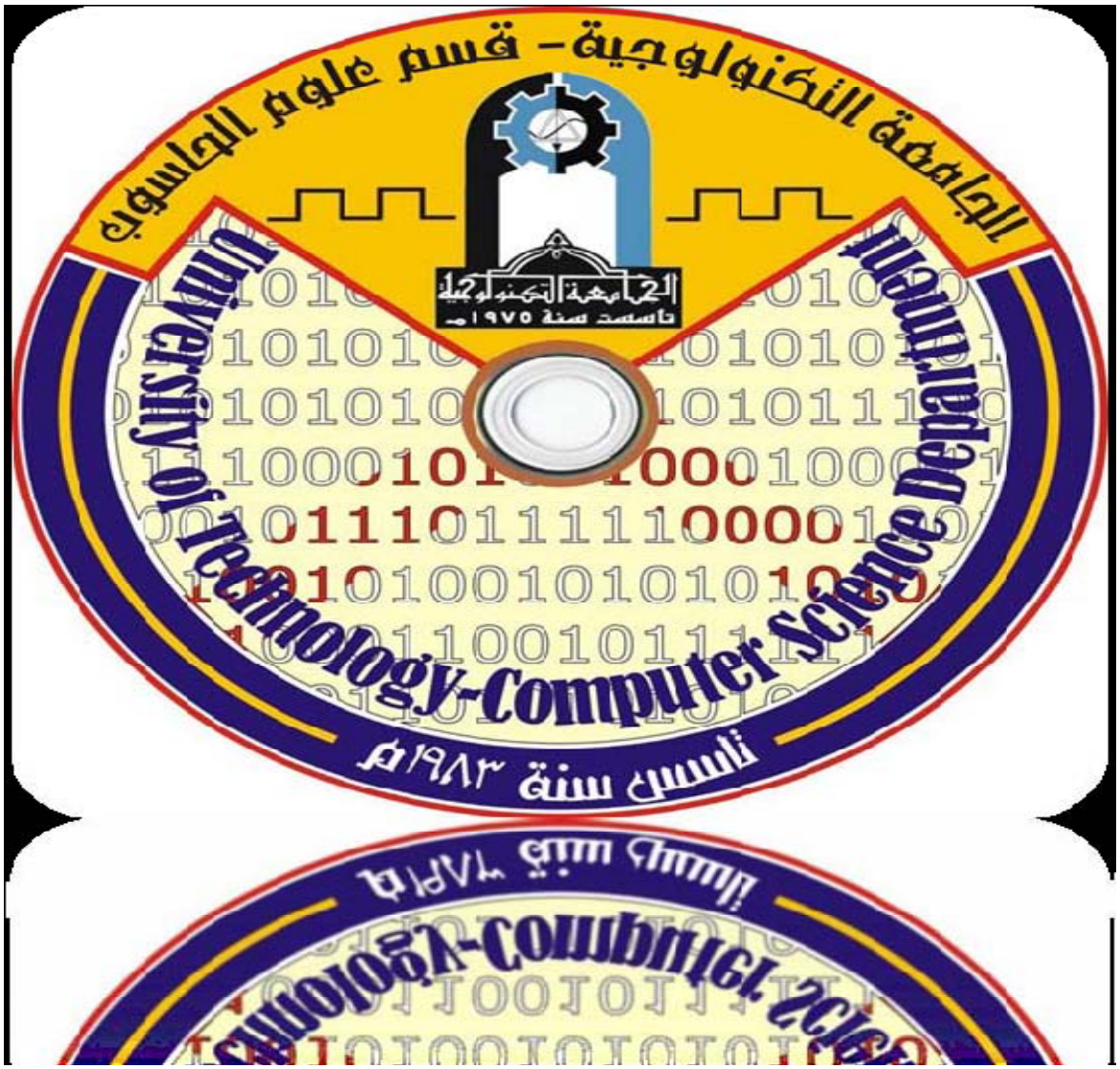


4th (Programmatic Branch)

Advance Windows Programming class

برمجة نوافذ المتقدمة

أعداد :م .علي حسن حمادي 2006 .... 2010 <==< 2012-2013

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter One:

Window NT Programming Fundamentals

CONTENTS:

The Components of a Window

Before moving on to specific aspects of Windows NT programming, a few important terms need to be defined. Figure 3 shows a standard window with each of its elements pointed out. Notice that the style of a window in NT 4 differs slightly from that provided by earlier versions. This is because Windows NT 4 uses the new "Windows 95-style" user interface. All windows have a border that defines the limits of the window and is used to resize the window. At the top of the window are several items. On the far left is the system menu icon (also called the title bar icon). Clicking on this box causes the system menu to be displayed. To the right of the system menu box is the window's title. At the far right are the minimize, maximize, and close boxes. (Versions of Windows NT prior to 4 did not include a close box.) The client area is the part of the window in which your program activity takes place. Windows may also have horizontal and vertical scroll bars that are used to move text through the window.

Figure 3: the element of a standard window

Some Windows NT Application Basics

Before developing the Windows NT application skeleton, some basic concepts common to all Windows NT programs need to be discussed.

WinMain()

All Windows NT programs begin execution with a call to WinMain(). (Windows programs do not have a main( ) function.) WinMain( ) has some special properties that differentiate it from other functions in your application. First, it must be compiled using the WINAPI calling convention. (You will also see APIENTRY used. Currently, they both mean the same thing.) By default, functions in your C or C++ programs use the C calling convention. However, it is possible to compile a function so that it uses a different calling convention.

The calling convention Windows NT uses to call WinMain( ) is WINAPI. The return type of WinMain( ) should be int.

The Window Procedure

All Windows NT programs must contain a special function that is not called by your program, but is called by Windows NT. This function is generally called the window procedure or window function. It is through this function that Windows NT communicates with your program. The window function is called by Windows NT when it needs to pass a message to your program, The window function receives the message in its parameters. All window functions must be declared as returning type LRESULT CALLBACK. The type LRESULT is a typdef that (.it the time of this writing) is another name for a long integer. The CALLBACK calling convention is used with those functions that will be called by Windows NT. In Windows terminology, any function that is called by Windows is referred to as a callback function.

In addition to receiving the messages sent by Windows NT, the window function must initiate any actions indicated by a message. Typically, a window function's body consists of a switch statement that links a special response to each message that the program will respond to. Your program need not respond to every message that Windows NT will send. For messages that your program doesn't care about, you can let Windows NT provide default processing. Since there are hundreds of different messages that Windows NT can generate, it is common for most messages simply to be processed by Windows NT and not your program.

All messages are 32-bit integer values. Further, all messages are accompanied by any additional information that the message requires.

Window Classes

When your Windows NT program first begins execution, it will need to define and register a window class. (Here, the word class is not being used its C++ sense. Rather, it means style or type.) When you register a window class, you are telling Windows NT about the form and function of the window However, registering the window class does not cause a window to come into existence. To actually create a window requires additional steps.

The Message Loop

As explained earlier, Windows NT communicates with your program by sending it messages. All Windows NT applications must establish a message loop inside the

WinMain() function. This loop reads any pending message from the application's message queue and then dispatches that message back to Windows NT, which then calls your program's window function with that message as a parameter. This may seem to be an overly complex way of passing messages, but it is, nevertheless, the way all Windows programs must function. (Part of the reason for this is to return control to Windows NT so that the scheduler can allocate CPU time as it sees fit rather than waiting for your application's time slice to end.)

Windows Data Types

As you will soon see, Windows NT programs do not make extensive use of standard C/C++ data types, such as int or char *. Instead, all data types used by Windows NT have been typdefed within the WINDOWS.H file and/or its subordinate files. This file is supplied by Microsoft (and any other company that makes a Windows NT C/C++ compiler) and must be included in all Windows NT programs. Some of the most common types are HANDLE, HWND, UINT, BYTE, WORD, DWORD, LONG, BOOL, LPSTR, and LPCSTR. HANDLE is a 32-bit integer that is used as a handle. As you will see, there are numerous handle types, but they are all the same size as HANDLE. A handle is simply a value that identifies some resource. For example, HWND is a 32-bit integer that is used as a window handle. Also, all handle types begin with an H. BYTE is an 8-bit unsigned character. WORD is a 16-bit unsigned short integer. DWORD is an unsigned long integer. UINT is an unsigned 32-bit integer. LONG is another name for long. BOOL is an integer. This type is used to indicate values that are either true or false. LPSTR is a pointer to a string and LPCSTR is a const pointer to a string.

In addition to the basic types described above, Windows NT defines several structures. The two that are needed by the skeleton program arc MSG and WNDCLASSEX. The MSG structure holds a Windows NT message and WNDCLASSEX is a structure that defines a window class. These structures will be discussed later in this lecture.

A Windows NT Skeleton

Now that the necessary background information has been covered, it is time to develop a minimal Windows NT application. As stated, all Windows NT programs have certain things in common. In this section a skeleton is developed that provides these necessary features. In the world of Windows programming, application skeletons are commonly used because there is a substantial "price of admission" when creating a Windows program. Unlike DOS

programs, for example, in which a minimal program is about 5 lines long, a minimal Windows program is approximately 50 lines long. A minimal Windows NT program contains two functions: WinMain( ) and the window function. The WinMain() function must perform the following general steps:

1. Define a window class.
2. Register that class with Windows NT.
3. Create a window of that class.
4. Display the window.
5. Begin running the message loop.

The window function must respond to all relevant messages. Since the skeleton program does nothing but display its window, the only message that it must respond to is the one that tells the application that the user has terminated the program.

Before discussing the specifics, examine the following program, which is a minimal Windows NT skeleton. It creates a standard window that includes a title, a system menu, and the standard minimize, maximize, and close boxes. The window is, therefore, capable of being minimized, maximized, moved, resized, and closed.

```
/* A minimal Windows NT skeleton. */
#include <windows.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs, int nWinMode) {HWND hwnd;MSG msg; WNDCLASSEX wcl;
/* Define a window class. */
wcl.cbSize = sizeof(WNDCLASSEX);
wcl.hInstance = hThisInst; /* handle to this instance */
wcl.lpszClassName = szWinName; /* window class name */
wcl.lpfnWndProc = WindowFunc; /* window function */
wcl.style = 0; /* default style */
  wcl.hIcon =LoadIcon(NULL, IDI_APPLICATION); /*standard icon*/
wcl.hIconSm = LoadIcon(NULL, IDI_WINLOGO); /* small icon */
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);/*cursor style*/
wcl.lpszMenuName = NULL; /* no menu"*/
```

```
wcl.cbClsExtra =0; /* no extra */
wcl.cbWndExtra = 0; /* information needed */
/* Make the window background white. */
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
/*Register the window class.*/
if(!RegisterClassEx(&wcl)) return 0;
/* Now that a window class has been registered, a window can be created. */
 hwnd = CreateWindow(
  szWinName, /* name of window class */
  "Windows NT Skeleton", /* title */
  WS_OVERLAPPEDWINDOW, /* window style - normal */
  CW_USEDEFAULT, /* X coordinate - let Windrows decide */
  CW_USEDEFAULT, /* Y coordinate - let Windows decide */
  CW_USEDEFAULT, /* width - let Windows decide */
  CW_USEDEFAULT, /* height - let Windows decide */
  HWND_DESKTOP, /* no parent window */
  NULL,
  hThisInst, /* handle of this instance of the program */
  NULL /* no additional arguments */ );
/* Display the window. */
ShowWindow (hwnd, nWinMode); UpdateWindow(hwnd);
/* create the message loop. */
while (GetMessage(&msg, NULL, 0, 0))
{TranslateMessage(&msg); /* allow use of keyboard */
DispatchMessage(&msg); /* return control to window NT */}
return msg.wParam; }/* end of WinMain() */
 /* This function is called by Windows NT and is passed messages from the MESSAGE QUEUE */
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam) { switch (message)
{case WM_DESTROY:/*terminate the program*/ PostQuitMessage(0);break;
 default:/* Let Window NT process any message not specified in the preceding switch statement.*/
return DefWindowProc(hwnd,message,wParam,lParam);}r eturn 0;} /*end WinFunc */
```

Let's go through this program step by step.

First, all Windows NT program must include the header file WINDOWS.H. As stated, this file (along With its support files) contains the API function prototypes and various typed, macro and definitions used by Windows NT. For example, the data types HWND and WNDCLASSEX are defined in WINDOWS.H (or its subordinate files).

The window function used by the program is called WindowFunc( ). It is declared as a callback function because this is the function that Windows NT calls to communicate with the program.

As stated, program execution begin with WinMain( ). WinMain() is passed four parameter. hThisInst and hPrevInst are handles. hThisInst refers to the current instance of program. Remember, Windows NT is a multitasking system,, so it is possible that more than one instance of your program may be running at the same time. For Windows NT, hPrevInst will always be NULL.

The IpszArgs parameter is a pointer to a string that holds any command line arguments specified when the application was begun. In Windows NT, the string contains the entire command line, including the name of the program itself. The nWinModc parameter contains a value that determines how the window will be displayed when your program begins execution.

Inside the function, three variables are created. The hwnd variable will hold the handle to the program's window. The msg structure variable will hold window messages and the wc1 structure variable will be used to define the window class.

Defining the Window Class

The first two actions that WinMain( ) takes are to define a window class and then register it. A window class is defined by filling in the fields defined by the WNDCLASSEX structure. Its fields are shown here.

```
UINT cbSize; /* size of the WNDCLASSEX structure */
UINT style; /* type of window */
WNDPROC pfnWndProc; /* address to window func */
int cbClsExtra; /* extra class info */
int cbhWndExtra; /* extra window info */
HINSTANCE hInstance; /* handle of this instance */
HICON hIcon; /* handle of standard icon */
HICON hIconSm; /* handle of small icon */
HCURSOR hCursor; /* handle of mouse cursor */
HBRUSH hbrBackground; /* background color */
LPCSTR lpszMenuName; /* name of main menu */
LPCSTR lpszClassName; /* name of window class */
```

As you can see by looking at the program, cbSize is assigned the size of the WNDCLASSEX structure. The hInstance member is assigned the current instance handle as

specified by hThisInst. The name of the window class is pointed to by lpszClassName, which points to the string "MyWin" in this case. The address of the window function is assigned to lpfnWndProc. In the program, no default style is specified, no extra information is needed, and no main menu is specified. While most programs will contain a main menu, none is required by the skeleton. (Menus are described in advance lecture.)

All Windows applications need to define a default shape for the mouse cursor and for the application's icons. An application can define its own custom version of these resources or it may use one of the built-in styles, as the skeleton does. In either case, handles to these resources must be assigned to the appropriate members of the WNDCLASSEX structure. To see how this is done, let's begin with icons.

Beginning with version 4, a Windows NT application has two icons associated with it: one standard size and one small. The small icon is used when the application is minimized and it is also the icon that is used for the system menu. The standard size icon (also frequently called the large icon) is displayed when you move or copy an application to the desktop. Standard icons are 32 x 32 bitmaps and small icons are 16x16 bitmaps. The style of each icon is loaded by the API function LoadIcon( ), whose prototype is shown here: HICON LoadIcon (HINSTANCE hInst, LPCSTR lpszName);

This function returns a handle to an icon. Here, hlnst specifies the handle of the module that contains the icon. The icon's name is specified in lpszName. However, to use one of the built-in icons, you must use NULL for the first parameter and specify one of the following macros for the second.

Icon Macro	Shape
IDI_APPLICATION	Default icon
IDI_ASTERISK	Information icon
IDI_EXCLAMATION	Exclamation point icon
IDI_HAND	Stop sign
IDI_QUESTION	Question mark icon
IDI_WINLOGO	Windows NT Logo

In the skeleton, IDI_APPLICATION is used for the standard icon and IDI_WINLOGO is used for the small icon.

To load the mouse cursor, use the API LoadCursor() function. This function has the following prototype: HCURSOR LoadCursor(HINSTANCE hlnst, LPCSTR lpszName);

This function returns a handle to a cursor resource. Here, hlnst specifies the handle of the module that contains the mouse cursor, and the name of the mouse cursor is specified in

lpszName. However, to use one of the built in cursors, you must use NULL for the first parameter and specify one of the built-in cursors using its macro for the second parameter. Some of the most common built-in cursors are shown here

Cursor Macro	Shape
IDC_ARROW	Default arrow pointer
IDC_CROSS	Cross hairs
IDC_IBEAM	Vertical I-beam
IDC_WAIT	Hourglass

The background color of the window created by the skeleton is specified as white and a handle to this brush is obtained using the API function GetStockObject(). A brush is a resource that paints the screen using a predetermined size, color, and pattern. The function GetStockObject() is used to obtain a handle to a number of standard display objects, including brushes, pens (which draw lines), and character fonts. It has this prototype:

HGDIOBJ GetStockObject(int object);

The function returns a handle to the object specified by object. (The type HGDIOBJ is a GDI handle.) Here are some of the built-in brushes available to your program:

Macro Name	Background Type
BLACK_BRUSH	Black
DKGRAY_BRUSH	Dark gray
HOLLOW_BRUSH	See through window
LTGRAY_BRUSH	Light gray
WHITE_BRUSH	White

You may use these macros as parameters to GetStockObject( ) to obtain a brush.

Once the window class has been fully specified, it is registered with Windows NT using the API function RegisterCIassEx( ), whose prototype is shown here.

ATOM RegisterCIassEx(CONST WNDCLASSEX *lpWClass);

The function returns a value that identifies the window class. ATOM is a typcdef that means WORD. Each window class is given a unique value. lpWCIass must be the address of a WNDCLASSEX structure.

Creating a Window

Once a window class has been defined and registered, your application can actually create a window of that class using the API function CreateWindow( ), whose prototype is shown here.

```
HWND CreateWindow(
LPCSTR lpszClassName, /* name of window class */
LPCSTR lpszWinName, /* title of window */
DWORD dwStyle, /* type of window */
int X, inl Y, /* upper-left coordinates */
int Width, int Height, /* dimensions of window */
HWND hParent, /* handle of parent window */
HMENU hMenu, /* handle of main menu */
HINSTANCE hThisInst, /* handle of creator */
LPVOID lpszAdditional /* pointer to additional info */ );
```

As you can see by looking at the skeleton program, many of the parameters to CreateWindow( ) may be defaulted or specified as NULL. In fact, most often the X, Y, Width, and Height parameters will simply use the macro CW_USEDEFAULT, which tells Windows NT to select an appropriate size and location for the window. If the window has no parent, which is the case in the skeleton, then hParent can be specified as HWND_DESKTOP. (You may also use NULL for this parameter.) If the window does not contain a main menu or uses the main menu defined by the window class, then hMenu must be NULL. (The hMenu parameter has other uses, too.) Also, if no additional information is required, as is most often the case, then lpszAdditional is NULL. (The type LPVOID is typedefed as void *. Historically, LPVOID stands for long pointer to void.)

The remaining four parameters must be explicitly set by your program. First, lpszCIassName must point to the name of the window class. (This is the name you gave it when it was registered.) The title of the window is a string pointed to by lpszWinName. This can be a null string, but usually a window will be given a title. The style (or type) of window actually created is determined by the value of dwStyle. The macro WS_OVERLAPPED-WINDOW specifies a standard window that has a system menu a border, and minimize, maximize, and close boxes. While this style of window is the most common, you can construct one to your own specifications. To accomplish this, simply OR together the various style macros that you want. Some other common styles are shown here

Style Macro	Window Feature
WS_OVERLAPPED	Overlapped window with border
WS_MAXIMIZEBOX	Maximize box
WS_MINIMIZEBOX	Minimize box
WS_SYSMENU	System menu
WS_HSCROLL	Horizontal scroll bar
WS_VSCROLL	Vertical scroll bar

The hThisInst parameter must contain the current instance handle of the application.

The CreatcWindow() function returns the handle of the window it creates or NULL if the window cannot be created.

Once the window has been created, it is still not displayed on the screen. To cause the window to be displayed, call the ShowWindow( ) API function. This function has the following prototype: **BOOL ShowWindow(HWND hwnd, int nHow);**

The handle of the window to display is specified in hwnd. The display mode is specified in nHow. The first time the window is displayed, you will want to pass WinMain( )'s nWinMode as the nHow parameter. Remember, the value of nWinMode determines how the window will be displayed when the program begins execution. Subsequent calls can display (or remove) the window as necessary. Some common values for nHow are shown here:

Display Macros	Effect
SW_HIDE	Removes the window
SW_MINIMIZE	Minimizes the window into an icon
SW_MAXIMIZE	Maximizes the window
SW_RESTORE	Returns a Window to normal size

The ShowWindow( ) function returns the previous display status of the window.

If the window was displayed, then nonzero is returned. If the window was not displayed, zero is returned.

Although not technically necessary for the skeleton, a call to UpdateWindow( ) is included because it is needed by virtually every Windows NT application that you will create. It essentially tells Windows NT to send a message to your application that the main window needs to be updated.

The Message Loop

The final part of the skeletal WinMain( ) is the message loop. The message loop is a part of all Windows applications. Its purpose is to receive and process messages sent by Windows NT. When an application is running, it is continually being sent messages. These messages are stored in the application's message queue until they can be read and processed. Each time your application is ready to read another message, it must call the API function GetMessage(), which has this prototype:

BOOL GetMessage(LPMSG msg, HWND hwnd, UINT min, UINT max);

The message will be received by the structure pointed to by msg. All Windows messages are of structure type MSG, shown here.

/* Message structure */

```
typedef struct tagMSG {
HWND hwnd; /* window that message is for */
UINT message; /* message */
WPARAM wParam; /* message-dependent info */
LPARAM lParam; /* more message-dependent info */
DWORD time; /* time message posted */
POINT pt; /* X,Y location of mouse */ } MSG;
```

In MSG, the handle of the window for which the message is intended is contained in hwnd. The message itself is contained in message. Additional information relating to each message is passed in wParam and lParani. The type WPARAM is a typedef for UINT and LPARAM is a typedef for LONG. The time the message was sent (posted) is specified in milliseconds in the time field.

The pt member will contain the coordinates of the mouse when the message was sent. The coordinates are held in a POINT structure which is defined like this:

```
typedef struct tagPOINT {
LONG x, y;
} POINT;
```

If there are no messages in the application's message queue, then a call to GetMessage( ) will pass control back lo Windows NT.

The hwnd parameter to GetMessage() specifies for which window messages will be obtained. It is possible (even likely) that an application will contain several windows and you may only want to receive messages for a specific window. If you want to receive nil messages directed at your application, this parameter must be NULL.

The remaining two parameters to GetMessage() specify a range of messages that will be received, Generally, you want your application to receive all messages. To accomplish this, .specify both min and max as 0, as the skeleton does.

GetMessage( ) returns zero when the user terminates the program, causing the message loop to terminate. Otherwise It returns nonzero.

Inside the message loop, two functions are called. The first is the API function TranslateMessage( ). The function translates virtual key codes generated by Windows NT into character messages. (Virtual keys are discussed later in this lecture.) Although It is not necessary for all applications, most call TranslateMessage() because It is needed to allow full integration of the keyboard into your application program.

Once the message has been read and translated, it is dispatched back to Windows NT using the DispatchMessage() API function. Windows NT then holds this message until it can pass it to the program's window function.

Once the message loop terminates, the WinMain() function ends by returning the value of msg.wParam to Windows NT. This value contains the return code generated when your program terminates.

The Window Function

The second function in the application skeleton is its window function. In this case the function is called WindowFunc(), but it could have any name you like. The window function is passed messages by Windows NT. The first four members of the MSG structure are its parameters. For the skeleton, the only parameter that is used is the message itself. However, in the next lecture you will learn more about the parameters to this function.

The skeleton's window function responds to only one message explicitly: WM_DESTROY. This message is sent when the user terminates the program. When this message is received, your program must execute a call to the API function PostQuitMessagc( ). The argument to this function is an exit code that is returned in msg.wParam inside WinMain(). Calling PostQuitMessage() causes a WM_QUIT message to be sent to your application, which causes GetMessage() to return false, thus stopping your program.

Any other messages received by WindowFunc() are passed along to Windows NT, via a call to DefWindowProc( ), for default processing. This step is necessary because all messages must be dealt with in one fashion or another.

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Two:
Application Essentials
(Message Box)

CONTENTS:

Although the skeleton developed in lecture 2 forms the framework for a Windows NT program, by itself it is useless. To be useful, a program must be capable of performing two fundamental operations. First, it must be able to respond to various messages sent by Windows NT. The processing of these messages is at the core of all Windows NT applications. Second, your program must provide some means of outputting information to the user. (That is, it must be able to display information on the screen.) Unlike programs that you may have written for other operating systems, outputting information to the user is a non-trivial task in Windows. In fact, managing output forms a large part of any Windows application. Without the ability to process messages and display information, no useful Windows program can be written. For this reason, message processing and the basic I/O operations are the subject of this lecture.

Message Boxes

The easiest way to output information to the screen is to use a message box. As you will see, many of the examples in this lecture make use of message boxes. A message box is a simple window that displays a message to the user and waits for an acknowledgment. Unlike other types of windows that you must create, a message box is a system-defined window that you may use. In general, the purpose of a message box is to inform the user that some event has taken place. However, it is possible to construct a message box that allows the user to select from among a few basic alternatives as a response to the message. For example, one common form of message box allows a user to select Abort, Retry, or Ignore.

NOTE: In the term message box, the word message refers to human-readable text that is displayed on the screen. It does not refer to Windows NT messages which are sent to your program's window function. Although the terms sound similar, message boxes and messages are two entirely separate concepts.

To create a message box, use the MessagcBox() API function. Its prototype is shown here:

int MessageBox(HWND hwnd, LPCSTR lpText, LPCSTR lpCaption, UINT MBType);

Here, hwnd is the handle to the parent window. The lpText parameter is a pointer to a string that will appear inside the message box. The string pointed to by lpCaption is used as the title for the box. The value of MBTypc determines the exact nature of the message box, including what type of buttons and icons will be present. Some of the most common values are shown in Table below.

VALUE	EFFECT
MB_ABORTRETRYIGNORE	Displays Abort, Retry, and ignore push bottom.
MB_ICONEXCLAMATION	Displays Exclamation-point icon.
MB_ICONHAND	Displays a stop sign icon.
MB_ICONINFORMATION	Displays an information icon.
MB_ICONQUESTION	Displays a question mark icon.
MB_ICONSTOP	Displays as MB_ICONHAND.

MB_OK	**Displays OK button.**
MB_OKCANCEL	**Displays OK and Cancel push buttons.**
MB_RETRYCANCEL	**Displays Retry and Cancel push buttons.**
MB_YESNO	**Displays Yes and No push buttons.**
MB_YESNOCANCEL	**Displays Yes, No, and Cancel push buttons.**

These macros are defined by including WINDOWS.H. You can OR together two or more of these macros so long as they are not mutually exclusive. MessageBox() returns the user's response to the box. The possible return values are shown here:

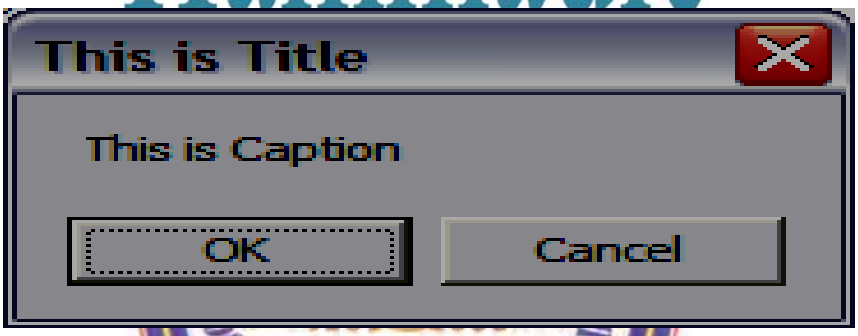
Button Pressed	Return Value
Abort	IDABORT
Retry	IDRETRY
Ignore	IDIGNORE
Cancel	IDCANCEL
No	IDNO
Yes	IDYES
Ok	IDOK

Remember, depending upon the value of MBType, only certain buttons will be present. Quite often message boxes are simply used to display an item of information and the only response offered to the user is the OK button. In these cases, the return value of a message box is simply ignored by the program.

To display a message box, simply call the **MessageBox()** function. Windows NT will display it at its first opportunity. MessageBox() automatically creates a window and displays your message in it. For example, this call to **MessageBox( )**

i=MessageBox(hwnd,"This is Caption", "This is Title", MB_OKCANCEL);

produce the following message box.



Depending on which button the user presses, i will contain either **IDOK** or **IDCANCEL**.

Message boxes are typically used to notify the user that some event has occurred. However, because message boxes are so easy to use, they make excellent debugging tools when you need a simple way to output something to the screen. As you will see, examples in this book will use a message box whenever a simple means of displaying information is needed.

Now that we have a means of outputting information, we can move on to processing messages.

Understanding Windows NT Messages

As it relates to Windows NT, a message is a unique 32-bit integer value. Windows NT communicates with your program by sending it messages. Each message corresponds to some event. For example, there are messages to indicate that the user has pressed a key, that the mouse has moved, or that a window has been resized.

