD (GetDialogBaseUnits ()) ;
 cyChar = HIWORD (GetDialogBaseUnits ()) ;

     // Create listbox and static text windows.

 hwndList = CreateWindow (TEXT ("listbox"), NULL,
                WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                cxChar, cyChar * 3,
                cxChar * 16 + GetSystemMetrics (SM_CXVSCROLL),
                cyChar * 5,
                hwnd, (HMENU) ID_LIST,
                (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                NULL) ;

 hwndText = CreateWindow (TEXT ("static"), NULL,
                WS_CHILD | WS_VISIBLE | SS_LEFT,
                cxChar, cyChar,
                GetSystemMetrics (SM_CXSCREEN), cyChar,
                hwnd, (HMENU) ID_TEXT,
                (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                NULL) ;

        FillListBox (hwndList) ;
```

```
        return 0 ;

    case WM_SETFOCUS :
        SetFocus (hwndList) ;
        return 0 ;
    case WM_COMMAND :
 if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_SELCHANGE)
 {

// Get current selection.

     iIndex  = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
     iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) + 1 ;
     pVarName = calloc (iLength, sizeof (TCHAR)) ;
     SendMessage (hwndList, LB_GETTEXT, iIndex, (LPARAM) pVarName) ;

         // Get environment string.

     iLength = GetEnvironmentVariable (pVarName, NULL, 0) ;
     pVarValue = calloc (iLength, sizeof (TCHAR)) ;
     GetEnvironmentVariable (pVarName, pVarValue, iLength) ;

         // Show it in window.

     SetWindowText (hwndText, pVarValue) ;
     free (pVarName) ;
     free (pVarValue) ;
 }
        return 0 ;

    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

5. Attempt any four: **[Marks 16]**

a) Explain polygon filling methods.

(2 marks for each method)

There are two polygon filling methods as follows

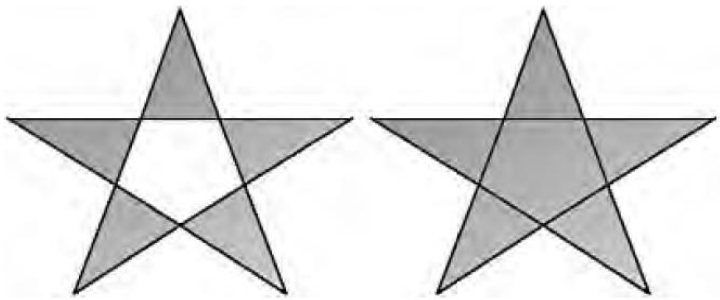
1) Alternate mode:-

In this mode you can imagine a line drawn from a point in an enclosed area to infinity. The enclosed area is filled only if that imaginary line crosses an odd number of boundary lines. This is why the points of the star are filled but the center is not.

2) Winding Mode: -

The example of the five-pointed star makes winding mode seem simpler than it actually is. When you're drawing a single polygon, in most cases winding mode will cause all enclosed areas to be filled. But there are exceptions. To determine whether an enclosed area is filled in winding mode, you again imagine a line drawn from a point in that area to infinity. If the imaginary line crosses an

odd number of boundary lines, the area is filled, just as in alternate mode. If the imaginary line crosses an even number of boundary lines, the area can either be filled or not filled. The area is filled if the number of boundary lines going in one direction (relative to the imaginary line) is not equal to the number of boundary lines going in the other direction.



Figures drawn with the two polygon-filling modes: ALTERNATE (left) and WINDING (right).

b) What actions can be taken on mouse buttons? Write a simple program for freehand drawing using mouse.

(1 mark for list of all actions; 3 marks for code any code is expected)

The following terms describe the actions you take with mouse buttons:

- *Clicking* Pressing and releasing a mouse button.
- *Double-clicking* Pressing and releasing a mouse button twice in quick succession.
- *Dragging* Moving the mouse while holding down a button.

```
case WM_LBUTTONDOWN:
    iCount = 0 ;
    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;

case WM_MOUSEMOVE:
    if (wParam & MK_LBUTTON && iCount < 1000)
    {
        pt[iCount  ].x = LOWORD (lParam) ;
        pt[iCount++].y = HIWORD (lParam) ;

        hdc = GetDC (hwnd) ;
        SetPixel (hdc, LOWORD (lParam), HIWORD (lParam), 0) ;
        ReleaseDC (hwnd, hdc) ;
    }
    return 0 ;


case WM_LBUTTONUP:
    InvalidateRect (hwnd, NULL, FALSE) ;
    return 0 ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
    ShowCursor (TRUE) ;

    for (i = 0 ; i < iCount - 1 ; i++)
        for (j = i + 1 ; j < iCount ; j++)
        {
            MoveToEx (hdc, pt[i].x, pt[i].y, NULL) ;
            LineTo   (hdc, pt[j].x, pt[j].y) ;
        }

    ShowCursor (FALSE) ;
    SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
```

```
EndPaint (hwnd, &ps) ;
  return 0 ;
```

c) How can I get the status of shift and toggle keys?