Although you could, in theory, refer to each message by its numeric value, in practice this is seldom done. Instead, there are macro names defined for all Windows NT messages. Typically, you will use the macro name, not the actual integer value, when referring to a message. The standard names for the messages are defined by including WINDOWS.H in your program. Here are some common Windows NT message macros:

WM_MOVE	WM_PAINT	WM_CHAR
WM_LBUTTONDOWN	WM_LBUTTONUP	WM_CLOSE
WM_SIZE	WM_HSCROLL	WM_COMMAND

Two other values accompany each message and contain information related to it. One of these values is of type **WPARAM**, the other is of type **LPARAM**. For Windows NT, both of these types translate into 32-bit integers. These values are commonly called **wParam** and **lParam**, respectively. The contents of **wParam** and **lParam** are determined by which message is received. They typically hold things like mouse coordinates; the value of a key press; or a system-related value, such as window size. As each message is discussed, the meaning of the values contained in **wParam,** and **lParam** will be described.

As mentioned in lecture 1, the function that actually processes messages is your program's window function. As you should recall, this function is passed four parameters: the handle of the window that the message is for, the message itself, **wParam**, and **lParam**.

Sometimes two pieces of information are encoded into the two words that comprise the **wParam** or **lParam** parameters. To provide easy access to each value, Windows defines two macros called **LOWORD** and **HIWORD.**

They return the low-order and high-order words of a long integer, respectively. They are used like this: *x = LOWORD (lParam) ; x = HIWORD (lParam) ;*

You will see these macros in use soon.

Windows NT defines a large number of messages. Although it is not possible to examine every message, this lecture discusses some of the most common ones.

Responding to a Keypress

One of the most common Windows NT messages is generated when a key is pressed. This message is called **WM_CHAR**. It is important to understand that your application never receives, per se, keystrokes directly from the keyboard: Instead, each time a key is pressed, a **WM_CHAR** message is sent to the active window (i.e., the one that currently has input focus). To see how this process works, this section extends the skeletal application developed in lecture 2 so that it processes keystroke messages. Each time **WM_CHAR** is sent, **wParam** contains the ASCII value -of the key pressed. **LOWORD(lParam)** contains the number of times the key has been repeated as a result of the key being held down. The bits of **HIWORD(lParam)** are encoded as shown in Table below.

Bit	Meaning
15	Set if the key is being released; cleared if the key is being pressed.
14	Set if the key was pressed before the message was sent; cleared if it was not pressed.
13	Set if the ALT key is also being pressed; cleared if ALT key is not pressed.
12	Used by Window NT.
11	Used by Window NT.
10	Used by Window NT.
9	Used by Window NT.
8	Set if the key pressed is an extended key provided by enhanced keyboard; cleared otherwise.
7 - 0	Manufacturer-dependent key code (i.e., the scan code)./

For our purposes, the only value that is important at this time is wParam, since it holds the key that was pressed. However, notice how detailed the information is that Windows NT supplies about the state of the system. Of course, you are free to use as much or as little of this information as you like. To process a **WM_CHAR** message, you must add it to the **switch** statement inside your program's window function. For example, here is a program that processes a keystroke by displaying the character on the screen using a message box.

```
/*Processing WM_CHAR messages.*/
#include   <windows.h>
#nclude   <string.h>
#include   <stdio.h>
LRESULT    CALLBACK    WindowFunc(HWND,      UINT,      WPARAM,
LPARAM);
char  szWinNarrte [ ]="MyWin";/*name  of  window class*/
char  str[255]   =   "";   /*  holds   output   string   */
int  WINAPI  WinMain(HINSTANCE  hThisInst,   HINSTANCE  hPrevInst,
LPSTR    lpszArgs,     int    nWinMode)   { HWND   hwnd; MSG msg;
WNDCLASSEX wcl;
/* Define a window class. */
```

```
wcl.cbSize = sizeof(WNDCLASSEX);
wcl.hInstance = hThisInst; /* handle to this instance */
wcl. lpszClassName= szWinName; /* window class name */
wcl.lpfnWndProc = WindowFunc; /* window function */
wcl.style = 0; /* default style */
wcl.hIcon = LoadIcon(NULL,IDI_APPLICATION);/*standard icon*/
wcl.hIconSm = LoadIcon(NULL,IDI_APPLICATION);/*small icon*/
wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /*cursor style*/
wcl.lpszMenuName = NULL; /* no main menu */
wcl.cbClsExtra =0; /* no extra */
wcl.cbWndExtra =0; /* information needed */
/* Make the window white. */
wcl.hbrBackground = GetStockObject(WHITE_BRUSH);
/* Register the window class. */
if ( !RegisterClassEx(&wcl) )return 0;
/*Now that a window class has been registered,a window can be
created.*/
hwnd = CreateWindow(
szWinName, /* name of window class */
"Processing WM_CHAR Messages", /* title */
WS_OVERLAPPEDWINDOW, /* window style - normal */
CW_USEDEFAULT, /* X coordinate - let Windows decide */
CW_USEDEFAULT, /* Y coordinate - let Windows decide */
CW_USEDEFAULT, /* width - let Windows decide */
CW_USEDEFAULT, /* height - let Windows decide */
HWND_DESKTOP, /* no parent window */
NULL,
hThisInst, /* handle of this instance of the program */
NULL /* no additional arguments */);
/*Display the window.*/ ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);
/* Create the message loop. */
while(GetMessage(&msg, NULL, 0, 0))
{ TranslateMessage(&msg);/*allow use of keyboard */
 DispatchMessage (&msg); /*return control to Windows NT */
}return msg. wParam;}
/* This function is called by Windows NT and is passed messages
from the message queue. */
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
```

```
                WPARAM wParam, LPARAM lParam)
{switch(message){
case WM_CHAR: /* process keystroke */
sprintf(str, "Character is %c", (char) wParam);
MessageBox(hwnd, str, "WM_CHAR Received",MB_OK); break;
case WM_DESTROY: /* terminate the program */
 PostQuitMessage(0); break;
default:/*Let Windows NT process any messages not specified in the
preceding switch statement.*/
 return DefWindowProc(hwnd, message, wParam, lParam);}
return 0;}
```

Sample output produced by this program is shown in Figure 2-1. In the program, look carefully at these lines of code from **WindowFunc( )**:

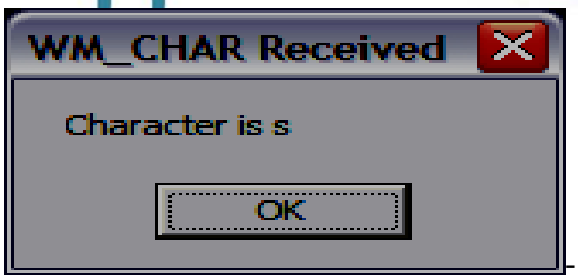


Figure 2-1: Output Program

```
case WM_CHAR: /* process keystroke */
sprintf(str, "Character is %c", (char) wParam);
MessageBox(hwnd, str, "WM_CHAR Received", MB_OK); break;
```

As you can see, the **WM_CHAR** message has been added to the **case** statement. When you run the program, each time you press a key, a **WM_CHAR** message is generated and sent to **WindowFunc()**. Inside the WM_CHAR case, the character received in **wParam** is converted into a string using **sprintf()** and then displayed using a message box.

A Closer Look at Keyboard Messages

While **WM_CHAR** is probably the most commonly handled keyboard message, it is not the only one. In fact, **WM_CHAR** is actually a synthetic message that is constructed by the **TranslateMessage( )** function inside your program's message loop. At the lowest level, Windows NT generates two messages each time you press a key. When a key is pressed, a **WM_KEYDOWN** message is sent. When the key is released, a **WM_KEYUP** message is posted. If possible, a **WM_KEYDOWN** message is translated into a **WM_CHAR** message by **TranslateMessage( )**. Thus, unless you include **TranslateMessage()** in your message loop, your program will not receive **WM_CHAR** messages. To prove this to yourself, try commenting out the call to

TranslateMessage() in the preceding program. After doing so, it will no longer respond to your keypresses.

The reason you will seldom use **WM_KEYDOWN** or **WM_KEYUP** for character input is that the information they contain is in a raw format. For example, the value in **wParam** contains the virtual key code, not the key's ASCII value. Part of what **TranslateMessaage( )** does is transform the virtual key codes into ASCII characters, taking into account the state of the shift key, etc. Also, **TranslateMessage( )** also automatically handles auto-repeat.

A virtual key is a device-independent key code. As you may know, there are keys on nearly all computer keyboards that do not correspond to the ASCII character set. The arrow keys and the function keys are examples. Each key that can be generated has been assigned a value, which is its virtual key code. All of the virtual key codes are defined as macros in the header file WINUSER.H,(which is automatically included in your program when you include WINDOWS.H). The codes begin with VK_. Here are some examples.

Virtual Key Code	Corresponding Key
VK_DOWN	Down Arrow
VK_LEFT	Left Arrow
VK_RIGHT	Right Arrow
VK_UP	Up Arrow
VK_SHIFT	Shift
VK_CONTROL	Control
VK_ESCAPE	ESC
VK_F1 through VK_F24	Function Keys
VK_HOME	HOME
VK_END	END
VK_INSERT	INSERT
VK_DELETE	DELETE
VK_PRIOR	PAGE UP
VK_NEXT	PAGE DN
VK_A through VK_Z	The letters of the alphabet
VK_0 through VK_9	The digit 0 through 9

Of course, the non-ASCII keys are not converted. This means that if your program wants to handle non-ASCII keypresses it must use **WM_KEYDOWN** or **WM_KEYUP** (or both). Here is an enhanced version of the preceding program that handles both **WM_KEYDOWN** and **WM_CHAR** messages. The handler for **WM_KEYDOWN** reports if the key is an arrow, shift, or control key. Sample output is shown in Figure 2-2.

```
/* Processing WM_KEYDOWN and WM_CHAR messages. */
#include <windows.h>
#include <string.h>
```

```
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
char str[255] = ""; /* holds output string */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
LPSTR lpszArgs, int nWinMode){HWND hwnd; MSG msg; WNDCLASSEX wcl;
/* Define a window class. */
wcl.cbSize = sizeof(WNDCLASSEX);wcl.hInstance = hThisInst;
wcl.lpszClassName = szWinName;wcl.lpfnWndProc = WindowFunc;
wcl.style = 0; wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wcl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor(NULL,IDC_ARROW); wcl.lpszMenuName = NULL;
wcl.cbClsExtra =0; wcl.cbWndExtra =0;
wcl.hbrBackground = GetStockObject(WHITE_BRUSH);
/*Register the window class.*/if (!RegisterClassEx(&wcl))return 0;
/*Now that a window class has been registered, a window can be
created. */
hwnd = CreateWindow(szWinName, "Processing WM_CHAR and WM_KEYDOWN
Messages", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,HWND_DESKTOP, hThisInst, NULL);
ShowWindow(hwnd,nWinMode); UpdateWindow(hwnd);
/* Create the message loop. */
while(GetMessage(&msg, NULL, 0, 0))
{TranslateMessage(&msg);DispatchMessage(&msg);}return msg.wParam;}
/* This function is called by Windows NT and is passed messages
from the message queue*/
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
 {switch (message) {
case WM_CHAR: /* process character */
sprintf(str, "Character is %c", (char) wParam);
MessageBox (hwnd, str, "WM_CHAR Received", MB_OK) ; break;
case WM_KEYDOWN: /* process raw keystroke */
        switch ( (char) wParam)
 { case VK_UP: strcpy(str, "Up Arrow"); break;
   case VK_DOWN: strcpy(str, "Down Arrow"); break;
   case VK_LEFT: strcpy(str, "Left Arrow"); break;
   case VK_RIGHT: strcpy(str, "Right Arrow");break ;
   case VK_SHIFT: strcpy(str, "Shift"); break;
```

```
    case VK_CONTROL: strcpy(str, "Control"); break;
    default:  strcpy(str, "Other Key"); }
  MessageBox(hwnd, str, "WM_KEYDOWN Received", MB_OK); break;
Case WM_DESTROY: PostQuitMessage(0);break;
Default   return   DefWindowProc(hwnd,   message,   wParam,   lParam);
}return 0; }
```

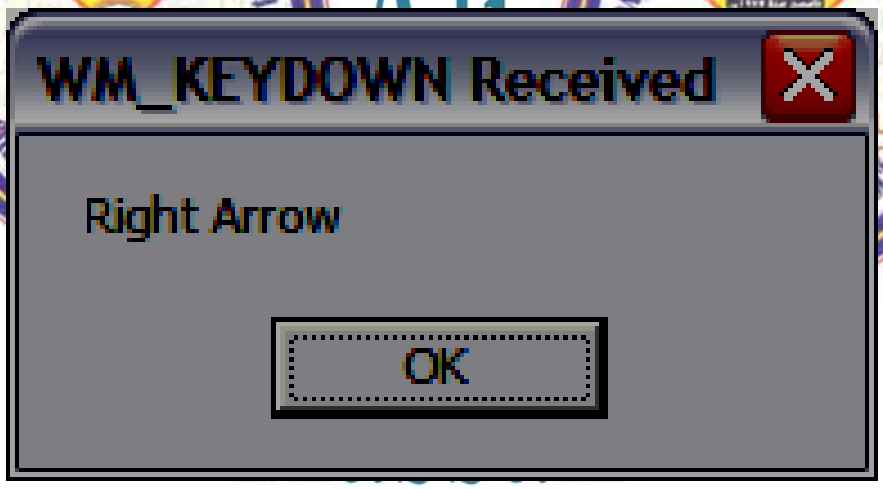


Figure 3-2: Sample output from WM_KEYDOWN program

When you try this program notice one important point: when you press a standard ASCII key, such as X, the program will receive two messages: one will be **WM_CHAR** and the other will be **WM_KEYDOWN**. The reason for this is easy to understand. Each time you press a key, a **WM_KEYDOWN** message is generated. (That is, all keystrokes generate a key down message.) If the key is an ASCII key, it is transformed into a **WM_CHAR** message by **TranslateMessage( ).**

H.W:/

- ***Check what are type of keys in keyboard (Virtual key or not) how can recognize***
- ***How can obtain code of virtual key or not virtual key***
- ***Write sub routine to work as "text editor"***
- ***Can be different between virtual key or other***

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Three:

Application Essentials

(Outputting Text to a Window)

CONTENTS:

Outputting Text to a Window

Although the message box is the easiest means of displaying information, it is obviously not suitable for all situations. There is another way for your program to output information: it can write directly to the client area of its window. Windows NT supports both text and graphics output. In this section, you will learn the hash s of text output. Graphics output is reserved for later in this lecture.

The first thing to understand about outputting text to a window is that you cannot use the standard C or C++ I/O system. The reason for this is that the standard C/C++ I/O functions and operators direct their output to standard output. However, In a Windows program, output is directed to a window. To see how text can be written to a window, let's begin with an example that output each character that you type to the program's window, instead of using a message box. Here is a version of WindowFunc() that does this,

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM
wParam, LPARAM lParam){HDC hdc;static unsigned j=0;
switch(message) {
case WM_CHAR: /* process keystroke */
  hdc=GetDC(hwnd); /* get device context */
  sprintf(str, "%c", (char) wParam); /* stringize character*/
  TextOut(hdc, j*10, 0, str, strlen(str)); /* output char */
  j++ /* try commenting-out this line */
  ReleaseDC(hwnd, hdc); /* release device context */ break;
case WM_DESTROY: /* terminate the program */
 PostQuitMessage(0); break;
default:/*Let Windows NT process any messages not specified in the
preceding switch statement.*/
return DefWindowProc(hwnd,message,wParam,lParam);} return 0;}
```

Look carefully at the code inside the **WM_CHAR** case. It simply echoes each character that you type to the program's window. Compared to using the standard C/C++ I/O function or operators, this code probably seems overly complex. The reason for this Windows must establish a link between your program and the screen. This link is called a device context **(DC)** and it is acquired by calling **GetDC( )**. For now, don't worry about the precise definition of a device context. It will be discussed in the next section. However, once you obtain a device context, you may write to the window. At the end of the process, the device

context is released using **ReleaseDC( ).** Your program must release the device context when it is done with it. Although the number of device contexts is limited only by the size of free memory, the number is still finite. If your program doesn't release the DC, eventually, the available DCs will be exhausted and a subsequent call to **GetDC()** will fail. Both **GetDC( )** and **ReleaseDC( )** are API functions. Their prototypes are shown here:

HDC GetDC(HWND hwnd);

int ReleaseDC(HWND hwnd, HDC hdc);

GetDC( ) returns a device context associated with the window whose handle is specified by hwnd. The type HDC specifies a handle to a device context. If a device context cannot be obtained, the function returns NULL.

ReleaseDC( ) returns true if the device context was released, false otherwise. The hwnd parameter is the handle of the window for which the device context is released. The hdc parameter is the handle of device context obtained through the call to **GetDC( ).**

The function that actually outputs the character is the API function **TextOut()**. Its prototype is shown here: **BOOL TextOut(HDC DC, int x, int y, LPCSTR lpstr, int nlength);**

The **TextOut( )** function outputs the string pointed to by lpstr at the window coordinates specified by x, y. (By default, these coordinates are in terms of pixels.) The length of the string is specified in nlength. The **TextOut()** function returns nonzero if successful, zero otherwise.

In the **WindowFunc( )**, each time a WM_CHAR message is received, the character that is typed by the user is converted, using **sprintf(),** into a string that is one character long, and then displayed in the window using **TextOut( ).** The first character is displayed at location 0, 0. Remember, in a window the upper left corner of the client area is location 0, 0. Window coordinates are always relative to the window, not the screen. Therefore, the first character is displayed in the upper left corner no matter where the window is physically located on the screen. The reason for the variable j is to allow each character to be displayed to the right of the preceding character. That is, the second character is displayed at 10, 0, the third at 20, 0, and so on. Windows does not support any concept of a text cursor which is automatically advanced. Instead, you must explicitly specify where each **TextOut( )** string will be written. Also, **TextOut( )** does not advance to the next line when a newline character is encountered, nor does it expand tabs. You must perform all these activities yourself.

Before moving on, you might want to try one simple experiment: comment out the line of code that increments j. This will cause all characters to be displayed at location 0, 0. Next, run the program and try typing several characters. Specifically, try typing a **W** followed by an i. Because Windows is a graphics-based system, characters are of different sizes and the overwriting of one character by another does not necessarily cause all of the previous character to be erased. For example, when you type a W followed by an i, part of the W will still be displayed. The fact that characters are proportional also explains why the spacing between characters that you type is not even.

Understand that the method used in this program to output text to a window is quite crude. In fact, no real Windows NT application would use this approach. Later in this lecture, you will learn how to manage text output in a more sophisticated fashion.

No Windows NT API function will allow output beyond the borders of a window. Output will automatically be clipped to prevent the boundaries from being crossed. To confirm this for yourself, try typing characters past the border of the window. As you will see, once the right edge of the window has been reached, no further characters are displayed.

At first you might think that using **TextOut( )** to output a single character is an inefficient application of the function. The fact is that Windows NT (and Windows in general) does not contain a function that simply outputs a character. Instead, Windows NT performs much of its user interaction through dialog boxes, menus, toolbars, etc. For this reason it contains only a few functions that output text to the client area. Further, you will generally construct output in advance and then use **TextOut( )** to simply move that output to the screen.

Here is the entire program that echoes keystrokes to the window. Figure 4-1 shows sample output.

```
/* Displaying text using TextOut ( ). */
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
char str[255] = ""; /* holds output string */
int WIMAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
LPSTR IpszArgs, int nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1;
```

```
/* Define a window class. */
 wc1.cbSize = sizeof (WNDCLASSEX) ;
wc1.hInstance = hThisInst; /* handle to this instance */
wc1.lpszClassName = szWinName; /* window class name */
wc1.lpfnWndProc = WindowFunc; /* window function */
wc1.style = 0; /* default style */
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* standard icon */
wc1.hIconSm = LoadIcon(NULL, IDI_APPLICATION); /* small icon */
 wc1.hCursor = LoadCursor(NULL, IDC_ARROW); /* cursor style */
wc1.lpszMenuName = NULL; /* no main menu */
wc1.cbClsExtra =0; /* no extra */
wc1 .cbWndExtra = 0; /* information needed */
/* Make the window white. */wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
/* Register the window class. */ if(!RegisterClassEx(&wcl))  return 0;
/* Now that a window class has been registered, a window can be created. */
hwnd = CreateWindow(
szWinName, /* name of window class */
"Display WM_CHAR Messages Using TextOut", /* title */
WS_OVERLAPPEDWINDOW, /* window style - normal */
CW_USEDEFAULT, /* X coordinate - let Windows decide */
CW_USEDEFAULT, /* Y coordinate - let Windows decide */
CW_USEDEFAULT, /* width - let Windows decide */
CW_USEDEFAULT, /* height - let Windows decide */
HWND_ DESKTOP,    /*  no parent window   */
NULL,
hThisInst,    /*  handle of the instance of the  program   */
NULL   /*  no  additional arguments  */ );
/* Display the window*/ ShowWindow (hwnd, nWinMode) ; UpdateWindow( hwnd) ;
while(GetMessage(&msg, NULL, 0, 0))
        { TranslateMessage(&msg);/*allow use of keyboard */
          DispatchMessage (&msg); /*return control to Windows NT */ }
return msg. wParam;}
```

```
LRESULT CALLBACK WindowFunc(HWND hwnd, U1NT message,
        WPARAM wParam, LPARAM lParam){ HDC hdc; static unsigned j=0;
switch (message) { case WM_CHAR: /* process keystroke */
                hdc=GetDC(hwnd); /* get device context */
                sprintf(str,"%c", (char) wParam); /* stringize character*/
                TextOut(hdc, j*10, 0, str, strlen(str)); /* output char */
                j++; /* try commenting-out this line */
                ReleaseDC(hwnd, hdc); /* release device context */ break;
        case WM_DESTROY: /* terminate the program */
                PostQuitMessage(0); break;
default: /* Let Windows NT process any messages not specified in the preceding switch
statement.*/return DefWindowProc(hwnd, message, wParam, lParam);} return 0;}
```

Figure 3-1:Sample window produced using TextOut

Device Contexts

The program in the previous section had to obtain a device context prior to outputting to the window. Also, that device context had to be released prior to the termination of that function. It is now time to understand what a device context is. A device context is a structure that describes the display environment of a window, including its device driver and various display parameters, such as the current type font. As you will see later in this lectures, you have substantial control over the display environment of a window.

Before your application can output information to the client area of the window a device context must be obtained. Until this is done, there is no linkage between your program and the window relative to output. Since **TextOut( )** and other output functions require a handle to a device context, this is a self-enforcing rule.

Processing the WM_PAINT Message

One of the most important messages that your program will receive is **WM_PAINT**. This message is sent when your program needs to restore the contents of its window. To understand why this important, run the program from the previous section and enter a few characters. Next, minimize and then restore the window. As you will see, the characters that you typed are not displayed after the window is restored. Also, if the window is overwritten by another window and then redisplayed, the characters are not redisplayed. The reason for this is simple: in general, Windows does not keep a record of what a window contains. Instead, it is your program's job to maintain the contents of a window. To help your program accomplish this, each time the contents of a window must be redisplayed, your program will be sent a **WM_PAINT** message. (This message will also be sent when your window is first displayed.) Each time your program receives this message it must redisplay the contents of the window.

Before explaining how to respond to a **WM_PAINT** message it might be useful to explain why Windows does not automatically rewrite your window. The answer is short and to the point:. In many situations, it is easier for your program, which has intimate knowledge of the contents of the window, to rewrite it than it would be for Windows to do so. While the merits, of this approach have been much debated by programmers, you should simply accept it, because it is unlikely to change.

The first step to processing a **WM_PAINT** message is to add it to the **switch** statement inside the window function. For example, here is one way to add a **WM_PAINT** case to the previous program

```
case WM_PAINT:  /* process a repaint request  */
hdc = BeginPaint(hwnd, &paintstruct); /* get DC  */
TextOut(hdc, 0, 0, str, strlen(str));
EndPaint(hwnd, &paintstruct); /* release DC  */ break;
```

Let's look at this closely. First, notice that a device context is obtained using a call to **BeginPaint( )** instead of **GetDC( )**. For various reasons, when you process a **WM_PAINT** message, you must obtain a device context using **BeginPaint( )**, which has this prototype:

HDC BeginPaint(HWND hwnd, PAINTSTRUCT *lpPS);

BeginPaint( ) returns a device context if successful or **NULL** on failure. Here, hwnd is the handle of the window for which the device context is being obtained. The second parameter is a pointer to a structure of type **PAINTSTRUCT**. On return, the structure pointed to by lpPS will contain information that your program can use to repaint the window. **PAINTSTRUCT** is defined like this:

```
typedef struct tagPAINTSTRUCT {
HDC hdc; /* handle to device context */
BOOL fErase; /* true if background must be erased */
RECT rcPaint; /* coordinates of region to redraw */
BOOL fRestore;  /* reserved */
BOOL fIncUpdate; /* reserved */
BYTE rgbReserved[32]; /* reserved */
} PAINTSTRUCT;
```

Here, **hdc** will contain the device context of the window that needs to be repainted. This DC is also returned by the call to **BeginPaint( ). fErase** will be nonzero if the background of the window needs to be erased. However, as long as you specified a background brush when you created the window, you can ignore the **fErase** member. Windows NT will erase the window for you.

The type **RECT** is a structure that specifies the upper left and lower right coordinates of a rectangular region. This structure is shown here:

```
typedef tagRECT {
LONG left, top; /* upper left */
LONG right, bottom; /* lower right */
} RECT;
```

In **PAINTSTRUCT**, the **rcPaint** element contains the coordinates of the region of the window that needs to be repainted. For now, you will not need to use the contents of rcPaint because you can assume that the entire window must be repainted. However, real programs that you write will probably need to utilize this information.

Once the device context has been obtained, output can be written to the window. After the window has been repainted, you must release the device context using a call to **EndPaint( )**, which has this prototype: **BOOL EndPaint(HWND hwnd, CONST PAINTSTRUCT *lpPS);**

EndPaint( ) returns nonzero. (It cannot fail.) Here, hwnd is the handle of the window that was repainted. The second parameter is a pointer to the **PAINTSTRUCT** structure used in the call to **BeginPaint( ).**

It is critical to understand that a device context obtained using **BeginPaint( )** must be released only through a call to **EndPaint().** Further, **BeginPaint( )** must only be used when a **WM_PAINT** message is being processed. Here is the full program that now processes **WM_PAINT** messages.

```
/* Process WM_PAINT Messages */
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
char str[255] = "Sample Output"; /* holds output string */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
int WinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1;
wc1 .cbSize = sizeof(WNDCLASSEX) ;
wc1.hlnstance = hThisInst; /* handle to this instance */
wc1 . lpszClassName = szWinName; /* window class name */
wc1.lpfnWndProc = WindowFunc; /* window function */
wc1.style = 0; /* default style */
wc1.hIcon= LoadIcon(NULL, IDI_APPLICATION) ; /* standard Icon */
 wc1.hIconSm = LoadIcon(NULL, IDI_APPLICATION); /* small icon */
wc1.hCursor = LoadCursor(NULL, IDC_ARROW); /* cursor style */
wc1.lpszMenuName = NULL; /* no main menu */
wc1.cbClsExtra =0; /* no extra */
wc1.cbWndExtra =0; /* information needed */
 wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
 if ( '. RegisterClassEx (&wc1) ) return 0 ;
hwnd = CreateWindow (szWinName, "Process WM_PAINT Messages",
      WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, NULL, hThisInst, NULL);
```

```
 ShowWindow(hwnd, nWinMode) ;  UpdateWindow(hwnd) ;
while (GetMessage(&msg, NULL, 0, 0))
 {TranslateMessage(&msg); /*allow use of keyboard */
   DispatchMessaget&msg); /* return control to Windows NT */}return msg. wParam;}
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam,
LPARAM lParam) {HDC hdc; static unsigned j = 0; PAINTSTRUCT paintstruct;
switch(message) { case WM_CHAR: /* process keystroke */
                  hdc = GetDC(hwnd); /* get device context */
                  sprintf(str, "%c", (char) wParam); /* stringize character */
                  TextOut(hdc, j*10,0, str, strlen(str)); /* output char */
                  j++; /* try commenting-out this line */
                  ReleaseDC(hwnd, hdc); /* release device context */ break;
               case WM_PAINT: /* process a repaint request */
                  hdc = BeginPaint(hwnd, &paintstruct); /* get DC */
                  TextOut(hdc, 0, 0, str, strlen(str));
                  EndPaint(hwnd, &paintstruct) ,- /* release DC */ break;
              case WM_DESTROY: /* terminate the program */
                  PostQuitMessage(O);break;
        default: /* Let Windows NT process any messages not specified in the preceding
switch statement. */return DefWindowProc(hwnd, message, wParam, lParam);} return 0;}
```

Before continuing, enter, compile, and run this program. Try typing a few characters and then minimizing and restoring the window. As you will see, each time the window is redisplayed, the last character you typed is automatically redrawn. The reason that only the last character is redisplayed is because **str** only contains the last character that you typed. You might find it fun to alter the program so that it adds each character to a string and then redisplays that string each time a **WM_PAINT** message is received. (You will see one way to do this in the next example.) Notice that the global array **str** is initialized to Sample Output and that this is displayed when the program begins execution. The reason for this is that when a window is created, a **WM_PAINT** message is automatically generated.

While the handling of the **WM_PAINT** message in this program is quite simple, it must be emphasized that most real-world applications will be more complex because most windows

contain considerably more output. Since it is your program's responsibility to restore the window if it is resized or overwritten, you must always provide some mechanism to accomplish this. In real-world programs, this is usually accomplished one of three ways. First, your program can regenerate the output by computational means. This is most feasible when no user input is used. Second, in some instances, you can keep a record of events and replay the events when the window needs to be redrawn. Finally, your program can maintain a virtual window that you copy to the window each time it must be redrawn. This is the most general method. (The implementation of this approach is described later in this lecture.) Which approach is best depends completely upon the application. Most of the examples in this book won't bother to redraw the window because doing so typically involves substantial additional code which often just muddies the point of an example. However, your programs will need to restore their windows in order to be conforming Windows NT applications.

Generating a WM_PAINT Message

It is possible for your program to cause a **WM_PAINT** message to be generated. At first, you might wonder why your program would need to generate a **WM_PAINT** message since it seems that it can repaint its window whenever it wants. However, this is a false assumption. Remember, updating a window is a costly process in terms of time. Because Windows is a multitasking system that might be running other programs that are also demanding CPU time, your program should simply tell Windows that it wants to output information, but let Windows decide when it is best to actually perform that output. This allows Windows to better manage the system and efficiently allocate CPU time to all the tasks in the system. Using this approach, your program holds all output until a **WM_PAINT** message is received.

In the previous example, the **WM_PAINT** message was received only when the window was resized or uncovered. However, if all output is held until a **WM_PAINT** message is received, then to achieve interactive I/O, there must be some way to tell Windows that it needs to send a **WM_PAINT** message to your window whenever output is pending. As expected, Windows NT includes such a feature. Thus, when your program has information to output, it simply requests that a **WM_PA1NT** message be sent when Windows is ready

to do so. To cause Windows to send a **WM_PAINT** message, your program will call the **InvalidateRect( )** API function. Its prototype is shown here:

BOOL InvalidateRect(HWND hwnd, CONST RECT *lpRect, BOOL bErase);

Here, hwnd is the handle of the window to which you want to send the **WM_PAINT** message. The **RECT** structure pointed to by lpRect specifies the coordinates within the window that must be redrawn. If this value is NULL then the entire window will be specified. If bErase is true, then the background will be erased. If it is zero, then the background is left unchanged. The function returns nonzero if successful; it returns zero otherwise. (In general, this function will always succeed.)

When **InvalidateRect( )** is called, it tells Windows that the window is invalid and must be redrawn. This, in turn, causes Windows to send a **WM_PAINT** message to the program's window function.

Here is a reworked version of the previous program that routes all output through the **WM_PAINT** message. The code that responds to a **WM_CHAR** message stores each character and then calls **InvalidateRect().** In this version of the program, notice that inside the **WM_CHAR** case, each character you type is added to the string str. Thus, each time the window is repainted, the entire string containing all the characters you typed is output, not just the last character, as was the case with the preceding program.

```
/* A Windows skeleton that routes output through the WM_PAINT message. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc (HWND, UINT, WPARAM, LPARAM) ;
char szWinName [ ] = "MyWin" ; /* name of window class */
char str[255] ="" ; /* hold output string */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
    LPSTR IpszArgs, int nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1;
wc1.cbSize = sizeof(WNDCLASSEX); wc1.hInstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
wc1.hCursor = LoadCursor(NULL, IDC_ARROW);
```

```
wc1.lpszMenuName = NULL; wc1.cbClsExtra = 0;wc1.cbWndExtra =0;
wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
 if(!RegisterClassEx(&wc1)) return 0;
hwnd = CreateWindow (szWinName, "Routing Output Through WM_PAINT",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, NULL, hThisInst
NULL);
ShowWindow(hwnd, nWinMode); UpdateWindow(hwnd);
while(GetMessage(&msg, NULL, 0, 0))
{ TranslateMessage(&msg); /* allow use of keyboard */
DispatchMessage(&msg); /* return control to Windows NT */} return msg.wParam; }
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message, WPARAM
wParam, LPARAM lParam) {HDC hdc; PAINTSTRUCT paintstruct ; char temp [2] ;
switch (message) { case WM_CHAR: /* process keystroke */
                    hdc = GetDC (hwnd) ; /* get device context */
                    sprintf (temp, "%c", (char) wParam);/*stringize character */
                    strcat(str, temp); /* add character to string */
                    InvalidateRect (hwnd,NULL,1); /*paint the screen*/ break;
          case WM_PAINT: /* process a repaint request */
                    hdc = BeginPaint (hwnd, &paintstruct) ; /* get DC */
                    TextOut(hdc, 0, 0, str, strlen (str ) ) ,- /* output char */
                    EndPaint (hwnd, &paintstruct ) ; /* release DC */ break;
          case WM_DESTROY: /* terminate the program */
                    PostQuitMessage (0) ; break;
    default:  return DefWindowProc (hwnd, message, wParam, lParam); } return 0; }
```

Many Windows applications route all (or most) output to the client area through WM_PAINT, for the reasons already stated. However, there is nothing wrong with outputting text or graphics as needed. Which method you use will depend on the exact nature of each situation.

H.W: How Can return all text in program " text editor " answer by code segment. (In Multi line)

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Four

Application Essentials

(Mouse Message)

CONTENTS:

Responding to Mouse Messages

Since Windows is, to a great extent, a mouse-based operating system, all Windows NT programs should respond to mouse input. Because the mouse is so important, there are several different types of mouse messages. The ones discussed in this lecture are:

WM_LBUTTONDOWN	**WM_LBUTTONUP**	**WM-LBUTTONDBLCK**
WM_RBUTTONDOWN	**WM_RBUTTONUP**	**WM_RBUTTONDBLCK**

While most computers use a two-button mouse, Windows NT is capable of handling a mouse with up to three buttons. These buttons are called the left, middle, and right. For the rest of this chapter we will only be concerned with the left and right buttons.

Let's begin with the two most common mouse messages, **WM_LBUTTONDOWN** and **WM_RBUTTONDOWN**. They are generated when the left button and right button are pressed, respectively.

When either a **WM_LBUTTONDOWN** or a **WM_RBUTTONDOWN** message is received, the mouse's current X, Y location is specified in **LOWORD(LParam)** and **HIWORD(lParam)**, respectively. The value of wParam contains various pieces of status information, which are described in the next section.

The following program responds to mouse messages. Each time you press a mouse button when the program's window contains the mouse cursor, a message will be displayed at the current location of the mouse pointer. Figure 5-1 shows sample output from this program.

```
/* Process Mouse Messages. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[]= "MyWin"; /* name of window class */
char str[255] = ""; /* holds output string */
int WINAPI WinMain(INSTANCE hThisInst, HINSTANCE hPrevInst,
LPSTR lpszArgs, int nWinMode) { HWND hwnd; MSG msg; WNDCLASSEX wc1;
wc1.cbSize = sizeof (WNDCLASSEX) ; wc1.hInstance= hThisInst,
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION); wc1.cbClsExtra =0;
wc1.hIconSm = LoadIcon(NULL, IDI_APPLICATION); wc1.cbWndExtra = 0;
```

```
wc1.hCursor = LoadCursor(NULL, IDC_ARROW); wc1.lpszMenuName = NULL;
wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
 if(!RegisterClassEx(&wcl))  return 0;
hwnd = CreateWindow(szWinName,"Display WM_CHAR Messages Using TextOut",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT, HWND_DESKTOP,NULL,hThisInst, NULL);
ShowWindow (hwnd, nWinMode) ; UpdateWindow( hwnd) ;
while(GetMessage(&msg, NULL, 0, 0))
        { TranslateMessage(&msg);/*allow use of keyboard */
          DispatchMessage (&msg); }retur n msg. wParam;}
LRESULT CALLBACK WindowFunc(HWND hwnd,  U1NT  message,
WPARAM wParam,   LPARAM  lParam){ HDC hdc;
switch(message) {  case WM_RBUTTONDOWN: /* process right button */
                        hdc = GetDC(hwnd); /* get DC */
                        sprintf(str, "Right button is down at %d, %d", LOWORD(lParam),
                                HIWORD(lParam));
                        TextOut(hdc,LOWORD(lParam),HIWORD(lParam),str,strlen(str));
                        ReleaseDC(hwnd, hdc);break;
                  case WM_LBUTTONDOWN: /* process left button */
                     hdc = GetDC(hwnd); /* get DC */
                     sprintf(str, "Left button is down at %d, %d", LOWORD(lParam),
                                HIWORD(lParam));
                     TextOut(hdc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
                     ReleaseDC(hwnd, hdc); /* Release DC */ break;
      case WM_DESTROY: /*terminate the program*/ PostQuitMessage(0); break;
            default: return DefWindowProc(hwnd, message, wParam, IParam); } return 0;}
```

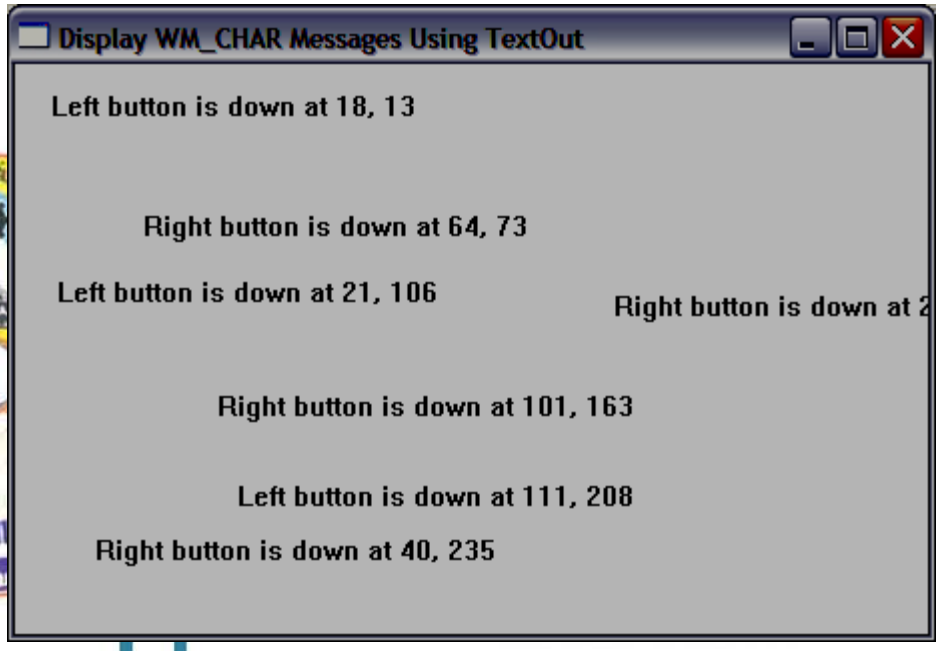


Figure 4-1: Sample output from the mouse message program

A Closer Look at Mouse Messages

For all of the mouse messages described in this lecture, the meaning of lParam and wParam is the same. As described earlier, the value of lParam contains the coordinates of the mouse when the message was generated. The value of wParam supplies information about the state of the mouse and keyboard. It may contain any combination of the following values:

MK_CONTROL
MK_SHIFT
MK_MBUTTON
MK_RBUTION
MK_LBUTTON

If the control key is pressed when a mouse button is pressed, then wParam will contain **MK_CONTROL**. If the shift key is pressed when a mouse button is pressed, then wParam will contain **MK_SHIFT**. If the right button is down when the left button is pressed, then wParam will contain **MK_RBUTTON**. If the left button is down when the right button is pressed, then wParam will contain **MK_LBUTTON**. If the middle button (if it exists) is down when one of the other buttons is pressed, then wParam will contain **MK_MBUTTON**. Before moving on, you might want to try experimenting with these messages.

Using Button Up Messages

When a mouse button is clicked, your program actually receives two messages. The first is a button down message, such as **WM_LBUTTONDOWN**, when the button is pressed. The second is a button up message, when the button is released. The button-up messages for the left and right buttons are called **WM_LBUTTONUP** and **WM_RBU TTONUP**.

For some applications, when selecting an item, it is better to process button-up, rather than button-down messages. This gives the user a chance to change his or her mind after the mouse button has been pressed.

Responding to a Double-Click

While it is easy to respond to a single-click, handling double-clicks requires a bit more work. First, you must enable your program to receive double-click messages. By default, double-click messages are not sent to your program. Second, you will need to add message response code for the double-click message to which you want to respond.

To allow your program to receive double-click messages, you will need, to specify **CS_DBLCLKS** in the style member of the **WNDCLASSEX** structure prior to registering the window class. That is, you must use a line of code like that shown here:

```
wc1.style = CS_DBLCLKS; /* allow double-clicks */
```

After you have enabled double-clicks, your program can receive these double-click messages: **WM_LBUTTONDBLCLK** and **WM_RBUTTONDBLCLK.** The contents of the lParam and wParam parameters are the same as for the other mouse messages.

As you know, a double-click is two presses of a mouse button in quick succession. You can obtain and/or set the time interval within which two presses of a mouse button must occur in order for a double-click message to be generated. To obtain the double-click interval, use the API function **GetDoubleClickTime()** , whose prototype is shown here:

```
UINT GetDoubleClickTime( void );
```

This function returns the interval of time (specified in milliseconds). To set the double-click interval, use **SetDoubleClickTime( )**. Its prototype is shown here:

```
BOOL SetDoubleClickTime ( UINT interval );
```

Here, interval specifies the number of milliseconds within which two presses of a mouse button must occur in order for a double-click to be generated. If you specify zero, then the

default double-click time is used. (The default interval is approximately half a second.) The function returns nonzero if successful and zero on failure.

The following program responds to double-click messages. It also demonstrates the use of **GetDoubleClickTime( ) and SetDoubleClickTime( )**. Each time you press the up arrow key, the double-click interval is increased. Each time you press the down arrow, the interval is decreased. Each time you double-click either the right or left mouse button, a message box that reports the current double-click interval is displayed. Since the double-click interval is a system-wide setting, changes to it will affect all other programs in the system. For this reason, when the program begins, it saves the current double-click interval. When the program ends, the original interval is restored. In general, if your program changes a system-wide setting, it should be restored before the program ends. Sample output is shown in This Program Below

```
/* Respond to double clicks and control the double-click interval */.
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[]="MyWin";  char str[255]=""; /*holds output string */
 UINT OrgDblClkTime; /* holds original double-click interval. */
int WIMAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
        LPSTR lpszArgs, int nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1;
wc1.cbSize = sizeof (WNDCLASSEX) ;wc1.hInstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc;
wc1.style = CS_DBLCLKS; /* enable double clicks */
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION); wc1.cbClsExtra =0;
wc1.hIconSm = LoadIcon(NULL, IDI_APPLICATION); wc1 .cbWndExtra = 0;
 wc1.hCursor = LoadCursor(NULL, IDC_ARROW);
wc1.lpszMenuName = NULL; wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
if(!RegisterClassEx(&wcl))  return 0;
hwnd = CreateWindow(szWinName,"Display WM_CHAR Messages Using TextOut",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,  CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP,NULL, hThisInst,NULL);
/* save original double-click time interval */ OrgDblClkTime=GetDoubleClickTime( );
```

```
ShowWindow (hwnd, nWinMode) ; UpdateWindow( hwnd) ;
while(GetMessage(&msg, NULL, 0, 0))
    { TranslateMessage(&msg); DispatchMessage (&msg); }retur n msg. wParam;}
LRESULT CALLBACK WindowFunc(HWND hwnd, U1NT message,
                WPARAM wParam, LPARAM lParam) { HDC hdc; UINT interval;
switch(message) { case WM_KEYDOWN:
if{(char)wParam==VK_UP) {/*increase interval*/ interval = GetDoubleClickTime();
                                    interval + = 100;
                                    SetDoubleClickTime(interval) ;}
if((char)wParam == VK_DOWN) { /* decrease interval */
                                interval = GetDoubleClickTime(); interval - = 100;
                                if(interval < 0) interval = 0;
                                SetDoubleClickTime(interval); }
sprintf(str, "New interval is %u milliseconds",interval);
MessageBox(hwnd, str, "Setting Double-Click Interval", MB_OK); break;
   case WM_RBUTTONDOWN:  hdc=GetDC(hwnd); /* get DC */
      sprintf(str,"Right button is down at %d, %d", LOWORD(lParam),HIWORD(lParam));
      TextOut(hdc,LOWORD(lParam),HIWORD(lParam),str,strlen(str));
       ReleaseDC(hwnd,hdc); break;
    case WM_LBUTTONDOWN: hdc=GetDC(hwnd); /* get DC */
      sprintf(str,"Left button is down at %d, %d",LOWORD(1Param),HIWORD(lParam));
      TextOut(hdc, LOWORD(IParam), HIWORD(IParam), str, strlen(str) ) ;
      ReleaseDC(hwnd, hdc); /* Release DC */break;
    case WM_LBUTTONDBLCLK:  interval  = GetDoubleClickTime ();
        sprintf(str,"Left ButtonXnlnterval  is %u milliseconds",  interval);
        MessageBox(hwnd, str, "DoubleClick", MB_OK); break;
 case WM_RBUTTONDBLCLK: interval  =  GetDoubleClickTime ( );
     sprintf(str,  "Right Button\nlnterval  is %u milliseconds",  interval);
     MessageBoxfhwnd,  str,  "Double Click",  MB_OK); break;
case WM_DESTROY: SetDoubleClickTime(OrgDblClkTime); PostQuitMessage(0);break;
default:  return DefWindowProc(hwnd, message, wParam, IParam); } return 0;}
```

Note: Output that leave the students But your carefully this program bellow have many errors correct this error and find what resultants obtain of this program below …!?

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Five:

Introducing Menus

CONTENTS:

This lesson begins our exploration of Windows NT's user interface components. If you are learning to program Windows for the first time, it is important to understand that your application will most often communicate with the user through one or more predefined interface components. There are several different types of these supported by Windows NT. This lecture introduces the most fundamental: the menu. Virtually any program you write will use one. As you will see, the basic style of the menu is predefined. You need only supply the specific information that relates to your application. Because of their importance, Windows NT provides extensive support for menus and the topic of menus is a large one. This lesson describes the fundamentals. Later, the advanced features are covered.

This lecture also introduces the resource. A resource is, essentially, an object defined outside your program but used by your program. Icons, cursors, menus, and bitmaps are common resources. Resources are a crucial part of nearly all Windows applications.

Menus Basics

The most common element of control within a Windows program is the menu. Windows NT supports three general types:

- The menu bar (or main menu)
- Pop-up submenus
- Floating, stand-alone pop-up menus

In a Windows application, the menu bar is displayed across the top of the window. This is frequently called the main menu. The menu bar is your application's top-level menu. Submenus descend from the menu bar and are displayed as pop-up menus. (You should be accustomed to this approach because it is used by virtually all Windows programs.) Floating pop-up menus are free-standing pop-up menus which are typically activated by pressing the right mouse button. In this lecture, we will explore the first two types: the menu bar and pop-up submenus. Floating menus are described in a later lesson.

It is actually quite easy to add a menu bar to your program's main window. To do so involves just these three steps:

1. Define the form of the menu in a resource file.
2. Load the menu when your program creates its main window.
3. Process menu selections.

In the remainder of this chapter, you will see how to implement these stops.

Since the first step is to define a menu in a resource file, it is necessary to explain resources and resource files.

Resources

Windows defines several common types of objects as resources. As mentioned at the beginning of this lesson, resources are, essentially, objects that are used by your program, but are defined outside your program. A menu is one type of resource. A resource is created separately from your program, but is added to the .EXE file when your program is linked. Resources are contained in resource files, which have the extension .RC. For small projects, the name of the resource file is often the same as that of your program's .EXE file. For example, if your program is called PROG.EXE, then its resource file will typically be called PROG.RC. Of course, you can call a resource file by any name you please as long as it has the .RC extension.

Depending on the resource, some are text-based and you create them using a standard text editor. Text resources are typically defined within the resource file. Others, such as icons, are most easily generated using a resource editor, but they still must be referred to in the RC file that is associated with your application. The example resource files in this lecture are simply text files because menus are text-based resources.

Resource files do not contain C or C++ statements. Instead, resource files consist of special resource statements. In the course of this lecture, the resource commands needed to support menus are discussed. Others are described as needed throughout this lectures.

Compiling .RC files

Resource files are not used directly by your program. Instead, they must be converted into a linkable format. Once you have created a .RC file, you compile it into a .RES file using a resource compiler. (Often, the resource compiler is called RC.EXE, but this varies.) Exactly how you compile a resource file will depend on what compiler you are using. Also, some integrated development environments handle this phase for you. For example, both Microsoft Visual C++ and Borland C++ compile and incorporate resource files automatically. In any event, the output of the resource compiler will be a .RES file and it is this file that is linked with your program to build the final Windows NT application.

Creating a Simple Menu

Menus are defined within a resource file by using the MENU resource statement. All menu definitions have this general form:

MenuName MENU [options]
{
menu items
}

Here, MenuName is the name of the menu. (It may also be an integer value identifying the menu, but all examples in this lecture will use the name when referring to the menu.) The keyword MENU tells the resource compiler that a menu is being created. There are only a few options that apply to Windows NT programs. They are shown here:

Option	Meaning
DISCARDABLE	Menu may be removed from memory when no longer needed
Characteristics info	Application-specific information, which is specified as a LONG value in info.
LANGUAGE lang, sub-lang	The language used by the resource is specified by lang and sub-lang. This is used by internationalized menus. The valid language identifiers are found in the header file WINNT.H.
VERSION ver	Application-defined version number is specified in ver.

Most simple applications do not require the use of any options and simply use the default settings.

There are two types of items that can be used to define the menu: **MENUITEMs** and **POPUPs**. A **MENUITEM** specifies a final selection. A **POPUP** specifies a pop-up submenu, which may contain other **MENUITEMs** or **POPUPs**. The general form of these two statements is shown here:

MENUITEM "ItemName". MenulD [, Options]
POPUP "PopupName" [, Options]

Here, ItemiName is the name of the menu selection, such as "Help" or "Save". MenuID is a unique integer associated with a menu item that will be sent to your application when a selection is made. Typically, these values are defined as macros inside a header file that is included in both your application code and its resource file. PopupName is the name of the pop-up menu. For both cases, the values for Options (defined by including WINDOWS.H) are shown in Table below.

Here is a simple menu that will be used by subsequent example programs. You should enter it at this time. Call the file MENU.RC.

Option	Meaning
CHECKED	A check mark is displayed next to the name. (Not applicable to top-level menus.)
GRAYED	The name is shown in gray and may not be selected.
HELP	May be associated with a help selection. Applies to MENUITEMs only.
INACTIVE	The option may not be selected.
MENUBARBREAK	For menu bar, causes the item to be put on a new line. For pop-up menus, causes the item to be put in a different column. In this case, the item is separated using a bar.
MENUBREAK	Same as MENUBARBREAK except that no separator bar is used.
SEPARATOR	Creates an empty menu item that acts as a separator. Applies to MENUITEMs only.

```
; Sample menu resource file.
 # include "menu.h"
MyMenu MENU
{POPUP "&File" {MENUITEM "&Open", IDM_OPEN
                MENUITEM "&Close", IDM_CLOSE
                MENUITEM "&Exit", IDM_EXIT}
 POPUP "&Options" {MENUITEM "&Colors", IDM_COLORS
                   POPUP "&Priority" {MENUITEM "&Low", IDM_LOW
                                      MENUITEM "&High", IDM_HIGH}
                   MENUITEM "&Fonts", IDM_FONT
                   MENUITEM "&Resolution" , IDM_RESOLUTION}
 MENUITEM "&Help", IDM_HELP}
```

This menu, called **MyMenu,** contains three top-level menu bar options: File, Options, and Help. The File and Options entries contain pop-up submenus. The Priority option activates a pop-up submenu of its own. Notice that options that activate submenus do not have menu ID values associated with them. Only actual menu items have ID numbers. In this menu, all menu ID values are specified as macros beginning with IDM. (These macros are defined in the header file MENU.H.) The names you give these values are arbitrary.

An & in an item's name causes the key that it precedes to become the shortcut key associated with that option. That is; once that menu is active, pressing that key causes that

menu item to be selected. It doesn't have to be the first key in the name, but it should be unless a conflict with another name exists.

NOTE: You can embed comments into a resource file on a line-by-line basis by beginning them with a semicolon, as the first line of the resource file shows. You may also use C and C++ style comments.

The MENU.H header file, which is included in MENU.RC, contains the macro definitions of the menu ID values. It is shown here. Enter it at this time.

```
#define IDM_OPEN        100
#define IDM_CLOSE       101
#define IDM_EXIT        102
#define IDM_COLORS      103
#define IDM_LOW         104
#define IDM_HIGH        105
#define IDM_FONT        106
#define IDM_RESOLUTION  107
#define IDM_HELP        108
```

This file defines the menu ID values that will be returned when the various menu items are selected. This tile will also be included in the program that uses the menu. Remember, the actual names and values you give the menu items are arbitrary, but each value must be unique. The valid range for ID values is 0 through 65,565.

Including a Menu in Your Program

Once you have created a menu, the easiest way to include that menu in a program is by specifying its name when you create the window's class. Specifically, you assign **lpszMenuName** a pointer to a string that contains the name of the menu. For example, to use the menu MyMeun, you would use this line when defining the window's class:

```
wcl.lpszMenuName = "MyMenu"; /* main menu */
```

Now **MyMenu** is the default main menu for all windows of its class. This means that all windows of this type will have the menu defined by **MyMenu**. (As you will see, you can override this class menu, if you like.)

Responding to Menu Selections:

Each time the user makes a menu selection, your program's window function is sent a **WM_COMMAND** command message. When that message is received, the value of **LOWORD(wParam)** contains the menu item's ID value. That is, **LOWORD(wParam)** contains the value you associated with the item when you defined the menu in its .RC file. Since **WM_COMMAND** is sent whenever a menu item is selected and the value associated with that item is contained in **LOWORD(wParam),** you will need to use a nested switch statement to determine which item was selected. For example, this fragment responds to a selection made from **MyMenu:**

```
switch(message) {case WM_COMMAND:
        switch(LOWORD(wParam)){
      case IDM_OPEN: MessageBox(hwnd, "Open File", "Open", MB_OK);break;
       case IDM_CLOSE:  MessageBox(hwnd, "Close File", "Close", MB_OK);break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?", "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);break;
     case IDM_COLORS:MessageBox(hwnd, "Set Colors", "Colors", MBJ3K);break;
    case IDM_LOW: MessageBox(hwnd, "Low", "Priority", MB_OK);break;
    case IDM_HIGH:    MessageBox(hwnd, "High", "Priority", MB_OK);break;
    case IDM_RESOLUTION:
            MessageBox(hwnd, "Resolution Options", "Resolution", MB_OK);break;
    case IDM_FONT:MessageBox(hwnd, "Font Options", "Fonts",  MB_OK);break;
   case IDM_HELP:MessageBox(hwnd, "No Help", "Help", MB_OK);break;} break;
```

For the sake of illustration, the response to each selection simply displays an acknowledgment of that selection on the screen. Of course, in a real application, the response to menu selections will perform the specified operations.

A Sample Menu Program

Here is a program that demonstrates the previously defined menu. Sample output from the program is shown in Figure 5-2.