(2 marks for shift status and 2 marks for toggle key status)

Shift key Status:

When you process a keystroke message, you may need to know whether any of the shift keys (Shift, Ctrl, and Alt) or toggle keys (Caps Lock, Num Lock, and Scroll Lock) are pressed. You can obtain this information by calling the *GetKeyState* function. For instance:

```
iState = GetKeyState (VK_SHIFT) ;
```

The *iState* variable will be negative (that is, the high bit is set) if the Shift key is down. The value returned from

```
iState = GetKeyState (VK_CAPITAL) ;
```

has the low bit set if the Caps Lock key is toggled on. This bit will agree with the little light on the keyboard. Generally, you'll use GetKeyState with the virtual key codes VK_SHIFT, VK_CONTROL, and VK_MENU (which you'll recall indicates the Alt key). You can also use the following identifiers with GetKeyState to determine if the left or right Shift, Ctrl, or Alt keys are pressed: VK_LSHIFT, VK_RSHIFT, VK_LCONTROL, VK_RCONTROL, VK_LMENU, VK_RMENU. These identifiers are used only with GetKeyState and GetAsyncKeyState (described below).

You can also obtain the state of the mouse buttons using the virtual key codes VK_LBUTTON, VK_RBUTTON, and VK_MBUTTON. However, most Windows programs that need to monitor a combination of mouse buttons and keystrokes usually do it the other way around by checking keystrokes when they receive a mouse message. In fact, shift-state information is conveniently included in the mouse messages,

Be careful with GetKey State. It is not a real-time keyboard status check. Rather, it reflects the keyboard status up to and including the current message being processed. For the most part, this is exactly what you want. If you need to determine if the user typed Shift-Tab, you can call GetKeyState with the VK_SHIFT parameter while processing the WM_KEYDOWN message for the Tab key. If the return value of Ge tKey State is negative, you know that the Shift key was pressed *before* the Tab key. And it doesn't matter if the Shift key has already been released by the time you get around to processing the Tab key. You know that the Shift key was down when Tab was pressed. GetKeyState does not let you

retrieve keyboard information independent of normal keyboard messages. For instance, you may feel a need to hold up processing in your window procedure until the user presses the F1 function key:

```
while (GetKeyState (VK_F1) >= 0) ; // WRONG !!!
```

Don't do it! This is guaranteed to hang your program (unless, of course, the WM_KEYDOWN message for F1 was retrieved from the message queue before you executed the statement). If you really need to know the current real-time state of a key, you can use GetAsyncKeyState.

Toggle key status:

When you process a keystroke message, you may need to know whether any of the shift keys (Shift, Ctrl, and Alt) or toggle keys (Caps Lock, Num Lock, and Scroll Lock) are pressed. You can obtain this information by calling theGetKeyState function. For instance:

```
iState = GetKeyState (VK_SHIFT) ;
```

The iState variable will be negative (that is, the high bit is set) if the Shift key is down. The value returned from

```
iState = GetKeyState (VK_CAPITAL) ;
```

has the low bit set if the Caps Lock key is toggled on. This bit will agree with the little light on the keyboard. Generally, you'll use GetKeyState with the virtual key codes VK_SHIFT, VK_CONTROL, and VK_MENU (which you'll recall indicates the Alt key). You can also use the following identifiers with GetKeyState to determine if the left or right Shift, Ctrl, or Alt keys are pressed: VK_LSHIFT, VK_RSHIFT, VK_LCONTROL, VK_RCONTROL, VK_LMENU, VK_RMENU. These identifiers are used only with GetKeyState and GetAsyncKeyState.

d) What is mouse capturing? Give the solution.

(2 marks for mouse capturing 2 marks for solution)

A window procedure normally receives mouse messages only when the mouse cursor is positioned over the client or nonclient area of the window. A program might need to receive mouse messages when the mouse is outside the window. If so, the program can "capture" the mouse.