```
/* Demonstrate menus. */
#tinclude <windows.h>
```

```
#include <string.h>
#nclude <stdio.h>
#include "menu.h"
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
int nWinMode) { HWND hwnd;  MSG msg; wndclassex wc1;
wc1.cbSize = sizeof(WNDCLASSEX); wc1.hlnstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION); wc1.cbClsExtra = 0;
 wc1.hIconSm = LoadIcon(NULL, IDI_WINLOGO); wc1.cbWndExtra =0;
wc1.hCursor = LoadCursor(NULL, IDC_ARROW); wc1.lpszMenuName = "MyMenu";
wc1.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
 if(!RegisterClassEx(&wcl)) return 0;
hwnd=CreateWindow(szWinName,"Introducing  Menus",  WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,CW_USEDEFAULT,  CW_USEDEFAULT,  CW_USEDEFAULT,
HWND_DESKTOP,NULL, hThisInst, NULL);
ShowWindow(hwnd, nWinMode); UpdateWindow(hwnd);
 while(GetMessage(&msg, NULL, 0, 0))
{TranslateMessage(&msg); DispatchMessage(&msg);} return msg.wParam; }
LRESULT  CALLBACK  WindowFunc(HWND  hwnd,  UINT  message,
WPARAM wParam, LPARAM lParam)  { int response;
switch(message) { case WM_COMMAND:
   switch(LOWORD(wParam))
        { case IDM_OPEN: MessageBox(hwnd, "Open File", "Open", MB_OK); break;
         case IDM_CLOSE: MessageBox(hwnd, "Close File", "Close", MB_OK); break;
case IDM_EXIT:
      response = MessageBox(hwnd, "Quit the Program?", "Exit", MB_YESNO);
      if(response == IDYES) PostQuitMessage(0); break;
  case IDM_COLORS:
            MessageBox(hwnd,  "Set Colors",  "Colors",  MB_OK); break;
 case IDM_LOW: MessageBox(hwnd, "Low", "Priority", MB_OK); break;
```

```
case IDM_HIGH: MessageBox(hwnd, "High", "Priority",MB_OK);break;
case IDM_RESOLUTION: MessageBox(hwnd,"Resolution Options","Resolution", MB_OK);break;
case IDM_FONT: MessageBox(hwnd, "Font Options", "Fonts",MB_OK);break;
case IDM_HELP: MessageBox(hwnd, "No Help", "Help", MB_OK);break; break;
case WM_DESTROY: /* terminate the program */ PostQuitMessage(0); break;
default: return DefWindowProc(hwnd, message, wParam, lParam); } return 0;}
```

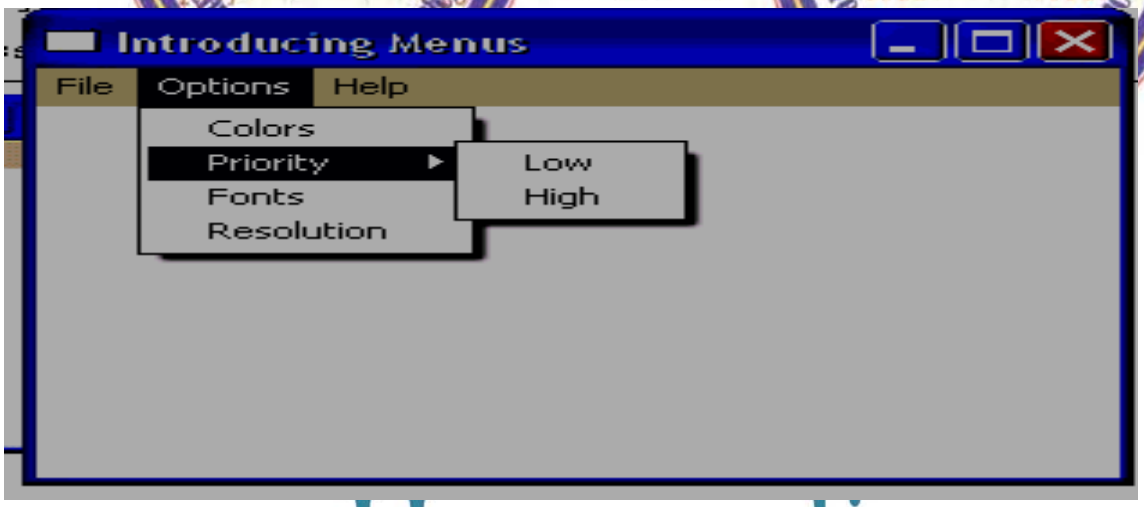


Figure 5.2: output of below program

In Depth: Using MessageBox() Responses

In the example menu program, when the user selects Exit, the following code sequence is executed:

```
case IDM_EXIT:
response = MessageBox(hwnd, "Quit the Program?", "Exit", MB_YESNO);
if(response == IDYES) PostQuitMessage(0); break;
```

As you can see, it contains two buttons: Yes and No. As discussed in Chapter (message box), a message box will return the user's response. In this case, it means that MessageBox( ) will return either IDYES or IDNO. If the user's response is IDYES, then the program terminates. Otherwise, it continues execution.

This is an example of a situation in which a message box is used to allow the user to select between two courses of action. As you begin to write your own Windows NT programs, keep in mind that the message box is useful in any situation in which the user must choose between a small number of choices.

Adding Menu Accelerator Keys

There is one feature of Windows that is commonly used in conjunction with a menu. This feature is the accelerator key. Accelerator keys are special keystrokes that you define which, when pressed, automatically select a menu option even though the menu in which that option resides is not displayed. Put differently, you can select an item directly by pressing an accelerator key, bypassing the menu entirely. The term accelerator key is an accurate description because pressing one is generally a faster way to select a menu item than first activating its menu and then selecting the item.

To define accelerator keys relative to a menu, you must add an accelerator key table to your resource file. An accelerator table has this general form:

TableNamee ACCELERATORS [acccl-options] {
Keyl, MenuID1 [, type] [options]
Key2, MenuID2 [, type] [options]
Key3, MenuID3 [, type] [options]
.
.
KeyN, MenuIDN [, type] [options] }

Here, Tublename is the name of the accelerator table. An **ACCELERATORS** statement can have the same options as those described for MENU. If needed, they are specified by accel-options. However, most applications simply use the default settings.

Inside the accelerator table, Key is the keystroke that selects the item and MenuID is the ID value associated with the desired item. The type specifies whether the key is a standard key (the default) or a virtual key. The options may be one of the following macros: **NOINVERT, ALT, SHIFT, and CONTROL**. **NOINVERT** prevents the selected menu item from being highlighted when its accelerator key is pressed. **ALT** specifies an alt key. **SHIFT** specifies a shift key. **CONTROL** specifies a ctrl key.

The value of Key will be either a quoted character, an ASCII integer value corresponding to a key, or a virtual key code. If a quoted character is used, then it is assumed to be an ASCII character. If it is an integer value, then you must tell the resource compiler explicitly that this is an ASCII character by specifying type as **ASCII**. If it is a virtual key, then type must be **VIRTKEY**.

If the key is an uppercase quoted character then its corresponding menu item will be selected if it is pressed while holding down the shift key. If it is a lowercase character, then its menu item will be selected if the key is pressed by itself. If the key is specified as a lowercase character and ALT is specified as an option, then pressing alt and the character will select the item. (If the key is uppercase and ALT is specified, then you must press shiff and alt to select the item.) Finally, if you want the user to press ctrl and the character to select an item, precede the key with a ^.

As explained in lecture 4, a virtual key is a system-independent code for a variety of keys. To use a virtual key as an accelerator, simply specify its macro for the key and specify VIRTKEY for its type. You may also specify **ALT, SHIFT**, or **CONTROL** to achieve the desired key combination.

Here are some examples:

"A", IDM_x ; select by pressing Shift-A

"a", IDM_x " ; select by pressing a

"^a", IDM_x " ; select by pressing Ctrl-a

"a", IDM_x, ALT ; select by pressing Alt-a

VK_F2, IDM_x ; select by pressing F2

VK_F2, IDM_x, SHIFT ; select by pressing Shift-F2

Here is the MENU.RC resource file that also contains accelerator key definitions for **MyMenu.**

```
;   Sample menu resource file and accelerators.
# include <windows.h>
# include   "menu.h"
MyMenu MENU
{POPUP "&File"{MENUITEM   "&Open\tF2",   IDM_OPEN
                    MENUITEM   "&Close\tF3",   IDM_CLOSE
                    MENUITEM   "&Exit \t Ctrl-X",    IDM_EXIT}
POPUP "^Options"{MENUITEM   "&Colors\t Ctrl-C", IDM_COLORS
                    POPUP"&Priority"{MENUITEM "&Low\tF4", IDM_LOW
                                     MENUITEM"&High\tF5", IDM_HIGH}
                                     MENUITEM "&Fonts\t Ctrl-F", IDM_FONT
                                     MENUITEM"&Resolution\tCtrl-R",IDM_RESOLUTION}
MENUITEM "&Help", IDM_HELP}
```

```
; Define menu accelerators
MyMenu ACCELERATORS
{  VK_F2, IDM_OPEN, VIRTKEY
 VK_F3, IDM_CLOSE, VIRTKEY
"^X", IDM_EXIT
"^C", IDM_COLORS
VK_F4, IDM_LOW, VIRTKEY
VK_F5, IDM_HIGH, VIRTKEY
"^F", IDM_FONT
"^R", IDM_RESOLUTION
VK_F1, IDM_HELP, VIRTKEY }
```

Notice that the menu definition has been enhanced to display which accelerator key selects which option. Each item is separated from its accelerator key using a tab. The header file WINDOWS.H is included because it defines the virtual key macros.

Loading the Accelerator Table

Even though the accelerators are contained in the same resource file as the menu, they must be loaded separately using another API function called LoadAccelerators( ), whose prototype is shown here:

HACCEL LoadAccelerators (HINSTANCE ThisInst, LPCSTR Name);

where ThisInst is the instance handle of the application and Name is the name of the accelerator table. The function returns a handle to the accelerator table or NULL if the table cannot be loaded.

You must call LoadAccelerators( ) soon after the window is created. For example, this shows how to load the MyMenu accelerator table:

```
HACCEL hAccel;
hAccel = LoadAccelerators(hThisInst, "MyMenu");
```

The value of hAccel will be used later to help process accelerator keys.

Translating Accelerator Keys

Although the LoadAccelerators() function loads the accelerator table, your program still cannot process accelerator keys until you add another API function to the message loop. This function is called **TranslateAcceIerator( )** and its prototype is shown here:

int TransIateAccclerator(HWND hwnd, HACCEL hAccel, LPMSG lpMess);

Here, hwnd is the handle of the window for which accelerator keys will be translated. hAcccl is the handle to the accelerator table that will be used. This is the handle returned by **LoadAccelerators( )**. Finally, lpMess is a pointer to the message. The **TrauslateAcceIerator( )** function returns true if an accelerator key was pressed and false otherwise.

TranslateAcccIerator( ) translates an accelerator keystroke into its corresponding **WM_COMMAND** message and sends that message to the window. In this message, the value of **LOWORD(wParam)** will contain the ID associated with the accelerator key. Thus, to your program, the **WM_COMMAND** message will appear to have been generated by a menu selection.

Since **TranslateAccelerator( )** sends a **WM_COMMAND** message whenever an accelerator key is pressed, your program must not execute **TranslateMessage( )** or DispatchMessage( ) when such a translation takes place. When using **TranslateAccelcrator( )**, your message loop should look like this:

```
while(GetMessage(&msg, NULL, 0, 0))
    {if(!TranslateAccelerator(hwnd, hAccel, &msg))
        {TranslateMessage(&msg); /*allow use of keyboard */
        DispatchMessage(&msg); /*return control to Windows*/}}
```

Trying Accelerator Keys

To try using accelerators, substitute the following version of **WinMain( )** into the preceding application and add the accelerator table to your resource file.

```
/* Process accelerator keys. */
#include    <windows.h>
#include    <string.h>
#include    <stdio.h>
#include    "menu.h"
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[ ] = "MyWin"; /* name of window class */
```

```
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR IpszArgs,
int nWinMode)  {HWND hwnd; MSG msg; WNDCLASSEX wc1; HACCEL hAccel;
wc1.cbSize = sizeof(WNDCLASSEX); wc1 .hInstance = hThisInst;
wc1 .lpszClassName = szWinName; wc1. lpfnWndProc = WindowFunc; wc1. style =0;
wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION) ;
wc1.hlconSm = LoadIcon (NULL, IDI_WINLOGO) ;
wc1.hCursor = LoadCursor (NULL, IDC_ARROW) ;
wc1. lpszMenuName = "MyMenu"; wc1 .cbClsExtra =0; wc1 .cbWndExtra = 0;
wc1.hbrBackground = GetStockObject (WHITE_BRUSH) ;
if ( !RegisterClassEx (&wc1) ) return 0 ;
hwnd = CreateWindow (szWinName,"Adding Accelerator Keys",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT,HWND_DESKTOP,NULL, hThisInst, NULL  );
/* load the keyboard accelerators */  hAccel = LoadAccelerators (hThisInst, "MyMenu")
/*Display the window.*/  ShowWindow (hwnd, nWinMode) ;  UpdateWindow(hwnd) ;
while(GetMessage(&msg, NULL, 0, 0)){if ( ! TranslateAccelerator (hwnd,' hAccel,. &msg) )
  {TranslateMessage(&msg); DispatchMessage(&msg)}} return msg. wParam; }
```

In Depth: A Closer Look at WM_COMMAND

As you know, each time you make a menu selection or press an accelerator key, a **WM_COMMAND** message is sent and the value in **LOWORD(wParam)** contains the ID of the menu item selected or the accelerator key pressed. However, using only the value in **LOWORD(wParam)** it is not possible to determine which event occurred. In most situations, it doesn't matter whether the user actually made a menu selection or just pressed an accelerator key. But in those situations in which it does, you can find out because Windows provides this information in the high-order word of **wParam**. If the value in **HIWORD(wParam)** is 0, then the user has made a menu selection. If this value is 1, then the user pressed an accelerator key. For example, try substituting the following fragment into the menu program. It reports whether the Open option was selected using the menu or by pressing an accelerator key.

```
case IDM_OPEN:
if(HIWORD(wParam))  MessageBox(hwnd, "Open File via Accelerator", "Open", MB_OK);
Else  MessageBox(hwnd, "Open File via Menu Selection", "Open", MB_OK);  break;
```

The value of **lParam** for **WM_COMMAND** messages generated by menu selections or accelerator keys is unused and always contains **NULL**.

As you will see in the next lecture, a **WM_COMMAND** is also generated n when the user interacts with various types of controls. In this case, the meanings of **lParam** and **wParam** are somewhat different. For example, the value of **lParam** will contain the handle of the control.

Non-Menu Accelerator Keys

Although keyboard accelerators are most commonly used to provide a fast means of selecting menu items, they are not limited to this role. For example, you can define an accelerator key for which there is no corresponding menu item. You might use such a key to activate a keyboard macro or to initiate some frequently used option. To define a non-menu accelerator key, simply add it to the accelerator table, assigning it a unique ID value.

As an example, let's add a non-menu accelerator key to the menu program,| The key will be ctrl-t and each time it is pressed, the current time and date are displayed in a message box. The standard ANSI C time and date functions are used to obtain the current time and date. To begin, change the key table so that it looks like this:

```
MyMenu ACCELERATORS {VK_F2, IDM_OPEN, VIRTKEY
                     VK_F3, IDM_CLOSE, VIRTKEY
                     "^X", IDM_EXIT
                     "^C", IDM_COLORS
                     VK_F4, IDM_LOW, VIRTKEY
                     VK_F5, IDM_HIGH, VIRTKEY
                     "^R", IDM_RESOLUTION
                     "^F", IDM_FONT
                     VK_F1, IDM_KELP, VIRTKEY
                     "^T", IDM_TIME}
```

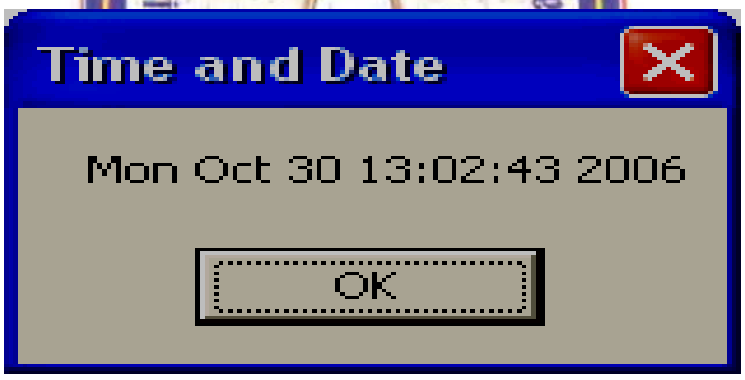
Next, add this line to MENU.H:

```
#define IDM_TIME  500
```

Finally, substitute this version of **WindowFunc( )** into the menu program You will also need to include the TIME.H header file.

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam,
LPARAM lParam)
  {int response; struct tm *tod; time_t t; char str [80];
switch(message) {case WM_COMMAND:
                    switch(LOWORD(wParam))
{case  IDM_OPEN:  MessageBox(hwnd,  "Open File",  "Open",  MB_OK);break;
 case IDM_CLOSE:  MessageBox(hwnd,  "Close File",  "Close",  MB_OK);break;
 case IDM_EXIT:response=MessageBox(hwnd,"Quit the Program?","Exit", MB_YESNO);
                    if(response  ==  IDYES)  PostQuitMessage(0); break;
 case IDM_COLORS:  MessageBox(hwnd,  "Set Colors",  "Colors",  MB_OK);break;
 case  IDM_LOW:  MessageBox(hwnd,  "Low",  "Priority",  MB_OK);break;
case  IDM_HIGH:  MessageBox(hwnd,  "High",  "Priority",  MB_OK);break;
case  IDM_RESOLUTION:  MessageBox(hwnd,"Resolution  Options","Resolution",
MB_OK);break;
case IDMJFONT:  MessageBox(hwnd,  "Font Options",  "Fonts",  MB_OK); break;
case IDM_TIME: /* show time */ t = time(NULL); tod = localtime(&t);
                    strcpyfstr, asctime(tod)); str(strlen(str)-1] = '\0'; /* remove /r/n */
                    MessageBox(hwnd, str, "Time and Date", MB_OK); break;
 case IDM_HELP: MessageBox(hwnd,"No Help","Help",MB_OK); break; }break;
 case WM_DESTROY:  PostQuitMessage(0); break;
 default:  return DefWindowProc(hwnd, message, wParam, lParam);} return 0; }
```

When you run this program, each time you press ctrl-t, you will see a message box similar to the following:



Overriding the Class Menu

In the preceding programs, the main menu has been specified in the **lpszMenuName** member of the **WNDCLASSEX** structure. As mentioned, this specifies a class menu that will be used by all windows that are created of its class. This is the way most main menus are specified for simple applications. However, there is another way to specify a main menu that uses the CreateWindow( ) function. As you may recall from Lecture 2, **CreateWindow( )** is defined like this:

```
HWND CreateWindow (
LPCSTR lpClassName, /* name of window class */
LPCSTR lpWinName, /* title of window */
DWORD dwStyle, /* type of window */
int X, int Y, /* upper-left coordinates */
int Width, int Height, /* dimensions of window */
HWND hParent, /* handle of parent window */
HMENU hMenu, /* handle of main menu */
HINSTANCE hThisInst, /* handle of creator */
LPVOID lpszAdditional /* pointer to additional info */   );
```

Notice the hMenu parameter. It can be used to specify a main menu for the window being created. In the preceding programs, this parameter has been specified as **NULL**. When hMenu is **NULL**, the class menu is used. However, if it contains the handle to a menu, then that menu will be used as the main menu for the window being created. In this case, the menu specified by hMenu overrides the class menu. Although simple applications, such as those shown in this book, do not need to override the class menu, there can be times when this is beneficial. For example, you might want to define a generic window class which your application will tailor to specific needs.

To specify a main menu using **CreateWindow( )**, you need a handle to the menu. The easiest way to obtain one is by calling the **LoadMenu( )** API function, shown here:

HMENU LoadMenu(HINSTANCE hInst, LPCSTR lpName);

Here, hInst is the instance handle of your application. A pointer to the name of the menu is passed in lpName. **LoadMenu( )** returns a handle to the menu if successful or NULL on

failure. Once you have obtained a handle to a menu, it can be used as the hMenu parameter to **CreateWindow( )**.

When you load a menu using **LoadMenu( )** you are creating an object that allocates memory. This memory must be released before your program ends. If the menu is linked to a window, then this is done automatically. However, when it is not, then you must free it explicitly. This is accomplished using the **DestroyMenu( )** API function. Its prototype is shown here: **BOOL DestroyMenu(HMENU hMenu);**

Here, hMenu is the handle of the menu being destroyed. The function returns nonzero if successful and zero on failure. As stated, you will not need to use **DestroyMenu( )** if the menu you load is linked to a window.

An Example that Overrides the Class Menu

To illustrate how the class menu can be overridden, let's modify the preceding menu program. To do so, add a second menu, called **PlaceHolder**, to the MENU.RC file, as shown here. ; Define two menus.

```
#include <windows.h>
#include "menu.h"
; Placeholder class menu.
PlaceHoIder MENU
{POPUP "&File"
{MENUITEM "&Exit\t Ctrl-X", IDM_EXIT}
MENUITEM "&Help", IDM_HELP}
; Menu used by CreateWindow.
 MyMenu MENU
{POPUP "&File"{MENUITEM "&Open\t F2", IDM_OPEN
                MENUITEM "&Close\t F3", IDM_CLOSE
                MENUITEM "&Exit\t Ctrl-X", IDM_EXIT }
POPUP "&Options" {MENUITEM "&Colors\t Ctrl-C", IDM_COLORS
                POPUP "&Priority"{MENUITEM "&Low\t F4", IDM_LOW
                MENUITEM "&High\tF5", IDM_HIGH}
                MENUITEM "&Font\t Ctrl-F", IDM_FONT
                MENUITEM "&Resolution\t Ctrl-R", IDM_RESOLUTION}
  MENUITEM "&Help", IDM_HELP}
```

```
; Define menu accelerators
MyMenu ACCELERATORS {VK_F2,    IDM_OPEN,    VIRTKEY
VK_F3, IDM_CLOSE, VIRTKEY
"^X", IDM_EXIT
"^C", IDM_COLORS
VK_F4, IDM_LOW, VIRTKEY
VK_F5, IDM_HIGH, VIRTKEY
"^F", IDM_FONT
"^R", IDM_RESOLUTION
VK_F1, IDM_HELP, VIRTKEY
"^T", IDM_TIME }
```

The **PlaceHoIder** menu will be used as the class menu. That is, it will be assigned to the lpszMemuName member of **WNDCLASSEX**. **MyMenu** will be loaded separately and its handle will be used in the hMenu parameter of **CreateWindow( ).** Thus, **MyMenu** will override **PlaceHoIder.**

The contents of MENU.H are shown here. They are unchanged from the original version except for the addition of **IDM_TIME**, from the previous section.

```
#define IDM_OPEN        100
#define IDM_CLOSE       101
#define IDM_EXIT        102
#define IDM_COLORS      103
#define IDM_LOW         104
#define IDM_HIGH        105
#define IDM_FONT        106
#define IDM_RESOLUTION  107
#define IDM_HELP        108
#define IDM_TIME        500
```

==========

Here is the complete program that overrides the class menu. This program incorporates all of the features discussed in this lecture. Since we have made so many changes to the menu program throughout the course of this chapter, the entire program is shown here for your convenience.

```
/* Overriding the class menu. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "menu.h"
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
 char szWinName[ ] = "MyWin"; /* name of window class */
int WINAPI WinMain(HINSTANCE hThislnst, HINSTANCE hPrevInst, LPSTR lpszArgs,
int nWinMode){HWND hwnd;MSG msg; WNDCLASSEX wcl;HACCEL hAccel;
HMENU hmenu;
wc1.cbSize = sizeof(WNDCLASSEX); wc1.hlnstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hlcon = LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm = LoadIcon (NULL, IDI_WINLOGO );
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);wc1.lpszMenuName = "PlaceHolder";
wc1.cbClsExtra =0;wc1.cbWndExtra =0;
wc1.hbrBackground=GetStockObject(WH1TE_BRUSH); if(IRegisterClassEx(&wc1))
return 0;
/* load main menu manually */ hmenu = LoadMenu(hThisInst, "MyMenu");
/* Now that a window class has been registered, a window can be created. */
hwnd = CreateWindow(szWinName, "Using an Alternative Menu",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,HWND_DESKTOP, hmenu, hThisInst,NULL );
/* load the keyboard accelerators */ hAccel = LoadAccelerators (hThisInst, "MyMenu");
/* Display the window. */  ShowWindow (hwnd, nWinMode);  UpdateWindow(hwnd) ;
while (GetMessage(&msg, NULL, 0, 0))
  {if ( !TranslateAccelerator (hwnd, hAccel, &msg) )
       {TranslateMessage(&msg) ; DispatchMessage(&msg) ; } } return msg.wParam;   }
/*This function is called by Windows NT and is passed messages from the message queue.
*/
```

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {int response;struct tm *tod;t i me_t t;char str[80];
switch(message)
{case WM_COMMAND:
switch(LOWORD(wParam)} {
case IDM_OPEN: MessageBox(hwnd, "Open File", "Open", MB_OK);break;
case IDM_CLOSE: MessageBox(hwnd, "Close File", "Close", MB_OK);break;
case IDM_EXIT: response=MessageBox(hwnd, "Quit the Program?","Exit", MB_YESNO);
    if(response == IDYES) PostQuitMessage(O); break;
case IEM_COLORS: MessageBox(hwnd, "Set Colors", "Colors", MB_OK);break;
case IDM_LOW: MessageBox(hwnd, "Low", "Priority", MB_OK);break;
case IDM_HIGH: MessageBox (hwnd, "High", "Priority", MB_OK) ;break;
case IDM_RESOLUTION: MessageBox(hwnd,"Resolution Options", "Resolution", MB_OK);  break;
case IDM_FONT: MessageBox(hwnd, "Font Options", "Fonts", KB_QK);break;
case IDM__TIME: /* show time */ t = time(NULL);
   tod = localtime(&t); strcpy(str, asctime(tod));
   str[strlen(str)-1] = '\0'; /* remove /r/n */
   MessageBox(hwnd, str, "Time and Date", MB_OK); break;
case IDM_HELP: MessageBox (hwnd, "No Help", "Help", MB_OK) ;break;}break;
case WM_DESTROY: /* terminate the program */ PostQuitMessage(0); break;
default:  return DefWindowProc(hwnd, message, wParam, IParam); }return 0; }
```

Pay special attention to the code inside **WinMain().** It creates a window class that specifies **PlaceHoIder** as its class menu. However, before a window is actually created, **MyMenu** is loaded and its handle is used in the call to **CreateWindow( ).** This causes the class menu to be overridden and **MyMenu** to be displayed. You might want to experiment with this program a little. For example, since the class menu is being overridden, there is no reason to specify one at all. To prove this, assign **lpszMenuName** the value **NULL.** The operation of the program is unaffected.

In this example, both **MyMenu** and **PlaceHoIder** contain menus that can be processed by the same window function. That is, they both use the same set of menu IDs. (Of course,

PlaceHoIder only contains two selections.) This allows either menu to work in the preceding program. Although you are not restricted in the form or structure of an overriding menu, you must always make sure that whatever menu you use, your window function contains the proper code to respond to it.

One last point: since **MyMenu** is linked to the window created by **CreateWindow( ),** it is destroyed automatically when the program terminates. There is no need to call **DestroyMenu( ).**

==========

REMEMBER: It is usually easier to specify the main menu using WNDCLASSEX rather than CreateWindow(). This is the approach used by the rest of the programs in this lesson.

Q: How window recognize in menu that selection manually or used accelerator keys.

Q: deference step in create menu and accelerator in program.

Q: give top deference between load menu and load accelerator .

Q what types of methods to including menu in program explain by details.

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Six:

Dialog Box

CONTENTS:

Although menus are an important part of nearly every Windows NT application, they cannot be used to handle all types of user responses. For example, it would be difficult to use a menu to input the time or date. To handle all types of input, Windows provides the dialog box. A dialog box is a special type of window that provides a flexible means by which the user can interact with your application. In general, dialog boxes allow the user to select or enter information that would be difficult or impossible to enter using a menu. In this lecture, you will learn how to create and manage a dialog box.

Also discussed in this lecture are three of Windows' standard controls. Within a dialog box, interaction with the user is performed through a control. In a sense, a dialog box is simply a container that holds various control elements.

As a means of illustrating the dialog box and several controls, a very simple database application will be developed. The database contains the titles of several books along with the names of their authors, publishers, and copyright dates. The dialog box created in this lecture will allow you to select a title and obtain information about it. While the database example is, necessarily, quite simple, it will give you the flavor of how a real application can effectively use a dialog box.

Dialog Boxes Use Controls

Windows NT supports several standard controls, including push buttons, check boxes, radio buttons, list boxes, edit boxes, combo boxes, scroll bars, and static controls. (Windows NT also supports several enhanced controls called common controls, which are discussed later in this next lecture.) In the course of explaining how to use dialog boxes, the examples in this lecture illustrate three of these controls: the push button, the list box, and the edit box. In the next lecture, other controls will be examined.

A **push button** is a control that the user "pushes on" to activate some response. You have already been using push buttons in message boxes. For example, the OK button that we have been using in most message boxes is a push button. A **list box** displays a list of items from which the user selects one (or more). List boxes are commonly used to display things such as file names.

An **edit box** allows the user to enter a string. Edit boxes provide all necessary text editing features.

Therefore, to input a string, your program simply displays an edit box and waits until the user has finished typing in the string. Typically, a combo box is a combination of a list box and an edit box.

It is important to understand that controls both generate messages (when accessed by the user) and receive messages (from your application). A message generated by a control indicates what type of interaction the user has had with the control. A message sent to the control is essentially an instruction to which the control must respond.

Modal vs. Modeless Dialog Boxes

There are two types of dialog boxes: modal and modeless. The most common dialog boxes are modal. A modal dialog box demands a response from the user before the program will continue. When a modal dialog box is active, the user cannot refocus input to another part of the application without first closing the dialog box. More precisely, the owner window of a modal dialog box is deactivated until the dialog box is closed. (The owner window is usually the one that activates the dialog box.)

A modeless dialog box does not prevent other parts of the program from being used. That is, it does not need to be closed before input can be refocused to another part of the program. The owner window of a modeless dialog box remains active. In essence, modeless dialog boxes are more independent than modal ones.

We will examine modal dialog boxes first, since they are the most common. A modeless dialog box example concludes this lecture.

Receiving Dialog Box Messages

A dialog box is a type of window. Events that occur within it are sent to your program using the same message-passing mechanism that the main window uses. However, dialog box messages are not sent to your program's main window function. Instead, each dialog box that you define will need, its own window function, which is generally called a dialog function or dialog procedure. This function must have this prototype. (Of course, the name of the function may be anything that you like.)

BOOL CALLBACK DFunc(HWND Hwnd, UINT message, WPARAM wParam, LPARAM lParam);

As you can see, a dialog function receives the same parameters as your program's main window function. However, it differs from the main window function in that it returns a true or false result. Like your program's main window function, the dialog box window function will receive many messages. If it processes a message, then it must return true. If it does not respond to a message, it must return false.

In general, each control within a dialog box will be given its own resource ID. Each time that control is accessed by the user, a **WM_COMMAND** message will be sent to the dialog function, indicating the ID of the control and the type of action the user has taken. The function will then decode the message and take appropriate actions. This process parallels the way messages are decoded by your program's main window function.

Activating a Dialog Box

To activate a modal dialog box (that is, to cause it to be displayed) you must call the **DialogBox( )** API function, whose prototype is shown here:

int DialogBox (HINSTANCE hThisInst, LPCSTR lpName, HWND hwnd, DLGPROC lpDFunc);

Here, hThisInst is a handle to the current application that is passed to your program in the instance parameter to **WinMain( )**. The name of the dialog box as defined in the resource file is pointed to by lpName. The handle to the window that owns the dialog box is passed in hwnd. (This is typically the handle of the window that calls **DialogBox( )**.) The lpDFunc parameter contains a pointer to the dialog function described in the preceding section. If DialogBox( ) fails, it returns -1.Otherwise, the return value is that specified by **EndDialog( )** ,discussed next.

Deactivating a Dialog Box

To deactivate (that is, destroy and remove from the screen) a modal dialog box, use **EndDialog( )**. It has this prototype**:** **BOOL EndDialog(HWND hdwnd, int nStatus);**

Here, hdwnd is the handle to the dialog box and nStatus is a status code returned by the **DialogBox()** function. (The value of nStatus may be ignored, if it is not relevant to your program.) This function returns nonzero if successful and zero otherwise. (In normal situations, the function is successful.)

Creating a Simple Dialog Box

To illustrate the basic dialog box concepts, we will begin with a simple dialog box. This dialog box will contain four push buttons called Author, Publisher, Copyright, and Cancel. When either the Author, Publisher, or Copyright button is pressed, it will activate a message box indicating the choice selected. (Later these push buttons will be used to obtain information from the database. For now, the message boxes are simply placeholders.) The dialog box will be removed from the screen when the Cancel button is pressed.

The Dialog Box Resource File

A dialog box is another resource that is contained in your program's resource file. Before developing a program that uses a dialog box, you will need a resource file that specifies one. Although it is possible to specify the contents of a dialog box using a text editor and enter its specifications as you do when creating a menu, this is seldom done. Instead, most programmers use a dialog editor. The main reason for this is that dialog box definitions involve the positioning of the various controls inside the dialog box, which is best done interactively. However, since the complete .RC files for the examples in this lecture are supplied in their text form, you should simply

enter them as text. Just remember that when creating your own dialog boxes, you will want to use a dialog editor.

Dialog boxes are defined within your program's resource file using the **DIALOG** statement. Its general form is shown here:

```
Dialog-name DIALOG [DISCARDABLE] X, Y, Width, Height
Features
{
Dialog-items
}
```

The Dialog-name is the name of the dialog box. The box's upper left corner will be at X, Y and the box will have the dimensions specified by Width and Height. If the box may be removed from memory when not in use, then specify it as DISCARDABLE. One or more optional features of the dialog box may be specified. As you will see, two of these are the caption and the style of the box. The Dialog-items are the controls that comprise the dialog box.

The following resource file defines the dialog box that will be used by the first example program. It includes a menu that is used to activate the dialog box, the menu accelerator keys, and then the dialog box itself. You should enter it into your computer at this time, calling it DIALOG.RC.

```
; Sample dialog box and menu resource file.
#include <windows.h>
#include "dialog.h"
MyMenu MENU
{POPUP "&Dialog" {MENUITEM "&Dialog\tF2", IDM_DIALOG
                  MENUITEM "&Exit\tF3", IDM_EXIT }
MENUITEM "&Help", IDM_HELP }
MyMenu ACCELERATORS
 {  VK_F2, IDM_DIALOG, VIRTKEY
    VK_F3, IDM_EXIT, VIRTKEY
    VK_F1, IDM_HELP, VIRTKEY  }
MyDB DIALOG 10, 10, 210, 110 CAPTION "Books Dialog Box"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{  DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14,
WS_CHILD | WS_VISIBLE | WS_TABSTOP
  PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14,
          WS_CHILD | WS_VISIBLE | WS_TABSTOP
```

```
PUSHBUTTON "Copyright", IDD_COPYRIGHT,11, 58, 36, 14,
        WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16
        WS_CHILD | WS_VISIBLE | WS_TABSTOP   }
```

This defines a dialog box called MyDB that has its upper left corner at location 10, 10. Its width is 210 and its height is 110. The string after CAPTION becomes the title of the dialog box. The **STYLE** statement determines what type of dialog box is created. Some common style values, including those used in this lecture, are shown in Table 7-1. You can OR together the values that are appropriate for the style of dialog box that desire. These style values may also be used by other controls.

Value	Meaning
DS_MODALFRAME	Dialog box has a modal frame. This style can be used with either modal or modeless dialog boxes.
WS_BORDER	Include a border.
WS_CAPTION	Include title bar.
WS_CH1LD	Create as child window.
WS_POPUP	Create as pop-up window.
WS_MAXIMIZEBOX	Include maximize box.
WS_MINIMIZEBOX	Include minimize box.
WS_SYSMENU	Include system menu.
WS_TABSTOP	Control may be tabbed to.
WS_VISIBLE	Box is visible when activated.

Within the MyDB definition are defined four push buttons. The first is the default push button. This button is automatically highlighted when the dialog box is first displayed. The general form of a push button declaration is shown here:

PUSHBUTTON "string", PBID, X, Y, Width, Height [, Style]

Here, string is the text that will be shown inside the push button. PDID is the value associated with the push button. It is this value that is returned to your program when this button is pushed. The button's upper left corner will be at X, Y and the button will have the dimensions specified by Width and Height. Style determines the exact nature of the push button. To define a default push button use the DEFPUSHBUTTON statement. It has the same parameters as the regular push buttons.

The header file DIALOG.H, which is also used by the example program, is shown here:

```
#define IDM_DIALOG 100
#define IDM_EXIT 101
#define IDM_HELP 102
#define IDD_AUTHOR 200
#define IDD_PUBLISHER 201
#define IDD_COPYRIGHT 202
```

Enter this file now.

The Dialog Box Window Function

As stated earlier, events that occur within a dialog box are passed to the window function associated with that dialog box and not to your program's main window function. The following dialog box window function responds to the events that occur within the MyDB dialog box.

```
/* A simple dialog function. */
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message, WPARAM wParam, LPARAM lParam)
{switch(message)
    { case WM_COMMAND:
   switch(LOWORD(wParam))
     {case IDCANCEL: EndDialog(hdwnd, 0); return 1;
      case IDD_COPYRIGHT: MessageBox(hdwnd,"Copyright","Copyright",MB_OK);return 1;
      case IDD_AUTHOR: MessageBox(hdwnd, "Author", "Author", MB_OK); return 1;
      case IDD_PUBLISHER:MessageBox(hdwnd, "Publisher", "Publisher", MB_OK);return1;}
    }return 0;}
```

Each time a control within the dialog box is accessed, a **WM_COMMAND** message is sent to **DialgFunc( ),** and **LOWORD(wParam)** contains the **ID** of the control affected.

DialogFunc( ) processes the four messages that can be generated by the box. If the user presses Cancel, then **IDCANCEL** is sent, causing the dialog box to be closed using a call to the API function **EndDialog( )**. (**IDCANCEL** is a standard ID defined by including WINDOWS.H.) Pressing either of the other three buttons causes a message box to be displayed that confirms the selection. As mentioned, these buttons will be used by later examples to display information from the database.

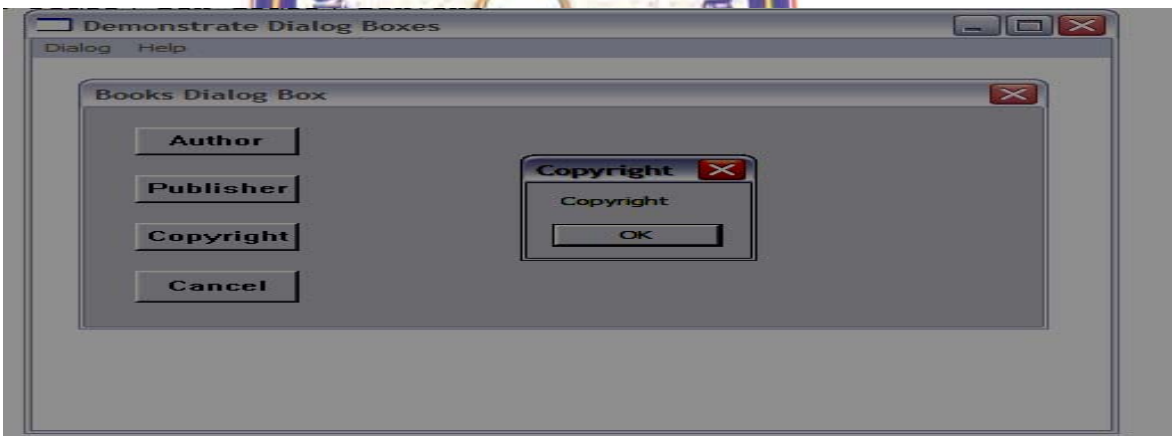
A First Dialog Box Sample Program

Here is the entire dialog box example. When the program begins execution, only the top-level menu is displayed on the menu bar. By selecting Dialog, the user causes the dialog box to be displayed. Once the dialog box is displayed, selecting a push button causes the appropriate response. A sample

screen is shown in Figure 6-2. Notice that the books database is included in this program, but is not used. It will be used by subsequent examples.

```
/* Demonstrate a modal dialog box. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "dialog.h"
#define NUMBOOKS 7
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[ ] = "MyWin"; /* name of window class */  HINSTANCE hInst;
/* books database */
struct booksTag {char title[40];
                 unsigned copyright;
                 char author[40];
                 char publisher[40];
                } books[NUMBOOKS] =
{{"C: The Complete Reference", 1995,"Herbert Schildt", "Osborne/McGraw-Hill"},
{"MFC Programming from the Ground Up", 1996, "Herbert Schildt", "Osborne/McGraw-Hill"},
{"Java: The Complete Reference", 1997, "Naughton and Schildt", "Osborne/McGraw-Hill"},
{"Design and Evolution of C++", 1994, "Bjarne Stroustrup", "Addison-Wesley"},
{"Inside OLE", 1995, "Kraig Brockschmidt", "Microsoft Press"},
{"HTML Sourcebook", 1996, "lan S. Graham", "John Wiley & Sons"},
{"Standard C++ Library", 1995, "P. J. Plauger", "Prentice-Hall"}};
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR IpszArgs, int
nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1; HANDLE hAccel;
wc1.cbSize = sizeof(WNDCLASSEX);wc1.hlnstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon =   LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm  = LoadIcon(NULL, IDI_WINLOGO);
wc1.hCursor  =  LoadCursor(NULL,  IDC_ARROW);  wc1.lpszMenuName  =  "MyMenu";
wc1.ClsExtra=0; wc1.cbWndExtra=0; wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
if(!RegisterClassEx(&wc1)) return 0;
hwnd=CreateWindow(szWinName, "Demonstrate Dialog Boxes",
```

```
wS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, NULL, hThisInst, NULL  );
hInst = hThisInst; /* save the current instance handle */
/* load accelerators */ hAccel = LoadAccelerators (hThisInst, "MyMenu");
/* Display the window. */ ShowWindow(hwnd, nWinMode) ; UpdateWindow(hwnd) ;
while (GetMessage(&msg, NULL, 0, 0))
{ if ( !TranslateAccelerator (hwnd, hAccel, &msg) )
{TranslateMessage(&msg) ; DispatchMes sage ( &msg ); }} return msg.wParam; }
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam) {int response;
switch (message) { case WM_COMMAND:
switch (LOWORD (wParam) ) {
    case IDM_DIALOG: DialogBox(hInst, "MyDB" , hwnd, (DLGPROC) DialogFunc) break;
     case IDM_EXIT: response=MessageBox(hwnd,"Quit the Program?","Exit",MB_YESNO) ;
          if (response == IDYES) PostQuitMessage (0) ; break;
              case IDM_HELP: MessageBox(hwnd, "No Help", "Help", MB_OK) ; break; } break;
case WM_DESTROY: /* terminate the program */ PostQuitMessage (0) ; break;
default: return DefWindowProc (hwnd, message, wParam, IParam);}          return 0; }
/* A simple dialog function. */ BOOL CALLBACK DialogFunc (HWND hdwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{ switch (message) { case WM_COMMAND:
switch (LOWORD (wParam) ) { case IDCANCEL: EndDialog(hdwnd, 0 ) ; return 1 ;
 case IDD_COPYRIGHT: MessageBox (hdwnd, "Copyright", "Copyright", MB_OK); return 1 ;
 case IDD_AUTHOR: MessageBox (hdwnd, "Author", "Author", MB_OK) ;return 1;
 case IDD_PUBLISHER: MessageBox (hdwnd, "Publisher", "Publisher", MB_OK); return 1 ;
                          } }return 0 ;}
```



Notice the global variable **hInst**. This variable is assigned a copy of the current instance handle passed to **WinMain( )**. The reason for this variable is that the dialog box needs access to the current instance handle. However, the dialog box is not created in **WinMain( ).** Instead, it is created in **WindowFunc( ).** Therefore, a copy of the instance parameter must be made so that it can be accessible outside of **WinMain( ).**

Adding a List Box

To continue exploring dialog boxes, let's add another control to the dialog box defined in the previous program. One of the most common controls after the push button is the list box. We will use the list box to display a list of the titles in the database and allow the user to select the one in which he or she is interested. The **L1STBOX** statement has this general form:

LISTBOX LBID, X, Y, Width, Height [, Style]

Here, LBID is the value that identifies the list box. The box's upper left corner will be at X, Y and the box will have the dimensions specified by Width and Height. Style determines the exact nature of the list box. To add a list box, you must change the dialog box definition in DIALOG.RC. First, add this list box description to the dialog box definition:

LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_VISIBLE
| WS_BORDER | WS_VSCROLL | WS_TABSTOP

Second, add this push button to the dialog box definition:

PUSHBUTTON "Select Book", IDD_SELECT, 103, 41, 54, 14, WS_CHILD
| WS_VISIBLE | WS_TABSTOP

After these changes, your dialog box definition should now look like this:

```
MyDB DIALOG 10, 10, 210, 110 CAPTION "Books Dialog Box" STYLE DS_MODALFRAME |
WS_POPUP |WS_CAPTION | WS_SYSMENU
DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14,
WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Copyright" , IDD_COPYRIGHT, 11, 58, 36, 14,
WS_CHILD| WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16,
WS_CHILD| WS_VISIBLE | WS_TABSTOP
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_VISIBLE |WS_VSCROLL |
WS_BORDER |WS_TABSTOP
PUSHBUTTON "Select Book", IDD_SELECT, 103, 41, 54, 14,
```

WS_CHILD | WS_VISIBLE | WS_TABSTOP}

You will also need to add these macros to DIALOG. H:

#define IDD_LB1 203

#define IDD_SELECT 204

IDD_LB1 identifies the list box specified in the dialog box definition in the resource file. **IDD_SELECT** is the ID value of the Select Book push button.

List Box Basics

When using a list box, you must perform two basic-operations. First, you must initialize the list box when the dialog box is first displayed. This consists of sending the list box the list that it will display. (By default, the list box will be empty.) Second, once the list box has been initialized, your program will need to respond to the user selecting an item from the list. List boxes generate various types of notification messages. A notification message describes what type of control event has occurred. (Several of the standard controls generate notification messages.) For the list box used in the following example, the only notification message we will use is **LBN_DBLCLK**. This message is sent when the user has double-clicked on an entry in the list. This message is contained in **HIWORD(wParam)** each time a **WM_COMMAND** is generated for the list box. (The list box must have the **LBS_NOTIFY** style flag included in its definition in order to generate **LBN_DBLCLK** messages.) Once a selection has been made, you will need to query the list box to find out which item has been selected, A list box is a control that receives messages as well as generating them. You can send a list box several different messages. To send a message to the list box (or any other control) use the **SendDlgItemMessage( )** API function. Its prototype is shown here: **LONG SendDlgItcmMessage (HWND Hwnd , int ID , UINT IDMsg , WPARAM wParam , LPARAM lParam);**

SendDlgItemMessage() sends the message specified by IDMsg to the control (within the dialog box) whose ID is specified by ID. The handle of the dialog box is specified in hdwnd. Any additional information required f the message is specified in wParam and lParam. The additional informal, if any, varies from message to message. If there is no additional information to pass to a control, the wParam and the lParam arguments should be zero. The value returned by **SendDlgItemMessage( )** contains the information requested by IDMsg.

Macro	Purpose
LB_ADDSTRING	Adds a string (selection) to the list box.
LB_GETCURSEL	Requests the index of the selected item.
LB_SETCURSEL	Selects an item.
LB-FINDSTRING	Finds a matching entry.
LB_SELECTSTR1NG	Finds a matching entry and selects It.
LB_GE'ITEXT	Obtains the text associated with an Item

Here are a few of the most common messages that you can send to a list box. Let's take a closer look at these messages.

LB_ADDSTRING adds a string to the list box. That is, the specified string becomes another selection within the box. The string must be pointed to by **lParam**.*(wParam is unused by this message.)* The value returned by list box is the index of the string in the list. If an error occurs, **LB_ERR** is returned.

The **LB_GETCURSEL** message causes the list box to return the index of the currently selected item. All list box indexes begin with zero. *Both lParam and wParam are unused.* If an error occurs, **LB_ERR** is returned. If no item is currently selected, then an error results.

You can set the current selection inside a list box using the **LB_SETCURSEL** command. For this message, *wParam specifies the index of the item to select. lParam is not used.* On error, **LB_ERR** is returned.

You can find an item in the list that matches a specified prefix using **LB_FINDSTRING**. That is, **LB_FINDSTRING** attempts to match a partial string with an entry in the list box. *wParam specifies the index at which point the search begins and lParam points to the string that will be matched. If a match is found, the index of the matching item is returned*. Otherwise, **LB_ERR** is returned. **LB_FINDSTRING** does not select the item within the list box.

If you want to find a matching item and select it, use **LB_SELECTSTRING**. It takes the same parameters as **LB_FINDSTRING** but also selects the matching item.

You can obtain the text associated with an item in a list box using **LB_GETTEXT**. In this case, *wParam specifies the index of the item and lParam points to the character array that will receive the null terminated string associated with that index*. The length of the string is returned if successful. **LB_ERR** is returned on failure.

Initializing the List Box

As mentioned, when a list box is created, it is empty. This means that you will need to initialize it each time the dialog box that contains it is displayed. This is easy to accomplish because each time a dialog box is activated, its window function is sent a **WM_INITDIALOG** message. Therefore, you will need to add this case to the outer switch statement in **DialogFunc( ).**

```
case WM_INITDIALOG: /* initialize list box */
  for(i=0; i<NUMBOOKS; i + +)
    SendDlgItemMessage(hdwnd, IDD_LB1,LB_ADDSTRING, 0, (LPARAM)books[i].title);
 /*select first item*/SendDlgItemMessage(hdwnd, IDD_LB1,LB_SETCURSEL, 0, 0);return 1;
```

This code loads the list box with the titles of **books** as defined in the books array. Each string is added to the list box by calling **SendDlgItemMessage( )** with the **LB_ADDSTRING** message. The string to add is pointed to by the *lParam* parameter. (The type cast to **LPARAM** is necessary in this case to convert a pointer into a unsigned integer.) In this example, each string is added to the list box in the order it is sent. (However, depending on how you construct the list box, it is possible to have the items displayed in alphabetical order.) If the number of items you send to a list box exceeds what it can display in its window, vertical scroll bars will be added automatically.

This code also selects the first item in the list box. When a list box is first created, no item is selected. While this might be desirable under certain circumstances, it is not in this case. Most often, you will want to automatically select the first item in a list box as a convenience to the user.

Processing a Selection

After the list box has been initialized, it is ready for use. There are essentially two ways a user makes a selection from a list box. First, the user may double-click on an item. This causes a **WM_COMMAND** message to be passed to the dialog box's window function. In this case, **LOWORD(wParam)** contains the ID associated with the list box and **HIWORD(wParam)** contains the **LBN_DBLCLK** notification message. Double-clicking causes your program to be immediately aware of the user's selection. The other way to use a list box is to simply highlight a selection (either by single-clicking or by using the array keys to move the highlight). The list box remembers the selection and waits until your program requests it. Both methods will be demonstrated in the example program.

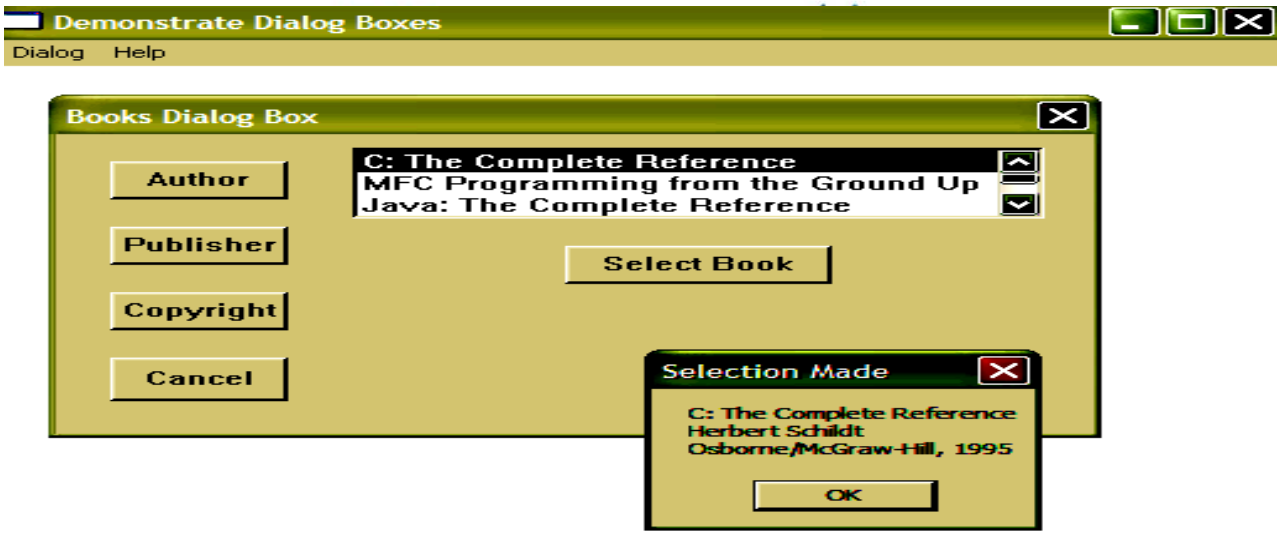
Once an item has been selected in a list box, you determine which item was chosen by sending the **LB_GETCURSEL** message to the list box. The list box then returns the index of the selected item. Remember, if this message is sent before an item has been selected, the list box returns **LB_ERR**. (This is one reason that it is a good idea to select a list box item when it is initialized.)

To process a list box selection, add these cases to the inner switch inside DialogFunc( ). You will also need to declare a long integer called i and a character array called str inside DialogFunc( ). Your dialog box will now look like that shown in Figure 7-3. Each time a selection is made because of a double-click or when the user presses the "Select Book" push button, the currently selected book has its information displayed.

```
case IDD_LB1: /* process a list box LBN_DBLCLK */
  /* see if user made a selection */
    if(HIWORD(wParam)==LBN_DBLCLK)
      { i = SendDlgItemMessage(hdwnd, IDD_LBl, LB_GETCURSEL, 0, 0);  /* get index */
```

```
sprintftstr, "%s\n%s\n%s, %u", books[i] .title, books[i].author, books[i].publisher, books[i].copyright);
MessageBox(hdwnd, str, "Selection Made", MB_OK);
/* get string associated with that index */
SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT, i, (LPARAM) str); } return 1;
case IDD_SELECT: /* Select Book button has been pressed */
i=SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0); /* get index */
sprintf(str, "%s\n%s\n%s, %u", books[i].title, books[i].author, books[i].publisher, books[i].copyright);
MessageBox(hdwnd, str, "Selection Made", MB_OK);
/* get string associated with that index */
SendDlgItemMessage (hdwnd, IDD_LB1, LB_GETTEXT, i, (LPARAM) str); return 1;
```

Notice the code under the **IDD_LB1** case. Since the list box can generate several different types of notification messages, it is necessary to examine the high-order word of **wParam** to determine if the user double-clicked on an item. That is, just because the control generates a notification message does not mean it is a double-click message. (You will want to explore the other list box notification messages on your own.)



Adding an Edit Box

In this section we will add an edit control to the dialog box. Edit boxes are particularly useful because they allow users to enter a string of their own choosing. The edit box in this example will be used to allow the user to enter the title (or part of a title) of a book. If the title is in the list, then it will t selected and information about the book can be obtained. Although the addition of an edit box

enhances our simple database application, it also serves another purpose. It will illustrate how two controls can work together.

Before you can use an edit box, you must define one in your resource file for this example, change **MyDB** so that it looks like this:

```
MyDB DIALOG 10, 10, 210, 110
CAPTION "Books Dialog Box"
STYLE DS_MODALFRAME | WS_POPUP |WS_CAPTION | WS_SYSMENU
{DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16,
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_VISIBLE|
    WS_BORDER |WS_VSCROLL | WS_TABSTOP
PUSHBUTTON "Select Book", IDD_SELECT, 103, 41, 54, 14,
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
EDITTEXT IDD_EB1, 65, 73, 130, 12, ES_LEFT | WS_VISIBLE WS_BORDER |
    ES_AUTOHSCROLL | WS_TAB5TOP
PUSHBUTTON "Title Search", IDD_DONE, 107, 91, 46, 14, WS_CHILD |
    WS_VISIBLE | WS_TABSTOP}
```

This version adds a push button called Title Search which will be used to tell the program that you entered the title of a book into the edit box. It also adds the edit box itself. The ID for the edit box is **IDD_EB1**. This definition causes a standard edit box to be created.

The **EDITTEXT** statement has this general form: EDITTEXT EDID, X, Y, Width, Height [.Style]

Here, EDID is the value that identifies the edit box. The box's upper left corner will be at X, Y and its dimensions are specified by Width and Height. Style determines the exact nature of the list box.

You must also add these macro definitions to DIALOG.H:

```
#define IDD_EB1 205
#define IDD_DONE 206
```

Edit boxes recognize many messages and generate several of their own. However, for the purposes of this example, there is no need for the program to respond to any messages. As you will see, edit

boxes perform the editing function on their own, independently. No program interaction is required. Your program simply decides when it wants to obtain the current contents of the edit box.

To obtain the current contents of the edit box, use the API function **GetDlgItemText( )**. It has this prototype: **UINT GetDlgItemText(HWND hdwnd, int ID, LPSTR lpstr, int Max);**

This function causes the edit box to copy the current contents of the box to the string pointed to by lpstr. The handle of the dialog box is specified by hdwnd. The ID of the edit box is specified by ID. The maximum number of characters to copy is specified by Max. The function returns the length of the string..

Although not required by all applications, it is possible to initialize the contents of an edit box using the SetDlgItemText( ) function. Its prototype is shown here:

BOOL SetDlgItemText(HWND hdwnd, int ID, LPSTR lpstr);

This function sets the contents of the edit box to the string pointed to by lpstr. The handle of the dialog box is specified by hdwnd. The ID of the edit box is specified by ID. The function returns nonzero if successful or zero on failure.

To add an edit box to the sample program, add this case statement to the inner switch of the **DialogFunc( )** function. Each time the Title Search button is pressed, the list box is searched for a title that matches the string that is currently in the edit box. If a match is found, then that title is selected in the list box. Remember that you only need to enter the first few characters of the title. The list box will automatically attempt to match them with a title.