This is the type of problem for which mouse capturing was invented. If the user is dragging the mouse, it should be no big deal if the cursor drifts out of the window for a moment. The program should still be in control of the mouse. Capturing the mouse is easier than baiting a mousetrap. One can only call

```
SetCapture (hwnd) ;
```

After this function call Windows sends all mouse messages to the window procedure for the window whose handle is hwnd. The mouse messages always come through as client-area messages, even when

the mouse is in a nonclient area of the window. The lParam parameter still indicates the position of the mouse in client-area coordinates. These *x* and *y* coordinates, however, can be negative if the mouse is to the left of or above the client area. When you want to release the mouse, call

ReleaseCapture () ;

```
    case WM_LBUTTONDOWN :
ptBeg.x = ptEnd.x = LOWORD (lParam) ;
ptBeg.y = ptEnd.y = HIWORD (lParam) ;

DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

SetCapture (hwnd) ;
SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

fBlocking = TRUE ;
return 0 ;

    case WM_MOUSEMOVE :
if (fBlocking)
{
    SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

    ptEnd.x = LOWORD (lParam) ;
    ptEnd.y = HIWORD (lParam) ;

    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
}
return 0 ;

case WM_LBUTTONUP :
if (fBlocking)
{
    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

    ptBoxBeg   = ptBeg ;
    ptBoxEnd.x = LOWORD (lParam) ;
    ptBoxEnd.y = HIWORD (lParam) ;

    ReleaseCapture () ;
    SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

    fBlocking = FALSE ;
    fValidBox  = TRUE ;
```

```
        InvalidateRect (hwnd, NULL, TRUE) ;
}
return 0 ;
```

e) Explain Registering Window Class.

(2 marks for WNDCLASS and 2 marks for code snippet)

Before you create an application window, you must register a window class by calling *RegisterClass*. This function requires a single parameter, which is a pointer to a structure of type WNDCLASS. This structure includes two fields that are pointers to character strings, so the structure is defined two different ways in the WINUSER.H header file. First, there's the ASCII version, WNDCLASSA:

```
typedef struct tagWNDCLASSA
{
UINT style ;
WNDPROC lpfnWndProc ;
int cbClsExtra ;
int cbWndExtra ;
HINSTANCE hInstance ;
HICON hIcon ;
HCURSOR hCursor ;
HBRUSH hbrBackground ;
LPCSTR lpszMenuName ;
LPCSTR lpszClassName ;
}
WNDCLASSA, * PWNDCLASSA, NEAR * NPWNDCLASSA, FAR * LPWNDCLASSA ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("HelloWin") ;

  HWND        hwnd ;
    MSG         msg ;
    WNDCLASS    wndclass ;

    wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc   = WndProc ;
    wndclass.cbClsExtra    = 0 ;
    wndclass.cbWndExtra    = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName  = NULL ;

   wndclass.lpszClassName = szAppName ;
```

```
if (!RegisterClass (&wndclass))
{
    MessageBox (NULL, TEXT ("This program requires Windows NT!"),
            szAppName, MB_ICONERROR) ;
    return 0 ;
}
```

f) What are system colors?

(2 marks for system colors; 2 marks for table any for getsyscolor / setsyscolor are expected)

System Colors:

Windows maintains 29 system colors for painting various parts of the display. You can obtain and set these colors using *GetSysColor* and *SetSysColors*. Identifiers defined in the windows header files specify the system color. Setting a system color with SetSysColors changes it only for the current Windows session.

One can change some (but not all) system colors using the Display section of the Windows Control Panel. The selected colors are stored in the Registry in Microsoft Windows NT and in the WIN.INI file in Microsoft Windows 98.

The Registry and WIN.INI file use keywords for the 29 system colors (different from the GetSysColor and SetSysColors identifiers), followed by red, green, and blue values that can range from 0 to 255. The following table shows how the 29 system colors are identified applying the constants used for GetSysColor and SetSysColors and also the WIN.INI keywords. The table is arranged sequentially by the values of the COLOR_ constants, beginning with 0 and ending with 28.

GetSysColor and SetSysColors	Registry Key or WIN.INI Identifer	Default RGB Value
COLOR_SCROLLBAR	Scrollbar	C0-C0-C0
COLOR_BACKGROUND	Background	00-80-80
COLOR_ACTIVECAPTION	ActiveTitle	00-00-80
COLOR_INACTIVECAPTION	InActiveTitle	08-80-80
COLOR_MENU	Menu	C0-C0-C0
COLOR_WINDOW	Window	FF-FF-FF
COLOR_WINDOWFRAME	WindowFrame	00-00-00
COLOR_MENUTEXT	MenuText	C0-C0-C0

6. Attempt any two: **[Marks 16]**

a) How windows sends mouse messages to the program?