```
case IDD_DONE: /* Title Search button pressed */
 /* get current contents of edit box */ GetDlgItemText (hdwnd, IDD_EB1, str, 80);
/* find a matching string in the list box */
i = SendDlgItemMessage( hdwnd, IDD_LB1, LB_FINDSTRING, 0, (LPARAM) str) ;
if(i != LB_ERR) { /* if match is found */
 /*select the matching title in list box*/
 SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, i, 0);
 /* get string associated with that index */
 SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT, i, (LPARAM) str);
 /* update text in edit box */ SetDlgItemText (hdwnd, IDD_EB1, str) ;
else MessageBox (hdwnd, str, "No Title Matching", MB_OK) ; return 1 ;
```

This code obtains the current contents of the edit box and looks for a match with the strings inside the list box. If it finds one, it selects the matching item in the list box and then copies the string from the list box back into the edit box. In this way, the two controls work together, complementing

each other. As you become a more experienced Windows NT programmer, you will find that there are often instances in which two or more controls can work together.

You will also need to add this line of code to the INITDIALOG case. It causes the edit box to be initialized each time the dialog box is activated.

/* initialize the edit box */ SetDlgItemText(hdwnd, IDD_EB1, books[0].title);

In addition to these changes, the code that processes the list box will be enhanced so that it automatically copies the name of the book selected in the list box into the edit box. These changes are reflected in the full program listing that follows. You should have no trouble understanding them.

The Entire Modal Dialog Box Program

The entire modal dialog box sample program that includes push buttons, a list box, and an edit box, is shown here. Notice that the code associated with the push buttons now displays information about the title currently selected in the list box.

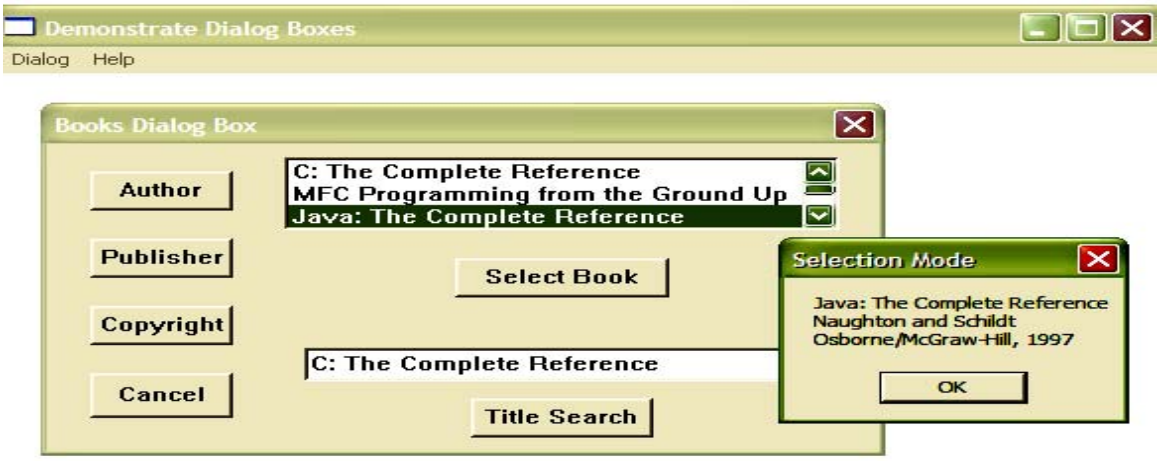
```
/*  A Complete model dialog box example. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "dialog.h"
#define NUMBOOKS 7
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[ ] = "MyWin"; /* name of window class */
HINSTANCE hInst;
/* books database */ struct booksTag {char title[40];
                                      unsigned copyright;
                                      char author[40];
                                      char publisher[40]; } books[NUMBOOKS] =
{{"C: The Complete Reference", 1995,"Herbert Schildt", "Osborne/McGraw-Hill"},
{"MFC Programming from the Ground Up", 1996, "Herbert Schildt", "Osborne/McGraw-Hill"},
{"Java: The Complete Reference", 1997, "Naughton and Schildt", "Osborne/McGraw-Hill"},
{"Design and Evolution of C++", 1994, "Bjarne Stroustrup", "Addison-Wesley"},
{"Inside OLE", 1995, "Kraig Brockschmidt", "Microsoft Press"},
{"HTML Sourcebook", 1996, "lan S. Graham", "John Wiley & Sons"},
{"Standard C++ Library", 1995, "P. J. Plauger", "Prentice-Hall"}};
```

```
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR IpszArgs, int
nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1; HANDLE  hAccel;
wc1.cbSize = sizeof(WNDCLASSEX);wc1.hInstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon =   LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm  = LoadIcon(NULL, IDI_WINLOGO);
wc1.hCursor   = LoadCursor(NULL, IDC_ARROW); wc1.lpszMenuName = "MyMenu";
wc1.ClsExtra=0;wc1.cbWndExtra=0;wc1.hbrBackground = GetStockObject(WHITE_BRUSH);
/* Register the window class. */ if(!RegisterClassEx(&wc1)) return 0;
hwnd = CreateWindow(szWinName, "Demonstrate Dialog Boxes",
WS_OVERLAPPEDWINDOW,  CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT, HWND_DESKTOP, NULL, hThisInst, NULL );
hInst = hThisInst; /* save the current instance handle */
/* load accelerators */ hAccel = LoadAccelerators (hThisInst, "MyMenu");
/* Display the window. */ ShowWindow(hwnd, nWinMode) ; UpdateWindow(hwnd) ;
while (GetMessage(&msg, NULL, 0, 0))
{ if ( !TranslateAccelerator (hwnd, hAccel, &msg) ) {TranslateMessage(&msg) ;
                                                    DispatchMes sage(&msg );}}return msg.wParam ;}
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message,
                                                    WPARAM wParam, LPARAM lParam) { int response;
switch (message)
{case WM_COMMAND:
      swi tch ( LOWORD { wParam) )
         {case IDM_DIALOG: DialogBox (hlnst, "MyDB", hwnd, (DLGPROC) DialogFunc) break;
          Case IDM_EXIT:response=MessageBox(hwnd,"Quit the Program?","Exit", MB_YESNO)
;                                                    if (response == IDYES) PostQuitMessage (0) ; break;
          case I DM HELP: MessageBox(hwnd, "No Help", "Help", MB_OK); break;} break;
case WM_DESTROY: /* terminate the program */ PostQuitMessage(0); break;
default :return DefWindowProc(hwnd, message, wParam, lParam); } return 0;}
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message, WPARAM wParam, LPARAM
lParam){ long  i; char str[255];
switch(message) { case WM_COMMAND:
                        switch(LOWORD(wParam)){
                           case ID_CANCEL: EndDialog(hdwnd, 0); return 1;
```

```
                    case IDD_COPYRIGHT:
                /*get index*/    i = SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETCURSEL,0, 0);
                            sprintf(str, "%u", books [  i ] .copyright);
                                MessageBox(hdwnd,str,"Copyright",MB_OK); return 1;
        case IDD_AUTHOR:
                i = SendDlgItemMessage(hdwnd, IDD_LB1,LB_GETCURSEL, 0, 0);  /*get index*/
                 sprintf(str, "%s", books[i].author);
                  MessageBox(hdwnd, str, "Author", MB_OK); return 1;
      case IDD_PUBLISHER:
                i = SendDlgItemMessage(hdwnd, IDD_LB1,LB_GETCURSEL, 0, 0);  /*get index*/
               sprintf(str, "%s", books[i].publisher);
                MessageBox(hdwnd, str, "Publisher", MB_OK); return 1;
      case IDD_DONE: /* Title Search button pressed */
/* get current contents of edit box */ GetDlgItemText(hdwnd, IDD_EB1, str, 80);
/*find a matching string in the list box*/
                    i=SendDlgItemMessage(hdwnd,IDD_LB1,LB_FINDSTRING,0,(LPARAM) str);
                      if(i != LB_ERR) { /* if match is found */
                       /*select the matching title in list box*/
                        SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, i, 0)
                       /* get string associated with that index */
                    SendDlgItemMessage(hdwnd, IDD_LB1,LB_GETTEXT, i, (LPARAM) str);
     /* update edit box */ SetDlgItemText(hdwnd, IDD_EB1, str); }
 else MessageBox(hdwnd, str, "No Title Matching", MB_OK);  return 1;
case IDD_LB1: /* process a list box LBN_DBLCLK */
   /* see  if user made a selection */
  if(HIWORD(wParam)==LBN_DBLCLK) {
i= SendDlgItemMessage (hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0); /* get index */
sprintf(str,"%s\n%s\n%s,%u",books[i].title,books[i].author, books[i].publisher, books[i].copyright);
MessageBox(hdwnd, str, "Selection Mode", MB_OK);
/*get string associated with that index*/
  SendDlgItemMessage(hdwnd,IDD_LB1,LB_GETTEXT,i, (LPARAM)str);
    /* update edit box */ SetDlgItemText(hdwnd, IDD_EB1, str); } return 1;
case IDD_SELECT: /* Select book button has been pressed */
```

```
  i= SendDlgItemMessage (hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0); /* get index */
sprintf(str,"%s\n%s\n%s,%u",books[i].title,books[i].author, books[i].publisher, books[i].copyright);
MessageBox(hdwnd, str, "Selection Mode", MB_OK);
/*get string associated with that index*/
  SendDlgItemMessage(hdwnd,IDD_LB1,LB_GETTEXT,i, (LPARAM) str);
   /* update edit box */ SetDlgItemText(hdwnd, IDD_EB1, str); } return 1;} break;
case WM_INITDIALOG: /* initialize list box */
   for ( i =0; i<NUMBOOKS; i + + );
SendDlgItemMessage(hdwnd, IDD_LB1, LB_ADDSTRING, 0, (LPARAM)books[i].title);
/* select first item */ SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, 0, 0);
/* initialize the edit box */ SetDlgItemText(hdwnd, IDD_EB1, books[0].title);
return 1;}  return 0;}
```

Figure 6-2 shows sample output created by the complete modal dialog box program.



Using a Modeless Dialog Box

To conclude this lecture, the modal dialog box used by the preceding program will be converted into a modeless dialog box. As you will see, using a modeless dialog box requires a little more work. The main reason for this is that a modeless dialog box is more independent than a modal dialog box. Specifically, the rest of your program is still active when a modeless dialog box is displayed. Also, both it and your application's window function continue to receive messages. Thus, some additional overhead is required in your application's message loop to accommodate the modeless dialog box.

To create a modeless dialog box, you do not use **DialogBox( )**. Instead, you must use the CreateDialog( ) API function. Its prototype is shown here: **HWND CreateDialog (HINSTANCE hTlnsInst, LPCSTR lpName, HWND hwnd, DLGPROC lp.DFunc)**

Here, hThisInst is a handle to the current application that is passed to your program in the instance parameter to **WinMain()**. The name of the dialog box as defined in the resource file is pointed to by lpName. The handle to the owner of the dialog box is passed in hwnd. (This is typically the handle to the window that calls **CreateDialog( )**.) The lpDFunc parameter contains a pointer to the dialog function. The dialog function is of the same type as that used for a modal dialog box. **CreateDialog( )** returns a handle to the dialog box. If the dialog box cannot be created, **NULL** is returned.

Unlike a modal dialog box, a modeless dialog box is not automatically visible, so you may need to call **ShowWindow( )** to cause it to be displayed after it has been created. However, if you add **WS_VISIBLE** to the dialog box's definition in its resource file, then it will be visible automatically.

To close a modeless dialog box your program must call **DestroyWindow( )** rather than **EndDialog()**. The prototype for DestroyWindow() is shown here:

BOOL DestroyWindow(HWND hwnd);

Here, hwnd is the handle to the window (in this case, dialog box) being closed. The function returns nonzero if successful and zero on failure.

Since your application's window function will continue receiving messages while a modeless dialog box is active, you must make a change to your program's message loop. Specifically, you must add a call to **IsDialogMessage( ).** This function routes dialog box messages to your modeless dialog box. It has this prototype:

BOOL IsDialogMessage(HWND hdwnd, LPMSG msg)

IN DEPTH: Disabling a Control: Sometimes you will have a control that is not applicable to all situations. When a control is not applicable it can be (and should be) disabled. A control that is disabled is displayed in gray and may not be selected. To disable a control, use the **EnableWindow()** API function, shown here:**BOOL EnableWindow(HWND hCntl,BOOL How);**

Here, hCntl specifies the handle of the window to be affected. (Remember, controls are simply specialized windows.) If How is nonzero, then the control is enabled. That is, it is activated. If How is zero, the control is disabled. The function returns nonzero if the control was already disabled. It returns zero if the control was previously enabled. To obtain the handle of a control, use the **GetDlgItem( )** API function. It is shown here: **HWND GetDlgItem(HWND hDwnd, int ID);**

Here, hDwnd is the handle of the dialog box that owns the control. The control ID is passed in ID. This is the value that you associate with the control in its resource file. The function returns the handle of the specified control or NULL on failure.

To see how you can use these functions to disable a control, the following fragment disables the Author push button. In this example **hwpb** is a handle of type **HWND**.

```
hwpb = GetDlgItem(hdwnd,  IDD_AUTHOR); /*get handle of button */
EnableWindow(hwpb, 0); /* disable it*/
```

On your own, you might want to try disabling and enabling the other controls used by the examples in this and later lectures.

Here, hdwnd is the handle of the modeless dialog box and msg is the message obtained from GetMessage() within your program's message loop. The function returns nonzero if the message is for the dialog box. It returns zero otherwise. If the message is for the dialog box, then it is automatically passed to the dialog box function. Therefore, to process modeless dialog box messages, your program's message loop must look something like this:

```
while (GetMessage(&msg, NULL, 0, 0)){
 if ( ! IsDialogMessage (hDlg, &msg) ) { /*not dialog box message*/
    if ( !TranslateAccelerator (hwnd, hAccel, &msg)) {
        TranslateMessage (&msg) ; /* translate keyboard message */
         DispatchMessage(&msg) ; /* return control to Windows */ }}}
```

As you can see, the message is processed by the rest of the message loop only if it is not a dialog box message.

Creating a Modeless Dialog Box

To convert the modal dialog box shown in the preceding example into a modeless one, surprisingly few changes are needed. The first change that you need to make is to the dialog box definition in the DIALOG. RC resource file. Since a modeless dialog box is not automatically visible, add **WS_VISIBLE** to the dialog box definition. Also, although not technically necessary, you can remove the **DS_MODALFRAME** style, if you like. Since we have made several changes to DIALOG. RC since the start of the chapter, its final form is shown here after making these adjustments.

```
; Sample dialog box and menu resource file.
#include <windows.h>
#include "dialog. h"
MyMenu MENU {
POPUP "&Dialog {MENUITEM "&Dialog\tF2", IDM_DIALOG
                MENUITEM &Exit\tF3", IDM_EXIT}
MENUITEM "&Help", IDM_HELP
MyMenu ACCELERATORS
        {VK_F2, IDM_DIALOG, VIRTKEY
         VK_F3, IDM_EXIT, VIRTKEY
```

```
        VK_F1, IDM_HELP, VIRTKEY}
MyDB DIALOG 10, 10, 210, 110 CAPTION  "Books Dialog Box"
STYLE WS_POPUP | WS_CAPTION |WS_SYSMENU | WS_VISIBLE
{DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
WS_CHILD | WS_VISIBLE | WS_TABSTOP
 PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14,
        WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14,
WS_CHILD | WS_VISIBLE | WS_TABSTOP
PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16,
WS_CHILD |WS_VISIBLE | WS_TABSTOP
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS BORDER |WS_VISIBLE |
        WS_VSCROLL | WS_TABSTOP
PUSHBUTTON "Select Book", IDD_SELECT, 103, 41, 54, 14,
WS_CHILD | WS_VISIBLE | WS_TABSTOP
EDITTEXT IDD_EB1, 65, 73, 130, 12, ES_LEFT | WS_VISIBLE |WS_BORDER |
        ES_AUTOHSCROLL | WS_TABSTOP
PUSHBUTTON "Title Search", IDD_DONE, 107, 91, 46, 14,
        WS CHILD |I WS_VISIBLE | WS_TABSTOP }
```

Next, you must make the following changes to the program:

1. Create a global handle called **hDlg**.
2. Add **IsDiaLogMessage( )** to the message loop.
3. Create the dialog box using **CreateDialog( )** rather than **DialogBox( ).**
4. Close the dialog box using **DestroyWindow( )** instead of **EndDialog( ).**

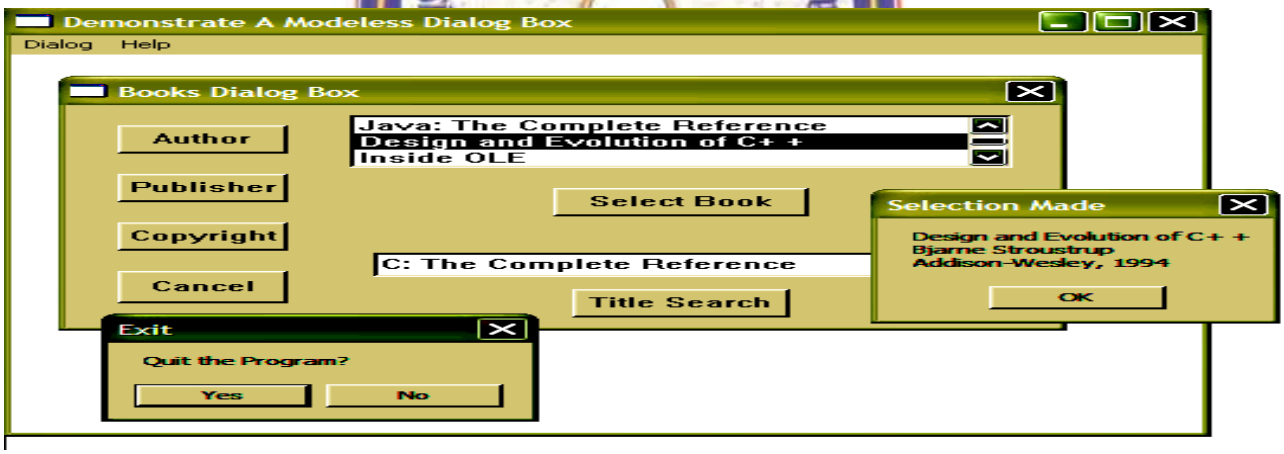
The entire listing (which incorporates these changes) for the modeless dialog box example is shown here. Sample output from this program is shown in Figure 7-4. (You should try this program on your own to fully understand the difference between modal and modeless dialog boxes.)

```
/* A modeless dialog box example. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "dialog.h"
#define NUKBOOKS 7
LRESULT CALLBACK WindowFunc (HWND, UINT, WPARAM, LPARAM) ;
```

```
BOOL CALLBACK DialogFunc (HWND, UINT, WPARAM, LPARAM) ;
char szWinName [ ] = "MyWin" ; /* name of window class */ H INSTANCE hInst;
HWND hDlg =0; /* dialog box handle */
/* books database */
struct booksTag { char title [40] ;
                  unsigned copyright;
                  char author[40];
                  char publisher [40] ; }
books[NUMBOOKS] = {
{"C: The Complete Reference", 1995,"Hebert Schildt", "Osborne/McGraw-Hill"},
{ "MFC Programming from the Ground Up", 1996, "Herbert Schildt", "Osborne/McGraw-Hill" },
{"Java:The Complete Reference",1997,"Naughton and Schildt","Osborne/McGraw-Hill " },
{"Design and Evolution of C+ + ", 1994", "Bjarne Stroustrup", "Addison-Wesley" },
{ Inside OLE" , 1995, "Kraig Brockschmidt" , "Microsoft Press" },
{"HTML Sourcebook", 1996, "lan S. Graham", "John Wiley & Sons" },
{"Standard C++ Library", 1995, "P. J. Plauger", "Prentice-Hall" } };
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR IpszArgs, int nWinMode) {HWND hwnd;  MSG rnsg;
WNDCLASSEX wcl;  HANDLE hAccel;
wc1.cbSize = sizeof(WNDCLASSEX);wc1.hInstance = hThisInst;
wc1.lpszClassName = szWinName; wc1.lpfnWndProc = WindowFunc;
wc1.style =0; wc1.hIcon =     LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm= LoadIcon(NULL, IDI_WINLOGO);
wc1.hCursor    = LoadCursor(NULL, IDC_ARROW);lpszMenuName = "MyMenu";
wc1.cbClsExtra = 0; wc1.cbWndExtra = 0;
wc1.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
/* Register the window class. */ if(!RegisterClassEx(&wc1)) return 0;
/* Now that a window class has been registered, a window can be created. */
hwnd=CreateWindow(szWinName,"Demonstrate A Modeless Dialog Box",
WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,HWND_DESKTOP,
NULL, hThisInst,  NULL);   hInst = hThisInst; /* save the current instance handle */
  /* load accelerators */ hAccel = LoadAccelerators (hThisInst, "MyMenu");
/* Display the window. */ ShowWindiow(hwnd, nWinMode) ; UpdateWindow(hwnd) ;
```

```
 while (GetMessage(&msg, NULL, 0, 0))
  {if ( ! IsDialogMessage (hDlg, &msg) )
     {/* is not a dialog message*/ if ( !TranslateAccelerator (hwnd, hAccel, &msg) )
     {TranslateMessage(&msg); DispatchMessage(&.msg);  } return msg.wParam; }
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
              WPARAM wParam, LPARAM lParam) { int response;
switch(message) { case WM_COMMAND:
  switch(LOWORD(wParam))
  {case IDM_DIALOG: hDlg = CreateDialog(hlnst, "MyDB", hwnd, (DLGPROC)
DialogFunc);break;
   case IDM_EXIT: response = MessageBox(hwnd, "Quit the Program?", "Exit", MB_YESNO);
                  if(response == IDYES) PostQuitMessage(0); break;
   case IDM_HELP: MessageBox(hwnd, "No Help", "Help", MB__OK) ; break; } break;
   case WM_DESTROY: /* terminate the program */ PostQuitMessage(0); break;
  default: return DefWindowProc(hwnd, message, wParam, lParam); } return 0;}
/* A simple dialog function. */
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message, WPARAM wParam, LPARAM
lParam) { long i; char str[255];
switch(message)
{case WM_COMMAND:
   switch(LOWORD(wParam))
    {case IDCANCEL: DestroyWindow(hdwnd); return 1;
     case IDD_COPYRIGHT:
        i = SendDlgltemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0);
       sprintf(str,"%u", books[i].copyright);
       MessageBox(hdwnd, str, "Copyright", MB_OK); return 1;
     case IDD_AUTHOR: i = SendDlgltemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0);
                      sprintf(str, "%s", books[i].author);
                      MessageBox(hdwnd, str, "Author", MB_OK); return 1;
  case IDD_PUBLISHER: i=SendDlgltemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0);
                      sprintftstr, "%s", books[i].publisher);
                      MessageBox(hdwnd, str, "Publisher", MB_OK); return 1;
 case IDD_DONE:/*get current contents of edit box*/ GetDlgItemText(hdwnd, IDD_EB1, str, 80);
       i=SendDlgItemMessage(hdwnd, IDD_LB1, LB_FINDSTRING, 0, (LPARAM) str);
```

```
        if(i != LB_ERR) /* if match is found */
                {SendDlgItemMessage(hdwnd,IDD_LB1, LB_SETCURSEL, i,0);
                SendDlgItemMessage(hdwnd,IDD_LB1,LB_GETTEXT, i, (LPARAM) str);
                        /*update text in edit box*/ SetDlgItemText(hdwnd, IDD_EB1,str);
                    else MessageBox(hdwnd, str, "No Title Matching", MB_OK); return 1;
      case IDD_LB1: /* process a list box LBN_DBLCLK */
        if(HIWORD(wParam)==LBN_DBLCLK) /* see  if user made a selection */
         {i=SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0);  /*get index*/
   sprintf(str,        "%s\n%s\n%s,        %u",books[i].title,        books[i].author,books[i].publisher,
books(i].copyright);
   MessageBox(hdwnd, str, "Selection Made", MB_OK);
        SendDlgItemMessage(hdwnd,IDD_LB1,LB_GETTEXT,i,(LPARAM)str);
   /*update edit box*/ SetDlgItemText(hdwnd, IDD_EBl, str); return 1;
  case IDD_SELECT: /* Select Book button has been pressed */
i = SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETCURSEL, 0, 0);  /* get index */
sprintf (str, "%s\n%s\n%s, %u", "books[i].title, books[i].author, books[i].publisher,
books[i].copyright);
MessageBox(hdwnd, str, "Selection Made", MB_OK);
/*get string associated with that index*/
SendDlgItemMessage(hdwnd, IDD_LBl, LB_GETTEXT, i, (LPARAM) str);
/* update edit box */ SetDlgItemText(hdwnd, IDD_EB1, str); return 1;} break;
case WM_INITDIALOG: /* initialize list box */
    for(i = 0; i<NUMBOOKS; i + + )
     SendDlgItemMessage(hdwnd, IDD_LB1,LB_ADDSTRING, 0, (LPARAM)books[i].title);
    /* select first item */ SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, 0, 0);
   /*initialize the edit box*/SetDlgItemText(hdwnd, IDD_EB1, books[0] .title);return 1;}return 0;}
```



ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Seven:

More Control (Dialog Box)

CONTENTS:

This lecture begins with a discussion of the scroll bar and illustrates its use in a short example program. Although scroll bars offer a bit more of a programming challenge than do the other standard controls, they are still quite easy to use. Next, check boxes and radio buttons are discussed. To illustrate the practical use of scroll bars, check boxes, and radio buttons, a simple countdown timer application is developed. You could use such a program as a darkroom timer, for example. In the process of developing the countdown timer, Windows NT timer interrupts and the **WM_TIMER** message are explored. The chapter concludes with a look at Windows static controls.

Scroll Bars

The scroll bar is one of Windows NT's most important controls. Scroll bars exist in two forms. The first type of scroll bar is an integral part of a normal window or dialog box. These are called standard scroll bars. The other type of scroll bar exists separately as a control and is called a scroll bar control. Both types of scroll bars are managed in much the same way.

Activating the Standard Scroll Bars

For a window to include standard scroll bars, you must explicitly request it. For windows created using **CreateWindow( )**, such as your application's main window, you do this by including the styles **WS_VSCROLL** and/or **WS_HSCROLL** in the style parameter. In the case of a dialog box, you include the **WS_VSCROLL** and/or **WS_HSCROLL** styles in the dialog box's definition inside its resource file. As expected, the **WS_VSCROLL** causes a standard vertical scroll bar to be included and **WS_HSCROLL** activates a horizontal scroll bar. After you have added these styles, the window will automatically display the standard vertical and horizontal scroll bars.

Receiving Scroll Bar Messages

Unlike other controls, a scroll bar control does not generate a **WM_COMMAND** message. Instead, scroll bars send either a **WM_VSCROLL** or a **WM_HSCROLL** message when either a vertical or horizontal scroll bar is accessed, respectively. The value of the low-order word of **wParam** contains a code that describes the activity. For the standard window scroll bars, **lParam** is zero. However, if a scroll bar control generates the message, then **lParam** contains its handle.

As mentioned, the value in **LOWORD(wParam)** specifies what type of scroll bar action has taken place. Here are some common scroll bar values:

SB_LINEUP	SB_LINEDOWN
SB_PAGEUP	SB_PAGEDOWN
SB_LINELEFT	SB_LINERIGHT
SB_PAGELEFT	SB_PAGERIGHT
SB_THUMBPOSITION	SB_THUMBTRACK

For vertical scroll bars, each time the user moves the scroll bar up one position, **SB_LINEUP** is sent. Each time the scroll bar is moved down one position, **SB_LINEDOWN** is sent. **SB_PAGEUP** and **SB_PAGEDOWN** are sent when the scroll bar is moved up or down one page. For horizontal scroll bars, each time the user moves the scroll bar left one position, **SB_LINELEFT** is sent. Each time the scroll bar is moved right one position, **SB_LINERIGHT** is sent. **SB_PAGELEFT** and **SB_PAGERIGHT** are sent when the scroll bar is moved left or right one page.

For both types of scroll bars, the **SB_THUMBPOSITION** value is sent after the slider box (thumb) of-the scroll bar has been dragged to a new position. The **SB_THUMBTRACK** message is also sent when the thumb is dragged to a new position. However, it is sent each time the thumb passes over a new position. This allows you to "track" the movement of the thumb before it is released.

When **SB_THUMBPOSITION** or **SB_THUMBTRACK** is received, the high-order word of wParam contains the current slider box position.

SetScrollInfo( ) and GetScrollInfo( )

Scroll bars are, for the most part, manually managed controls. This means that in addition to responding to scroll bar messages, your program will also need to update various attributes associated with a scroll bar. For example, your program must update the position of the slider box manually. Windows NT contains two functions that help you manage scroll bars.

The first is **SetScrollInfo( )**, which is used to set various attributes associated with a scroll bar. Its prototype is shown here:

int SetScrollInfo(HWND hwnd, int which, LPSCROLLINFO lpSI, BOOL repaint);

Here, hwnd is the handle that identifies the scroll bar. For window scroll bars, this is the handle of the window that owns the scroll bar. For scroll bar controls, this is the handle of the scroll bar itself. The value of which determines which scroll bar is affected. If you arc setting the attributes of the vertical window scroll bar, then this parameter must be **SB_VERT**. If you are setting the attributes of the horizontal window scroll bar, this value

must be **SB_HORZ**. However, to set a scroll bar control, this value must be **SB_CTL** and hwnd must be the handle of the control. The attributes are set according to the information pointed to by lpSI (discussed shortly). If repaint is true, then the scroll bar is redrawn. If false, the bar is not redisplayed. The function returns the position of the slider box.

To obtain the attributes associated with a scroll bar, use GctScrolIIInfo(), shown here:

BOOL GetScrollInfo(HWND hwnd, int which, LPSCROLLINFO lpSI);

The hwnd and which parameters are the same as those just described for **SetScrollInfo( ).** The information obtained by **GetScrollInfo( )** is put into the structure pointed to by lpSI. The function returns nonzero if successful and zero on failure.

The lpSI parameter of both functions points to a structure of type **SCROLLINFO**, which is defined like this:

```
typedef struct tagSCROLLINFO
{UINT cbSize; /* size of SCROLLINFO */
UINT fMask; /* Operation performed */
int nMin; /* minimum range */
int nMax; /* maximum range */
UINT nPage; /* Page value */
int nPos; /* slider box position */
int nTrackPos; /* current tracking position */ } SCROLLINFO;
```

Here, **cbSize** must contain the size of the **SCROLLINFO** structure. The value or values contained in **fMask** determine which of the remaining members are meaningful. Specifically, when used in a call to **SetScrollInfo( )**, the value in **fMask** specifies which scroll bar values will be updated. When used with **GetScrollInfo( )**, the value in **fMask** determines which settings will be obtained. **fMask** must be one or more of these values. (To combine values, simply OR them together.)

SIF_ALL	**Same as SIF_PAGE | SIF_POS | SIF_RANGE | SIF_TRACKPOS.**
SIF_DISABLENOSCROLL	**Scroll bar is disabled rather than removed if its range is set to zero.**
SIF_PAGE	**nPage contains valid information.**
SIF_POS	**nPos contains valid information.**
SIF_RANGE	**nMin and nMax contain valid information.**
SIF_TRACKPOS	**nTrackPos contains valid information.**

nPage contains the current page setting for proportional scroll bars.

nPos contains the position of the slider box.

nMin and **nMax** contain the minimum and maximum range of the scroll bar.

nTrackPos contains the current tracking position. The tracking position is the current position of the slider box while it is being dragged by the user. This value cannot be set.

Working with Scroll Bars

As stated, scroll bars are manually managed controls. This means that your program will need to update the position of the slider box within the scroll bar each time it is moved. To do this you will need to assign **nPos** the value of the new position, assign **fMask** the value **SIF_POS**, and then call **SetScrollInfo( )**. For example, to update the slider box for the vertical scroll bar, your program will need to execute a sequence like the following:

```
SCROLLINFO si;
si.cbSize = sizeof(SCROLLINFO) ;
si.fMask = SIF_POS;
si.nPos = newposition;
SetScrollInfo(hwnd, SB_VERT, &si, 1);
```

The range of the scroll bar determines how many positions there are between one end and the other. By default, window scroll bars have a range of 0 to 100. However, you can set their range to meet the needs of your program. Control scroll bars have a default range of 0 to 0, which means that the range needs to be set before the scroll bar control can be used. (A scroll bar that has a zero range is inactive.)

A Sample Scroll Bar Program

The following program demonstrates both vertical and horizontal standard scroll bars. The scroll bar program requires the following resource file:

```
; Demonstrate scroll bars.
#include "scroll. h"
# include <windows.h>
MyMenu MENU {POPUP "&Dialog"{MENUITEM "&Scroll Bars\tF2", IDM_DIALOG
                              MENUITEM "&Exit\tF3", IDM_EXIT}
            MENUITEM "&Help", IDM_HELP}
MyMenu ACCELERATORS {VK_F2, IDM_DIALOG, VIRTKEY
                     VK_F3, IDM_EXIT, VIRTKEY
```

VK_F1, IDM_HELP, VIRTKEY}

MyDB DIALOG 18, 18, 142, 92 CAPTION "Using Scroll Bars"

STYLE DS_MODALFRAME |WS_POPUP |WS_VSCROLL |WS_HSCROLL |WS_CAPTION | WS_SYSMENU { }

As you can see, the dialog box definition is empty. The scroll bars are added automatically because of the **WS_VSCROLL** and **WS_HSCROLL** style specifications.

You will also need to create this header tile, called SCROLL.H:

```
#define   IDM_DIALOG   100
#define   IDM_EXIT     101
#define   IDM_HELP     102
```

The entire scroll bar demonstration program is shown here. The vertical scroll bar responds to the **SB_LINEUP**, **SB_LINEDOWN**, **SB_PAGEUP**, **SB_PAGEDOWN**, **SB_THUMBPOSITION**, and **SB_THUMBTRACK** messages by moving the slider box appropriately. It also displays the current position of the thumb. The position will change as you move the slider. The horizontal scroll bar only responds to **SB_LINELEFT** and **SB_LINERIGHT**. Its thumb position is also displayed. (On your own, you might try adding the necessary code to make the horizontal scroll bar respond to other messages.) Notice that the range of both the horizontal and vertical scroll bar is set when the dialog box receives a **WM_INITDIALOG** message. You might want to try changing the range of the scroll-bars and observing the results. Sample output from the program is shown in Figure7-1

One other point: notice that the thumb position of each scroll bar is displayed by outputting text into the client area of the dialog box using TextOut( ). Although a dialog box performs a special purpose, it is still a window with the same basic characteristics as the main window. /* Demonstrate Standard Scroll Bars */

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "scroll.h"
#define VERTRANGEMAX 200
#define HORZRANGEMAX 50
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPAAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
```

```
char szWinName[ ] = "MyWin"; /* name of window class */
HINSTANCE hlnst;
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,LPSTR
                   lpszArgs,int nWinMode)
{HWND hwnd; MSG msg; WNDCLASSEX wc1; HANDLE hAccel;
wc1.cbSize=sizeof(WNDCLASSEX);wc1.hInstance=hThisInst;
wc1.lpszClassName=szWinName;wc1.lpfnWndProc=WindowFunc;wc1.style= 0;
wc1.hIcon=LoadIcon(NULL,IDI_APPLICATION); wc1.cbClsExtra=0;
wc1.hIconSm=LoadIcon(NULL, IDI_WINLOGO); wc1.cbWndExtra=0;
wc1.hCursor = LoadCursor(NULL, IDC_ARROW); wc1.lpszMenuName="MyMenu";
wc1.hbrBackground=GetStockObject(WHITE_BRUSH);
 if(!RegisterClassEx(&wc1)) return 0;
hwnd=CreateWindow(szWinName,"ManagingScrollBars",WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
HWND_DESKTOP, NULL, hThisInst, NULL);
hInst = hThisInst; /* save the current instance handle */
/* load accelerators */  hAccel = LoadAccelerators(hThisInst, "MyMenu");
ShowWindow(hwnd, nWinMode) ; UpdateWindow(hwnd) ;
while (GetMessage(&msg, NULL, 0, 0))
   {if ( ! TranslateAccelerator (hwnd, hAccel, &msg) )
         {TranslateMessage (&msg) ; /*translate keyboard messages*/
          DispatchMessage(&msg);/*return control to Windows NT*/} }return msg.wParam;}
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam){ int response;
switch(message)
{case WM_COMMAND:
  switch(LOWORD(wParam))
 {case IDM_DIALOG: DialogBox(hInst,"MyDB",hwnd, (DLGPROC) DialogFunc) ; break;
 CaseIDM_EXIT: response=MessageBox(hwnd,"Quit the Program?","Exit", MB_YESNO);
                  if(response == IDYES) PostQuitMessage(0); break;
  case IDM_HELP: MessageBox(hwnd, "Try the Scroll Bar", "Help", MB_OK) break;
}break;
```

```
 case WM_DESTROY: /* terminate the program */ PostQuitMessage(0);break;
default: return DefWindowProc(hwnd, message, wParam, lParam); } return 0;}
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,WPARAM wParam,
LPARAM lParam) {char str[80];  static int vpos = 0;static int hpos= 0; static
SCROLLINFO si; HDC hdc; PAINTSTRUCT paintstruct;
switch(message) {
case WM_COMMAND:
    swi tch(LOWORD(wParam))
       { case IDCANCEL: EndDialog(hdwnd, 0); return 1; }break;
        case WM_INITDIALOG: si.cbSize = sizeof(SCROLLINFO);
                            si.fMask = SIF_RANGE;
                            si.nMin = 0; si.nMax = VERTRANGEMAX;
                            SetScrollInfo(hdwnd, SB_VERT, &si, 1);
                            si.nMax = HORZRANGEMAX;
                            SetScrollInfo(hdwnd, SB_HORZ, &si, 1); vpos = hpos = 0;
return 1;
case WM_PAINT: hdc = BeginPaint(hdwnd, &paintstruct);
                           sprintf(str, "Vertical: %d", vpos);
                           TextOut(hdc, 1, 1, str, strlen(str));
                            sprintf(str,"Horizontal:%d", hpos); TextOut(hdc, 1, 30, str, strlen(str));
                            EndPaint(hdwnd, &paintstruct); return 1;
case WM_VSCROLL:
  switch(LOWORD(wParam))
   {case SB_LINEDOWN:  vpos++;
                              if(vpos>VERTRANGEMAX) vpos = VERTRANGEMAX; break;
    case SB_LINEUP: vpos- -; if(vpos<0) vpos = 0; break;
   case SB_THUMBPOSITION: vpos = HIWORD(wParam); break;
    case SB_THUMBTRACK: vpos = HIWORD(wParam) break;
   case SB_PAGEDOWN: vpos += 5;
                 if(vpos>VERTRANGEMAX) vpos=VERTRANGEMAX; break;
 case SB_PAGEUP: vpos -= 5; if(vpos<0) vpos = 0;}
/* update vertical bar position */         si.fMask = SIF_POS;      si.nPos = vpos;
```

```
SetScrollInfo(hdwnd, SB_VERT, &si, 1);  hdc = GetDC(hdwnd);
sprintf(str, "Vertical: %d   ", vpos); TextOut(hdc, 1, 1, str, strlen(str));
ReleaseDC(hdwnd, hdc); return 1;,
case WM_HSCROLL:  switch(LOWORD(wParam)){/*Try adding the other event handling
code for the horizontal scroll bar, here. */
  case SB_LINERIGHT: hpos++;
                     if(hpos>HORZRANGEMAX) hpos=HORZRANGEMAX; break;
  case SB_LINELEFT: hpos- -; if(hpos<0) hpos = 0; break;
  case SB_THUMBPOSITION: hpos = HIWORD(wParam); break;
  case SB_THUMBTRACK: hpos = HIWORD(wParam) break;}
 /* update horizontal bar position */ si.fMask = SIF_POS;  si.nPos = hpos ;
SetScrollInfo(hdwnd, SB_HORZ, &si, 1);hdc = GetDC(hdwnd);
 sprintf(str, "Horizontal-. %d   ", hpos); TextOut(hdc, 1, 30, str, strlen(str));
ReleaseDC(hdwnd, hdc);  return 1;} return 0;}
```

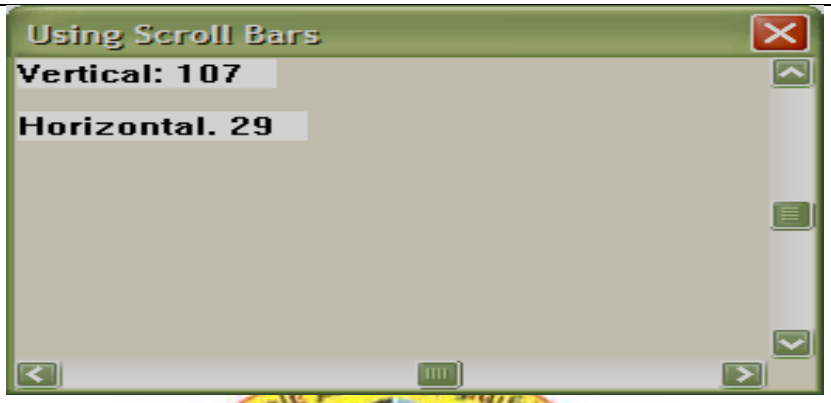


Figure 7-1 output of above program

Using a Scroll Bar Control

A scroll bar control is a stand-alone scroll bar; it is not attached to a window. Scroll bar controls are handled much like standard scroll bars, but two important differences exist. First, the range of a scroll bar control must be set because it has a default range of zero. Thus, it is initially inactive. This differs from standard scroll bars, whose default range is 0 to 100.

The second difference has to do with the meaning of **lParam** when a scroll bar message is received. Recall that all scroll bars — standard or control — generate a **WM_HSCROLL** or a **WM_VSCROLL** message, depending upon whether the scroll bar is horizontal or vertical. When these messages a generated by a standard scroll bar, **lParam** is always zero.

However, when they are generated by a scroll bar control, the handle of the control is passed in **lParam**. In windows that contain both standard and control scroll bars, you will need to make use of this fact to determine which scroll bar generated the message.

Creating a Scroll Bar Control

To create a scroll bar control in a dialog box, use the SCROLLBAR statement, which has this general form: **SCROLLBAR SBID, X, Y, Width, Height [Style]**

Here, SBID is the value associated with the scroll bar. The scroll bar's upper left corner will be at X, Y and the scroll bar will have the dimensions specified by Width and Height. Style determines the exact nature of the scroll bar. Its default style is **SBS_HORZ**, which creates a horizontal scroll bar. For a vertical scroll bar, specify the **SBS_VERT** style. If you want the scroll bar to be able to receive keyboard focus, include the **WS_TABSTOP** style.

Demonstrating a Scroll Bar Control

To demonstrate a control scroll bar, one will be added to the preceding; program. First, change the dialog box definition as shown here. This version adds a vertical scroll bar control.

```
MyDB DIALOG 18, 18, 142, 92 CAPTION "Adding a Control Scroll Bar"
STYLE DS_MODALFRAME |WS_POPUP |WS_CAPTION |WS_SYSMENU
          |WS_VSCROLL  WS_HSCROLL
{
 SCROLLBAR ID_SB1, 110, 10, 10, 70, SBS_VERT | WS_TABSTOP
}
```

Then, add this line to SCROLL.H: #define ID_SB1 200

Next, you will need to add the code that handles the control scroll bar. This code must distinguish between the standard scroll bars and the scroll bar control, since both generate **WM_VSCROLL** messages. To do this, just remember that a scroll bar control passes its handle in **lParam.** For standard scroll bars, **lParam** is zero. For example, here is the **SB_LINEDOWN** case that distinguishes between the standard scroll bar and the control scroll bar.

```
case SB_LINEDOWN:
if((HWND)lParam==GetDlgItem(hdwnd, ID_SBl)
  {/*is control scroll bar*/ cntlpos++;
```

if(cntlpos>VERTRANGEMAX) cntlpos = VERTRANGEMAX;}

else {/* is window scroll bar */ vpos++;

if(vpos>VERTRANGEMAX) vpos = VERTRANGEMAX;}break;

Here, the handle in **lParam** is compared with the handle of the scroll bar control, as obtained using **GetDlgItem( )**. If the handles are the same, then the message was generated by the scroll bar control. If not, then the message came from the standard scroll bar.

As mentioned in Lecture 7, the **GetDlgItem( )** API function obtains the handle of a control. Its prototype is: **HWND GetDlgItem(HWND hDwnd, int ID);**

Here, hDwnd is the handle of the dialog box that owns the control. The control ID is passed in ID. This is the value you associate with the control in its resource file. The function returns the handle of the specified control or **NULL** on failure.

Here is the entire program that includes the vertical scroll bar control. It is the same as the preceding program except for the following additions: First, **DialogFunc( )** defines the static variable cntlpos, which holds the position of the control scroll bar. Second, the control scroll bar is initialized inside **WM_INITDIALOG**. Third, all the handlers that process **WM_VSCROLL** messages determine whether the message came from the standard scroll bar or the control scroll bar. Finally, code has been added to display the current position of the control scroll bar.

Sample output is shown in Figure 7-2. On your own, try adding a horizontal control scroll bar. /* Demonstrate a Control Scroll Bar */

```
#include<windows.h>
#include<string.h>
#include<stdio.h>
#include"scroll.h"
#define VERTRANGEMAX 200
#define HORZRANGEMAX  50
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */ HINSTANCE hInst;
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
int nWinMode) {HWND hwnd; MSG msg; WNDCLASSEX wc1;   HANDLE hAcce;
wc1.cbSize=sizeof(WNDCLASSEX);wc1.hInstance=hThisInst;
wc1.lpszClassName=szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
```

wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION);

wc1.hIconSm = LoadIcon(NULL, IDI_WINLOGO);

wc1.hCursor = LoadCursor(NULL, IDC_ARROW); wc1. lpszMenuNarne = "MyMenu";

wc1.cbClsExtra = 0;wc1.cbWndExtra = 0;

wc1.hbrBackground = GetStockObject (WHITE_BRUSH),; if(!RegisterClassEx(&wc1))

return 0;

hwnd=CreateWindow(szWinName,"ManagingScrollBars",WS_OVERLAPPEDWINDOW,

CW_USEDEFAULT,CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

HWND_DESKTOP, NULL, hThisInst, NULL);

hInst = hThisInst; hAccel = LoadAccelerators(hThisInst, "MyMenu");

ShowWindow(hwnd,nWinMode); UpdateWindow(hwnd) ;

while (GetMessage (&msg, NULL, 0, 0))

{if (ITranslateAccelerator (hwnd, hAccel, &msg))

{TranslateMessage (&msg) ;DispatchMessage (&msg) ;}}return msg.wParam;}

LRESULT CALLBACK WindowFunc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam) {int response;

switch (message)

{case WM_COMMAND:

switch(LOWORD(wParam))

{case IDM_DIALOG:DialogBox(hInst,"MyDB",hwnd, (DLGPROC) DialogFunc); break;

case IDM_EXIT:response=MessageBox(hwnd, "Quit the Program?","Exit", MB_YESNO);

if (response == IDYES) PostQuitMessage (0); break;

case IDM_HELP: MessageBox(hwnd, "Try the Scroll Bar", "Help", MB_OK) break;}break;

case WM_DESTROY: PostQuitMessage (0),break;

default: return DefWindowProc (hwnd, message, wParam, lParam); } return 0 ;}

BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,WPARAM wParam, LPARAM lParam) {char str[80]; static int vpos = 0;static int hpos=0;static int cntlpos=0;

static SCROLLINFO si; HDC hdc; PAINTSTRUCT paintstruct;

switch(message)

{case WM_COMMAND:

switch(LOWORD(wParam)){case IDCANCEL:EndDialog(hdwnd,0);return 1;}break;

case WM_INITDIALOG:

```
 si.cbSize = sizeof(SCROLLINFO);
 si.fMask = SIF_RANGE;
si.nMin = 0; si.nMax = VERTRANGEMAX;
/* set range of standard vertical scroll bar */ GetScrollInfo(hdwnd, SB_VERT, &si, 1);
/*set range of scroll bar */SetScrollInfo(GetDlgItem(hdwnd, ID_SBl), SB_CTL, &si, 1)
si.nMax = HORZRANGEMAX;
   /* set range of standard horizontal scroll bar */ SetScrollInfo(hdwnd, SB_HORZ, &si, 1);
     vpos = hpos = cntlpos = 0; return 1;
  case WM_PAINT:
hdc = BeginPaint(hdwnd, kpaintstruct)
sprintf(str, "Vertical: %d", vpos);
TextOut(hdc, 1, 1, str, strlen(str));
sprintf(str, "Horizontal: %d", hpos);
TextOut(hdc, 1, 30, str, strlen(str))
sprintf(str, "Scroll Bar Control: %d ",cntlpos);
      TextOutfhdc, 1, 60, str, strlen(str));
       EndPaint(hdwnd, &paintstruct); return 1;
case WM_VSCROLL:   switch(LOWORD(wParam)}
   {case SB_LINEDOWN:
if ( (HWND) lParam==GetDlgItem(hdv:nd, ID_SB1))
 {/*is control scroll bar*/cntlpos++;  if(cntlpos>VERTRANGEKAX)
                                        cntlpos = VERTRANGEMAX; }
      else{/* is window scroll bar */ vpos++;
   if(vpos>VERTRANGEMAX) vpos = VERTRANGEMAX; }break;
     case SB_LINEUP:
if((HWND)lParam==GetDlgItem(hdwnd, ID_SBl))
 {/*is control scroll bar*/ cntlpos--;
 if(cntlpos<0) cntlpos = 0;}
 else{/* is window scroll bar */ vpos--; if(vpos<0) vpos = 0; }break;
     case SB_THUMBPOSITION:
if((HWND)lParam==GetDlgItern(hdwnd, ID_SB1))
 {/*is control scroll bar */cntlpos = HIWORD(wParam); /* get current position */ }
```

```
else{/* is window scroll bar */vpos = HIWORD(wParam); /* get current position */}break;
    case SB_THUMBTRACK:
if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1))
 {/*is control scroll bar*/ cntlpos = HIWORD(wParam); /* get current position */ }
else{/*is window scroll bar*/ vpos = HIWORD(wParam); /* get current position */}break;
     case SB_PAGEDOWN:
    if ( (HWND) lP.iram==GetDlgItem(hdwnd, ID_SB1))
    {/*is control scroll bar*/cntlpos += 5; if(cntlpos>VERTRANGEMAX)
cntlpos=VERTRANGEMAX;}
   else{/*is window scroll bar*/vpos += 5;
 if(vpos>VERTRANGEMAX) vpos=VERTRANGEMAX;}break;
     case SB_PAGEUP:
  if((HWND)!Param==GetDlgItem(hdwnd, ID_SB1))
   {/*is control scroll bar */cntlpos -= 5; if(cntlpos<0) cntlpos = 0; }
 else {/* is window scroll bar */vpos -= 5;if(vpos<0) vpos = 0; } break; }
if((HWND)lParam==GetDlgItem(hdwnd, ID_SBl)) {
    /* update control scroll bar position */
   si.fMask = SIF_POS;
   si.nPos = cntlpos;
   SetScrollInfo((HWND)lParam, SB_CTL, &si, 1);
   hdc = GetDC(hdwnd);
   sprintffstr, "Scroll Bar Control: %d   ", cntlpos)
   TextOutthdc, 1, 60, str, strlen(str));   ReleaseDC(hdwnd, hdc); }
     else {/*update standard scroll bar position */
        si.fMask = SIF_POS;
        si.nPos = vpos;
        SetScrollInfo(hdwnd, SB_VERT, &si, 1);
        hdc = GetDC(hdwnd);
       sprintf(str, "Vertical: %d  ", vpos);
       TextOut(hdc, 1, 1, str, strlen(str));
       ReleaseDC(hdwnd, hdc);} return 1;
case WM_HSCROLL:
```

```
swi tch(LOWORD(wParam)) {
/* Try adding the other event handling code for the horizontal scroll bar, here. */
  case SB_LINERIGHT: hpos++;
        if(hpos>HORZRANGEMAX) hpos = HORZRANGEMAX; break;
case SB_LINELEFT: hpos--; if(hpos<0) hpos = 0; }
/* update horizontal scroll bar position */ si.fMask = SIF_POS;
si.nPos = hpos;
SetScrollInfo(hdwnd, SB_HORZ, &si, 1);
 hdc = GetDC(hdwnd);
     sprintf(str, "Horizontal: %d   ", hpos);
    TextOutfhdc, 1, 30, str, strlen(str));
    ReleaseDC(hdwnd, hdc); return 1; }
 return 0;}
```

See in figure 8-2 of output above program

Figure 7-2 Output Program

Check Boxes

A check box is a control that is used to turn on or off an option. It consists of a small rectangle which can either contain a check mark or not. A check box has associated with it a label that describes what option the box represents. If the box contains a check mark, the box is said to be checked and the option is selected. If the box is empty, the option will be deselected.

A check box is a control that is typically part of a dialog box and is generally defined within the dialog box's definition in your program's resource file. To add a check box to a dialog box definition, use either the **CHECKBOX** or **AUTOCHECKBOX** command, which have these general forms:

CHECKBOX "string", CBID, X, Y, Width, Height [, Style]

AUTOCHECKBOX "string", CBID, X, Y, Width, Height [, Style]

Here, string is the text that will be shown alongside the check box. CBID is the value associated with the check box. The box's upper left corner will be at X, Y and the box plus its associated text will have the dimensions specified by Width and Height. Style determines the exact nature of the check box. If no explicit style is specified, then the check box defaults to displaying the string on the right and allowing the box to be tabbed to. When a check box is first created, it is unchecked.

As you know from using Windows NT, check boxes are toggles. Each time you select a check box, its state changes from checked to unchecked, and vice versa. However, this is not necessarily accomplished automatically. When you use the **CHECKBOX** resource command, you are creating a manual check box, which your program must manage by checking and unchecking the box each time it is selected. (That is, a manual check box must be manually toggled by your program.) However, you can have Windows NT perform this housekeeping function for you if you create an automatic check box using **AUTOCHECKBOX**. When you use an automatic check box, Windows NT automatically toggles its state (between checked and not checked) each time it is selected. Since most applications do not need to manually manage a check box, we will be using only **AUTOCHECKBOX** in the examples that follow.

Obtaining the State of a Check Box

A check box is either checked or unchecked. You can determine the status of a check box by sending it the message **BM_GETCHECK** using the **SendDlgItemMessage( )** API function. (**SendDlgItemMessage( )** is described in Lecture 7.) When sending this message, both wParam and lParam are zero. The check box returns **BST_CHECKED (1)** if the box is checked and **BST_UNCHECKED (0)** otherwise.

Checking a Check Box

A check box can be checked by your program. To do this, send the check box a **BM_SETCHECK** message using **SendDlgItemMessage( ).** In this case, wParam determines whether the check box will be checked or cleared. If wParam is **BST_CHECKED**, the check box is checked. If it is **BST_UNCHECKED**, the box is cleared. In both cases, lParam is zero.

As mentioned, manual check boxes will need to be manually checked or cleared by your program each time they are toggled by the user. However, when using an automatic check box your program will need to explicitly check or clear a check box during program initialization only. When you use an automatic check box, the state of the box will be changed automatically each time it is selected.

Check boxes are cleared (that is, unchecked) each time the dialog box that contains them is activated. If you want the check boxes to reflect their previous state, then you must initialize them. The easiest way to do this is to send them the appropriate **BM_SETCHECK** messages when the dialog box is created. Remember, each time a dialog box is activated, it is sent a **WM_INITDIALOG** message. When this message is received, you can set the state of the check boxes (and anything else) inside the dialog box.

Check Box Messages

Each time the user clicks on the check box or selects the check box and then presses the space bar, a **WM_COMMAND** message is sent to the dialog function and the low-order word of **wParam** contains the identifier associated with that check box. If you are using a manual check box, then you will want to respond to this command by changing the state of the box.

IN DEPTH The 3-State Check Box

Windows NT provides an interesting variation of the check box called the 3-state check box. This check box has three possible states: checked, cleared, or grayed. (When the control is grayed, it is disabled.) Like its relative, the 3-state check box can be implemented as either an automatic or manually managed control using the AUTO3STATE and STATE3 resource commands, respectively. Their general forms are shown here:

STATES "string", ID, X, Y, Width, Height [, Style]

AUTO3STATE "string1, ID, X, Y, Width, Height I Style]

Here, string is the text that will be shown alongside the check box. ID is the value associated with the check box. The box's upper left corner will be at X, Y and the box plus its associated text will have the dimensions specified by Width and Height. Style determines the exact nature of the check box. If no explicit style is specified, then the check box defaults to displaying the string on the right and allowing the box to be tabbed to. When a 3-state check box is first created, it is unchecked.

In response to a **BM_GETCHECK** message, 3-state check boxes return **BST_UNCHECKED** if unchecked, **BST_CHECKED** if checked, and **BST_INDETERMINATE** if grayed. Correspondingly, when setting a 3-statc check box using **BM_SETCHECK**, use **BST_UNCHECKED** to clear it, **BST_CHECKED** to check it, and **BST_INDETERMINATE** to gray it.

Radio Buttons

The next control that we will examine is the radio button. Radio buttons are used to present mutually exclusive options. A radio button consists of a label and a small circular button. If the button is empty, then the option is not selected. If the button is filled, then the option is selected. Windows NT supports two types of radio buttons: manual and automatic. The manual radio button (like the manual check box) requires that you perform all management functions. The automatic radio button performs the management functions for you. Because automatic radio buttons are used almost exclusively, they are the only ones examined here.