(4 marks for explanation 4 marks for syntax and functions)

A window procedure receives mouse messages whenever the mouse passes over the window or is clicked within the window, even if the window is not active or does not have the input focus. Windows defines 21 messages for the mouse. However, 11 of these messages do not relate to the client area. These are called "nonclient-area messages," and Windows applications usually ignore them. When the mouse is moved over the client area of a window, the window procedure receives the message WM_MOUSEMOVE. When a mouse button is pressed or released within the client area of a window, the window procedure receives the messages in this table:

Button	*Pressed*	*Released*	*Pressed (Second Click)*
Left	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Middle	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Right	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

Window procedure receives MBUTTON messages only for a three-button mouse and RBUTTON messages only for a two-button mouse. The window procedure receives DBLCLK (double-click) messages only if the window class has been defined to receive them

For all these messages, the value of lParam contains the position of the mouse. The low word is the *x*-coordinate, and the high word is the *y*-coordinate relative to the upper left corner of the client area of the window. You can extract these values using the LOWORD and HIWORD macros:

x = LOWORD (lParam) ;

y = HIWORD (lParam) ;

The value of wParam indicates the state of the mouse buttons and the Shift and Ctrl keys. You can test wParam using these bit masks defined in the WINUSER.H header file. The MK prefix stands for "mouse key."

MK_LBUTTON Left button is down

MK_MBUTTON Middle button is down

MK_RBUTTON Right button is down

MK_SHIFT Shift key is down

MK_CONTROL Ctrl key is down

For example, if you receive a WM_LBUTTONDOWN message, and if the value

wparam & MK_SHIFT

Is TRUE (nonzero), you know that the Shift key was down when the left button was pressed. As you move the mouse over the client area of a window, Windows does not generate a WM_MOUSEMOVE message for every possible pixel position of the mouse. The number of WM_MOUSEMOVE messages

your program receives depends on the mouse hardware and on the speed at which your window procedure can process the mouse movement messages.

The rate of WM_MOUSEMOVE messages when you experiment with the CONNECT program described below. If you click the left mouse button in the client area of an inactive window, Windows changes the active window to the window that is being clicked and then passes the WM_LBUTTONDOWN message to the window procedure. When your window procedure gets a WM_LBUTTONDOWN message, your program can safely assume the window is active. However, your window procedure can receive a WM_LBUTTONUP message without first receiving a WM_LBUTTONDOWN message. This can happen if the mouse button is pressed in one window, moved to your window, and released. Similarly, the window procedure can receive a WM_LBUTTONDOWN without a corresponding WM_LBUTTONUP message if the mouse button is released while positioned over another window.

b) Explain how to put strings in the list box. Describe the list box styles.

(5 marks for putting strings in the list box and 3 marks for list box style)

Putting Strings in the List Box

After you've created the list box, the next step is to put text strings in it. You do this by sending messages to the list box window procedure using the SendMessage call. The text strings are generally referenced by an index number that starts at 0 for the topmost item. In the examples that follow, hwndList is the handle to the child window list box control, and iIndex is the index value. In cases where you pass a text string in the SendMessage call, the lParam parameter is a pointer to a null-terminated string.

In most of these examples, the SendMessage call can return LB_ERRSPACE (defined as -2) if the window procedure runs out of available memory space to store the contents of the list box. SendMessage returns LB_ERR (-1) if an error occurs for other reasons and LB_OKAY (0) if the operation is successful. You can test SendMessage for a nonzero value to detect either of the two errors.

If you use the LBS_SORT style (or if you are placing strings in the list box in the order that you want them to appear), the easiest way to fill up a list box is with the LB_ADDSTRING message:

```
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) szString) ;
```

If you do not use LBS_SORT, you can insert strings into your list box by specifying an index value with LB_INSERTSTRING:

```
SendMessage (hwndList, LB_INSERTSTRING, iIndex, (LPARAM) szString) ;
```

For instance, if iIndex is equal to 4, szString becomes the new string with an index value of 4 the fifth string from the top because counting starts at 0. Any strings below this point are pushed down. An iIndex value of -1 adds the string to the bottom. You can use LB_INSERTSTRING with list boxes that have the LBS_SORT style, but the list box contents will not be re-sorted.

You can delete a string from the list box by specifying the index value with the LB_DELETESTRING message:

```
SendMessage (hwndList, LB_DELETESTRING, iIndex, 0) ;
```

You can clear out the list box by using LB_RESETCONTENT:

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
```

The list box window procedure updates the display when an item is added to or deleted from the list box. If you have a number of strings to add or delete, you may want to temporarily inhibit this action by turning off the control's redraw flag:

```
SendMessage (hwndList, WM_SETREDRAW, FALSE, 0) ;
```

After you've finished, you can turn the redraw flag back on:

```
SendMessage (hwndList, WM_SETREDRAW, TRUE, 0) ;
```

A list box created with the LBS_NOREDRAW style begins with the redraw flag turned off.

You create a list box child window control with CreateWindow using "listbox" as the window class and WS_CHILD as the window style. However, this default list box style does not send WM_COMMAND messages to its parent, meaning that a program would have to interrogate the list box (via messages to the list box controls) regarding the selection of items within the list box. Therefore, list box controls almost always include the list box style identifier LBS_NOTIFY, which allows the parent window to receive WM_COMMAND messages from the list box. If you want the list box control to sort the items in the list box, you can also use LBS_SORT, another common style.

By default, list boxes are single selection. Multiple-selection list boxes are relatively rare. If you want to create one, you use the style LBS_MULTIPLESEL. Normally, a list box updates itself when a new item is added to the scroll box list. You can prevent this by including the style LBS_NOREDRAW. You will probably not want to use this style, however. Instead, you can temporarily prevent the repainting of a list box control by using the WM_SETREDRAW message that I'll describe a little later.

By default, the list box window procedure displays only the list of items without any border around it. You can add a border with the window style identifier WS_BORDER. And to add a vertical scroll bar for scrolling through the list with the mouse, you use the window style identifier WS_VSCROLL.

c) Explain the functions:

i) Create Window( )

ii) Show Window( )

iii) Update Window( )

iv) Message Box( )

(2 marks for each function)

Create Window ( )

To create a window call Create Window() function. The information passed to the Register Class function is specified in a data structure, the information passed to the Create Window function is specified as separate arguments to the function.

```
hwnd = CreateWindow (szAppName,          // window class name
TEXT ("The Hello Program"),              // window caption
WS_OVERLAPPEDWINDOW,                     // window style
CW_USEDEFAULT,                           // initial x position
CW_USEDEFAULT,                           // initial y position
CW_USEDEFAULT,                           // initial x size
CW_USEDEFAULT,                           // initial y size
NULL,                                    // parent window handle
NULL,                                    // window menu handle
hInstance,                               // program instance handle
NULL) ;                                  // creation parameters
```

Show Window ( )

The ShowWindow function puts the window on the display. If the second argument to *ShowWindow* is SW_SHOWNORMAL, the client area of the window is erased with the background brush specified in the window class.

```
ShowWindow (hwnd, iCmdShow) ;
```

The first argument is the handle to the window just created by CreateWindow. The second argument is the iCmdShow value passed as a parameter to WinMain. This determines how the window is to be initially displayed on the screen, whether it's normal, minimized, or maximized. The user probably selected a preference when adding the program to the Start menu. The value you receive from WinMain and pass to ShowWindow is SW_SHOWNORMAL if the window is displayed normally,

SW_SHOWMAXIMIZED if the window is to be maximized, and SW_SHOWMINNOACTIVE if the window is just to be displayed in the taskbar.

Update Window ( )

The function call causes the client area to be painted. It accomplishes this by sending the window procedure a WM_PAINT message.

```
UpdateWindow (hwnd) ;
```

Message Box ()

The MessageBox function is designed to display short messages. The little window that MessageBox displays is actually considered to be a dialog box, although not one with a lot of versatility.

The first argument to MessageBox is normally a window handle. The second argument is the text string that appears in the body of the message box, and the third argument is the text string that appears in the caption bar of the message box. The fourth argument to MessageBox can be a combination of constants beginning with the prefix MB_ that are defined in WINUSER.H. You can pick one constant from the first set to indicate what buttons you wish to appear in the dialog box:

```
#define MB_OK                   0x00000000L
```

You can also use a constant that indicates the appearance of an icon in the message box:

```
#define MB_ICONHAND             0x00000010L
```

the MessageBox function returns the value 1, but it's more proper to say that it returns IDOK, which is defined in WINUSER.H as equaling 1. Depending on the other buttons present in the message box, the MessageBox function can also return IDYES, IDNO, IDCANCEL, IDABORT, IDRETRY, or IDIGNORE.

```
  /*------------------------------------------------------------
  HelloMsg.c -- Displays "Hello, Windows 98!" in a message box
            (c) Charles Petzold, 1998
 ------------------------------------------------------------*/

#include <windows.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                PSTR szCmdLine, int iCmdShow)
{
    MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0) ;

    return 0 ;
}
```