Like other controls, automatic radio buttons are defined in your program's resource file, within a dialog definition. To create an automatic radio button, use

AUTORADIOBUTTON, which has this general form:

AUTORADIOBUTTON "string", RB1D, X, Y, Width, Height [, Style]

Here, string is the text that will be shown alongside the button. RBID is the value associated with the radio button. The button's upper left corner will be at X,Yand the button plus its associated text will have the dimensions specified by Width and Height. Style determines the exact nature of the radio button. If no explicit style is specified, then the button defaults to displaying the string on the right and allowing the button to be tabbed to. By default, a radio button is unchecked.

As stated, radio buttons are generally used to create groups of mutually exclusive options. When you use automatic radio buttons to create such a group, then Windows NT automatically manages the buttons in a mutually exclusive manner. That is, each time the user selects one button, the previously selected button is turned off. Also, it is not possible for the user to select more than one button at any one time.

A radio button (even an automatic one) may be set to a known state by your program by sending it the **BM_SETCHECK** message using the **SendDlgItemMessage( )**function. The value of wParam determines whether the button will be checked or cleared. If wParam is **BST_CHECKED**, then the button will be checked. If it is **BST_UNCHECKED**, the box

will be cleared. By default, a radio button is unchecked. You can obtain the status of a radio button by sending it the **BM_GETCHECK** message. The button returns **BST_CHECKED** if the button is selected and **BST_UNCHECKED** if it is not.

Generating Timer Messages

Using Windows NT, it is possible to establish a timer that will interrupt your program at periodic intervals. Each time the timer goes off, Windows NT sends a WM_TIMER message to your program. Using a timer is a good way to "wake up your program" every so often. This is particularly useful when your program is running as a background task. To start a timer, use the SetTimer( ) API function, whose prototype is shown here:

UINT SetTimer(HWND hwnd, UINT ID, UINT wLength, TIMERPROC lpTFunc);

Here, hwnd is the handle of the window that uses the timer. Generally, this window will be either your program's main window or a dialog box window. The value of ID specifies a value that will be associated with this timer. (More than one timer can be active.) The value of wLength specifies the length of the period, in milliseconds. That is, wLength specifies how much time there is between interrupts. The function pointed to by lpTFunc is the timer function that will be called when the timer goes off. However, if the value of lpTFunc is NULL, then the window function associated with the window specified by hwnd will be called each time the timer goes off and there is no need to specify a separate timer function. In this case, when the timer goes off, a WM_TIMER message is put into your program's message queue and processed like any other message. This is the approach used by the example that follows. The SetTimer( ) function returns ID if successful. If the timer cannot be allocated, zero is returned. If you wish to define a separate timer function, it must be a callback function that has the following prototype (of course, the name of the function may be different):

VOID CALLBACK TFunc(HWND hwnd, UINT msg, UINT TimerID, DWORD SysTime);

Here, hwnd will contain the handle of the timer window, msg will contain the message WM_TIMER, TimerlD will contain the ID of the timer that went off, and SysTime will contain the current system time.

Once a timer has been started, it continues to interrupt your program until you either terminate the application or your program executes a call to the KillTimer() API function, whose prototype is shown here: **BOOL KillTimer(HWND hwnd, UINT ID);**

Here, hwnd is the window that contains the timer and ID is the value that Identifies that particular timer. The function returns nonzero if successful and zero on failure.

Each time a WM_TIMER message is generated, the value of wParam contains the ID of the timer and lParam contains the address of the timer callback function (if it is specified). For the example that follows, lParam will be NULL.

The Countdown Timer Resource and Header Files

The countdown timer uses the following resource file:

```
; Demonstrate scroll bars, check boxes, and radio buttons.
#include "cd.h"
#include <windows.h>
MyMenu MENU{POPUP "&Dialog"{MENUITEM "&Timer\tF2", IDM_DIALOG
                            MENUITEM "&Exit\tF3", IDM EXIT}
           MENUITEM "&Help", IDM_HELP}
MyMenu ACCELERATORS {VK_F2, IDM_DIALOG, VIRTKEY
                     VK_F3, IDM_EXIT, VIRTKEY
                     VK_F1, IDM_HELP, VIRTKEY}

MyDB DIALOG 18, 18, 152, 92 CAPTION "A Countdown Timer"
STYLE DS_MODALFRAME | WS_POPUP | WS_VSCROLL |WS_CAPTICN | WS_SYSMENU
{PUSHBUTTON "Start", IDD_START, 10, 60, 30, 14, WS_CHILD | WS_VISIBLE | WS_TABSTOP
 PUSHBUTTON "Cancel", IDCANCEL, 60, 60, 30, 14,WS_CHILD | WS_VISIBLE | WS_TABSTOP
 AUTOCHECKBOX "Show Countdown", IDD_CB1, 1, 20, 70, 10
 AUTOCHECKBOX "Beep At End", IDD_CB2, 1, 30, 50, 10
 AUTORADIOBUTTON "Minimize", IDD_RB1, 80, 20, 50, 10
 AUTORADIQBUTTON "Maximize", IDD_RB2, 80, 30, 50, 10
 AUTORADIOBUTTON "As-Is", IDD_RB3, 80, 40, 50, 10}
```

The header file required by the timer program is shown here. Call this file CD.H.

```
#define     IDM_DIALOG   100
#define     IDM_EXIT   101
```

```
#define        IDM_HELP   102
#define         IDD_START   300
#define        IDD_TIMER   301
#define              IDD_CB1   400
#define        IDD_CB2   401
#define              IDD_RB1   402
#define                 IDD_RE2   403
#define                 IDD_RB3   404
```

The Countdown Timer Program

The entire countdown timer program is shown here. Sample output from this program is shown in Figure 7-3. /* A Countdown Timer */

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "cd.h"
#define VERTRANGEMAX 200
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
char szWinName[ ] = "MyWin"; /* name of window class */
HINSTANCE hlnst; HWND hwnd;
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,LPSTR IpszArgs,
int nWinMode)
{MSG  msg; WNDCLASSEX wc1; HANDLE hAccel;
wc1.cbSize=sizeof(WNDCLASSEX);wc1.hInstance=hThisInst;
wc1.lpszClassName=szWinName; wc1.lpfnWndProc = WindowFunc; wc1.style = 0;
wc1.hIcon =  LoadIcon(NULL, IDI_APPLICATION);
wc1.hlconSm= Loadlcon(NULL, IDI_WINLOGO);
wc1.hCursor  = LoadCursor(NULL, IDC_ARROW);
wc1.lpszMenuName="MyMenu";wc1.cbClsExtra=0;wc1.cbWndExtra=0;
wc1.hbrBackground=  GetStockObject(WHITE_BRUSH);  if  (  !RegisterClassEx(&wc1)  )
return 0;
```

```
hwnd = CreateWindow(szWinName, "Demonstrating Controls",WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW__USEDEFAULT,
HWND_DESKTOP, NULL, hThisInst, NULL ); hInst = hThisInst;
/* load accelerators */ hAccel = LoadAccelerators (hThisInst, "MyMenu");
/*Display the window.*/ ShowWindow(hwnd, nWinMode); UpdateWindow(hwnd) ;
while (GetMessagef^msg, NULL, 0, 0))
{if ( ITranslateAccelerator (hwnd, hAccel, &msg) )
  {TranslateMessage (&msg) ;DispatchMessage (&msg) ;} } return msg.wParam;}
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message, WPARAM
wParam,LPARAM lParam) { int response;
 switch (message)
   {case WM_COMMAND:
switch (LOWORD(wParam) )
  { case IDM_DIALOG: DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
break;
    case IDM_EXIT: response=MessageBox (hwnd,"Quit the Program?","Exit" ,
MB_YESNO) ;
                   if (response == IDYES) PostQuitMessage (0) ; break;
case IDM_HELP: MessageBox (hwnd, "Try the Timer", "Help", MB_OK) ; break;
           }break;
  case WM_DESTROY: /* terminate the program */ PostQuitMessage (0) ; break;
    default:  return DefWindowProc (hwnd, message, wParam, lParam) ; } return 0;}
BOOL CALLBACK DialogFunc (HWND hdwnd, UINT message, WPARAM wParam,
LPARAM lParam) {char str [80] ;static int vpos=0;static SCROLLINFO si;
HDC hdc; PAINTSTRUCT paintstruct ; static int t;
switch(message)
 {case WM_COMMAND:
       switch (LOWORD (wParam) )
         {case IDCANCEL: EndDialog (hdwnd, 0) ; return 1;
       case IDD_START: /* start the timer */
         SetTimer (hdwnd, IDD_TIMER, 1000, NULL) ; t = vpos ;
```

```
if(SendDlgItemMessage(hdwnd,IDD_RB1,BM_GETCHECK,0,0)==BST_CHECKED)
        ShowWindow(hwnd, SW_MINIMIZE);
        Else                if(SendDlgltemMessage(hdwnd,IDD_RB2,BM_GETCHECK,0,0)
==BST_CHECKED)
     ShowWindow(hwnd, SW_MAXIMIZE); return 1;}break;
     case WM_TIMER: if(t==0) {KillTimer(hdwnd, DD_TIMER); /*timer went off*/

if(SendDlgltemMessage(hdwnd,IDD_CB2,BM_GETCHECK,0,0)==BST_CHECKED)
          MessageBeep (MB_OK) ;   MessageBox(hdwnd, "Timer Went Off", "Timer",
MB_OK);
        ShowWindow(hwnd, SW_RESTORE); return 1;}
           t- -;/*see if countdown is to be displayed*/
        if(SendDlgltemMessage(hdwnd,IDD_CB1,    BM_GETCHECK,    0,    0)    ==
BST_CHECKED)
           {hdc = GetDC(hdwnd); sprintf(str, "Counting: %d   ", t);
           TextOutthdc, 1, 1, str, strlen(str)); ReleaseDC(hdwnd, hdc);return 1;
 case WM_INITDIALOG:
      si.cbSize = sizeof(SCROLLINFO);
   si.fMask = SIF_RANGE;
   si.nMin = 0; si.nMax = VERTRANGEMAX;
   SetScrollInfo(hdwnd, SB_VERT, &si, 1);
     /* check the As-Is radio button */
       SendDlgItemMessage(hdwnd,IDD_RB3,BM_SETCHECK,    BST_CHECKED,    0);
return 1

 ca se WM_PAINT:
      hdc = BeginPaint(hdwnd, &paintstruct);
   sprintf(str, "Interval: %d", vpos);
   TextOutthdc, 1, 1, str, strlen(str));
     EndPaint (hdwnd, &paintstruct) ,-  return 1;
case WM_VSCROLL:
 switch (LOWORD(wParam) )
```

```
{case SBLINEDOWN: vpos++;
                  if(vpos>VERTRANGEMAX) vpos=VERTRANGEMAX;  break;
 case SB_LINEUP: vpos--; if(vpos<0) vpos = 0;break;
 case SB_THUMBPOSITION: vpos = HIWORD(wParam); /*get current position*/break;
 case SB_THUMBTRACK: vpos = HIWORD(wParam); /* get current position */break;
case SB_PAGEDOWN: vpos += 5;
                  if(vpos>VERTRANGEMAX)      vpos=VERTRANGEMAX;
break;
case  SB_PAGEUP: vpos  -=  5; if(vpos<0)  vpos  =  0; }
  si.fMask = SIF_POS;
  si.nPos   = vpos;
   SeCScrollInfoCndwnd,  SB_VERT,  &si,  1);
  hdc  =  GetDC(hdwnd);
  sprintf(str,  "Interval:  %d  ",  vpos); TextOutfhdc,  1,  1,  str,  strlen(str));
  ReleaseDCM hdwnd,  hdc); return  1;} return 0; }
```

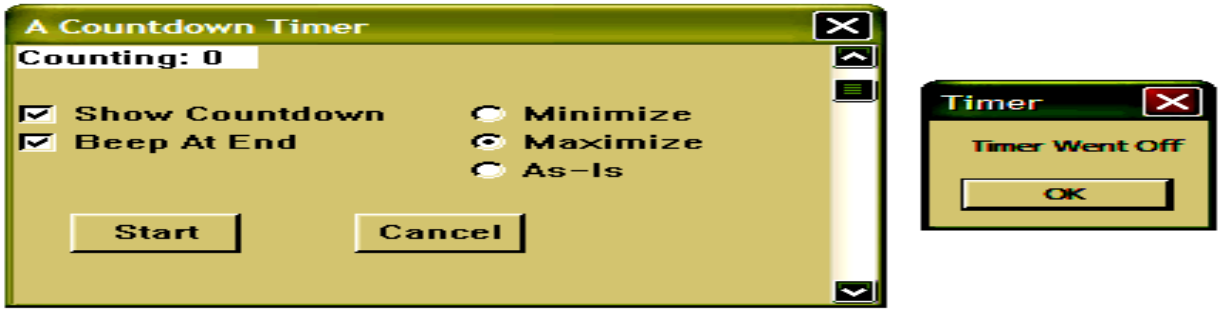


Figure 7-3 Output Count-Down Time Program

A Closer Look at the Countdown Program

To better understand how each control in the countdown program operates, let's take a closer look at it now. As you can see, the vertical scroll bar is used to set the delay. It uses much of the same code that was described earlier In this chapter when scroll bars were examined and no further explanation is needed. However, the code that manages the check boxes and radio buttons deserves detailed attention.

As mentioned, by default no radio button is checked when they are first created. Thus, the program must manually select one each time the dialog box is activated. In this example, each time **a WM_INITDIALOG** message Is received, the As-Is radio button **(IDD_RB3)**

is checked using this statement. **SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, BST_CHECKED, 0);**

To start the timer, the user presses the Start button. This causes the following code to execute:

```
CaseIDD_START:/*start the timer*/SetTimer(hdwnd,IDD_TIMER,1000,NULL); t=vpos;
    if(SendDlgItemMessage(hdwnd,IDD_RB1,BM_GETCHECK,0,0) == BST_CHECKED)
        ShowWindow (hwnd, SW_MINIMIZE) ;
  Else
if(SendDlgItemMessage(hdwnd,IDD_RB2,BM_GETCHECK,0,0)==BST_CHECKED)
            ShowWindow(hwnd, SW_MAXIMIZE); return 1;
```

Here, the timer is set to go off once every second (1,000 milliseconds). The value of the counter variable t is set to the value determined by the position of the vertical scroll bar. If the Minimize radio button is checked, the program windows are minimized. If the Maximize button is checked, the program windows are maximized. Otherwise, the program windows are left unchanged. Notice that the main window handle**, hwnd**, rather than the dialog box handle, **hdwnd**, is used in the call to **ShowWindow().** To minimize or maximize the program, the main window handle-not the handle of the dialog box—must be used. Also, notice that **hwnd** is a global variable in this program.

This allows it to be used inside **DialogFunc( )**. Each time a WM_TIMER message is received, the.following code executes:

```
case WM_TIMER: /* timer went off */
  if(t==0) { KillTimer(hdwnd, IDDJTIMER);
        if(SendDlgItemMessage(hdwnd,IDD_CB2, BM_GETCHECK, 0, 0) ==
BST_CHECKED)
                MessageBeep(MB_OK);
          MessageBox(hdwnd, "Timer Went Off", "Timer", MB_OK);
         ShowWindow(hwnd, SW_RESTORE) ; return 1;} t- -;
/* see if countdown is to be displayed */
if(SendDlgItemMessage(hdwnd,IDD_CB1, BM_GETCHECK, 0, 0) == BST_CHECKED)
{hdc = GetDC(hdwnd); sprintf(str,"Counting: %d",t); TextOut(hdc,1,1,str,strlen(str));
ReleaseDC(hdwnd, hdc);} return 1;
```

If the countdown has reached zero, the timer is killed, a message box informing the user that the specified time has elapsed is displayed, and the window is restored to its former size, if necessary. If the Beep At End button Is checked, then the computer's speaker is beeped using a call to the API function **MessageBeep( ).** If there is still time remaining, then the counter variable t is decremented. If the Show Countdown button is checked, then the time remaining in the countdown is displayed.

MessageBeep( ) is a function you will probably find useful in other programs that you write. Its prototype is shown here: **BOOL MessageBeep(UINT sound);**

Here, sound specifies the type of sound that you want to make. It can be -1, which produces a standard beep, or one of these built in values:

MB_ICONASTERISK	**MB_ICONEXCLAMATION**	**MB_ICONHAND**
MB_ICONQUESTION	**MB_OK**	

MB_OK also produces a standard beep. **MessageBeep()** returns nonzero if successful or zero on failure.

As you can see by looking at the program, since automatic check boxes and radio buttons arc mostly managed by Windows NT, there is surprisingly little code within the countdown program that actually deals with these two controls. In fact, the ease of use of check boxes and radio buttons helps make them two of the most commonly used control elements.

Static Controls

Although none of the standard controls are difficult to use, there is no question that the static controls are the easiest. The reason for this is simple: a static control is one that neither receives nor generates any messages. In short, the term static control is just a formal way of describing something that is simply displayed in a dialog box. Static controls include **CTEXT, RTEXT**, and **LTEXT**, which are static text controls; and **GROUPBOX**, which is used to visually group other controls.

The **CTEXT** control outputs a string that is centered within a predefined area. **LTEXT** displays the string left justified. **RTEXT** outputs the string right justified. The general forms for these controls are shown here:

CTEXT "text", ID, X, Y, Width, Height [, Style]

RTEXT 'text", ID, X, Y, Width, Height [, Style]

LTEXT "text", ID, X, Y, Width, Height [, Style]

Here, text is the text that will be displayed. ID is the value associated with the text. The text will be shown in a box whose upper left corner will be at X, Y and whose dimensions are specified by Width and Height. Style determines the exact nature of the text box. Understand that the box itself is not displayed. The box simply defines the space that the text is allowed to occupy.

The static text controls provide a convenient means of outputting text to a dialog box. Frequently, static text is used to label other dialog box controls or to provide simple directions to the user. You will want to experiment with the static text controls on your own.

A group box is simply a box the surrounds other dialog elements and is generally used to visually group other items. The box may contain a title. The general form for GROUPBOX is shown here: **GROUPBOX "title", ID, X, Y, Width, Height [, Style]**

Here, title is the title to the box. ID is the value associated with the box. The upper left corner will be at X,Yand its dimensions are specified by Width and Height. Style determines the exact nature of the group box. Generally, the default setting is sufficient.

To see the effect of a group box, add the following definition to the resource file you created for the countdown program. GROUPBOX "Display As", 1, 72, 10, 60, 46

After you have added the group box, the dialog box will look like that shown in Figure 7-4. Remember that although a group box makes the dialog box look different, its function has not been changed.

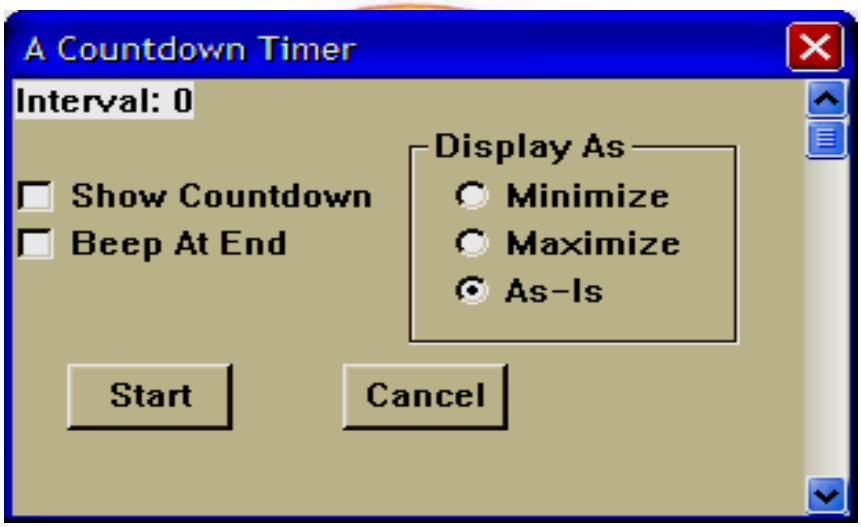


Figure 7-4 The Count-Down dialog box that includes a group box static control

Stand Alone Controls

Although controls are most often used within a dialog box, they may also be free-standing within the client area of the main window. To create a free-standing control, simply use the **CreateWindow( )** function, specifying the name of the control class and the style of control that you desire. The standard control class names are shown here:

BUTTON
COMBOBOX
EDIT
L1STBOX
SCROLLBAR
STATIC

Each of these classes has several style macros associated with it that can be used to customize the control. However, it is beyond the scope of this book to describe them. A list of these style macros can be found by examining WINDOWS.H (and its support files) or by referring to an API reference guide.

The following code creates a free-standing scroll-bar and push button.

```
hsbwnd = CreateWindow(
"SCROLLBAR", /* name of scroll bar class */
"", /* no title */
SBS_HORZ | WS_CHILD | WS_VISIBLE, /* horizontal scroll bar */
10, 10, /* position */
    120, 20, /* dimensions */
hwnd, /* parent window */
NULL, /* no control ID needed for scroll bar */
hThisInst, /* handle of this instance of the program */
NULL /* no additional arguments */
```

```
hpbwnd = CreateWindow(
"BUTTON", /* name of pushbutton class */
"Push Button", /* text inside button */
BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE, /* pushbutton*/
10, 60, /* position */
90, 30, /* dimensions */
hwnd, /* parent window */
(HWND) 500, /* control ID*/
hThisInst, /* handle of this instance of the program */
NULL /* no additional arguments */);
```

As the push button shows, when required, the ID associated with a free-standing control is specified in the ninth parameter to **CreateWindow( )**. As you should recall, this is the parameter that we used to specify the handle of a menu that overrides the class menu. However, when creating a control, you use this parameter to specify the control's ID.

When a free-standing control generates a message, it is sent to the parent window. You will need to add code to handle these messages within the parent's window function.

ALI HASSAN HAMMEDIE
WINDOWS PROGRAMMING LECTURS
COMPUTER SCIENCE DEPARTMENT
UINVERSITY OF TECHNOLOGY
IRAQ-BAGHDAD

Chapter Eight: Work with picture & icon & cursor

Contents:

A bitmap is a display object that contains a rectangular graphical image. The term comes from the fact that a bitmap contains a set of bits which defines the image. Since Windows is a graphics-based operating system, it makes sense that you can include graphical images as resources in your applications. However, bitmaps have broader application than simply providing support for graphical resources. As you will see, bitmaps underlie many of the features that comprise the Windows graphical user interface. They can also help solve one of the must fundamental programming problems facing the Windows programmer: repainting a window. Once you gain mastery over the bitmap you are well on your way to taking charge of many other aspects of Windows. In addition to bitmaps proper, there are two specialized types: the **icon** and the **cursor**. As you know, an icon represents some resource or object. The cursor indicates the current mouse position. So far, we have only been using built-in icons and cursors. Here you will learn to create your own, custom versions of these items.

Two Types of Bitmaps

There are two general types of bitmaps supported by Windows NT: device-dependent and device-independent. Device-dependent bitmaps (DDE) are designed for use with a specific device. Device-independent bitmaps (DIB) are not tied to a specific device. Device-dependent bitmaps were initially the only type available in Windows. However, all versions of Windows since 3.0 have included device-independent bitmaps, too. DIBs are most valuable when you are creating a bitmap that will be used in environments other than the one in which it was created. For example, if you want to distribute a bitmap, a device-independent bitmap is the best way to do this. However, DDEs are still commonly used when a program needs to create a bitmap for its own, internal use. In fact, this is the main reason that DDEs remain widely used. Also, Win32 provides various functions that allow you to convert between DDEs and DIBs, should you need to. The organization of a DDB differs from that of a DIB. However, for the purposes of this chapter, the differences are not important. In fact, the binary format of a bitmap is seldom significant from the application's perspective because Windows provides high-level API functions that manage bitmaps,you will seldom need to "get your hands dirty" with their internals. For the purposes of this chapter, we will be using device-dependent bitmaps because we will be focusing on bitmaps used by the program that creates them.

Two Ways to Obtain a Bitmap

A bitmap can be obtained two different ways: it may be specified resource or it may be created dynamically, by your program, resource is a graphical image that is defined outside your program, but specified in the program's resource file. A dynamic bitmap is created by your program during its execution. Each type is discussed in this chapter, beginning with the bitmap resource.

Using a Bitmap Resource

In general, to use a bitmap resource you must follow these three steps:

1. The bitmap must be specified within your program's resource file.
2. The bitmap must be loaded by your program.
3. The bitmap must be selected into a device context.

This section describes the procedures necessary to accomplish these steps.

Creating a Bitmap Resource

Bitmap resources are not like the resources described in the preceding chapters, such as menus, dialog boxes, and controls. These resources are defined using textual statements in a resource file. Bitmaps are graphical images that must reside in special, bitmap files. However, the bitmap must still be referred to in your program's resource file. A bitmap resource is typically created using an image editor. An **image editor** will usually be supplied with your compiler. It displays an enlarged view of your bitmap. This allows you easily to construct or alter the image example; a custom bitmap is displayed inside the Microsoft C++ image editor.

Except for specialized bitmaps, such as icons and cursors, the It bitmap is arbitrary and under your control. Within reason, you can create bitmaps as large or as small as you like. To try the example that bitmap must be 256 x 128 pixels. Call your bitmap file BP.BMP. if you want your program to produce the results shown in the figures in this chapter, then make your bitmap look like the one shown you have defined your bitmap, create a resource file called BP.RC that contains this line.

As you can guess, the **BITMAP** statement defines a bitmap resource called **MyBP** that is contained in the file BP.BMP. The general form of the **BITMAP** statement is:

BitmapName BITMAP Filename

Here, BitmapName is the name that identifies the bitmap. This name is used by your program to refer to the bitmap. Filename is the name of the file that contains the bitmap.

Displaying a Bitmap

Once you have created a bitmap and included it in your application's resource file, you may display it whenever you want in the client area of a window. However, displaying a bitmap requires a little work on your part. The following discussion explains the proper procedure.

Before you can use your bitmap, you must load it and store its handle. This can be done inside **WinMain( )** or when your application's main window receives a WM_CREATE message. A **WM_CREATE** message is sent to a window when it is first created, out before it is visible. **WM_CREATE** is a good place to perform any initializations that relate to (and are subordinate to) a window. Since the bitmap resource will be displayed within the client area of the main window, it makes sense to load the bitmap when the window receives the **WM_CREATE** message. This is the approach that will be used in this chapter. To load the bitmap, use **the LoadBitmap**( ) API function, whose prototype is shown here:

HBITMAP LoadBitmap(HINSTANCE hThisInst, LPCSTR lpszName);

The current instance is specified in hThisInst and a pointer to the name of the bitmap as specified in the resource file is passed in lpszName. The function returns the handle to the bitmap, or **NULL** if an error occurs. For example:

```
HBITMAP hbit; /* handle of bitmap */
/*…...*/
hbit = LoadBitmap(hInst, "MyBP"); /* load bitmap */
```

This fragment loads a bitmap called MyBP and stores a handle to it in hbit.

When it comes time to display the bitmap, your program must follow these four steps:

1. Obtain the device context so that your program can output to the window.
2. Obtain an equivalent memory device context that will hold the bitmap until it is displayed. (A bitmap is held in memory until it is copied to your window.)
3. Select the bitmap into the memory device context.
4. Copy the bitmap from the memory device context to the window device context. This causes the bitmap to be displayed.

To see how the preceding four steps can be implemented, consider the following fragment. It causes a bitmap to be displayed at two different locations each time a **WM_PAINT** message is received.

```
HDC hdc,   memdc; PAINTSTRUCT   ps;
```

```
case WM_PAINT:
hdc = BeginPaint(hwnd, &ps); /* get device context */
memdc = CreateCompatibleDC(hdc); /* create compatible DC */
SelectObject(memdc, hbit); /* select bitmap */
/*display image */
BitBlt(hdc,10,10,256,128,memdc,0,0,SRCCOPY);
/* display image */
BitBlt(hdc,300,100,256,128,memdc,0,0,SRCCOPY);
EndPaint(hwnd, &ps); /* release DC */
DeleteDC(memdc); /* free the memory context */ break;
```

Let's examine this code, step by step.

First, two device context handles are declared, **hdc** will hold the current window device context as obtained by **BeginPaint( )**. The other, called **memdc**, will hold the device context of the memory that stores the bitmap until it is drawn in the window.

Within the **WM_PAINT** case, the window device context is obtained. This is necessary because the bitmap will be displayed in the client area of the window and no output can occur until your program is granted a device context. Next, a memory context is created that will hold the bitmap. This memory device context must be compatible with the window device context. The compatible memory device context is created using the **CreateCompatibleDC( )** API function. Its prototype is shown here:

HDC CreateCompatibleDC(HDC hdc);

This function returns a handle to a region of memory that is compatible with the device context of the window, specified by hdc. This memory will be used to construct an image before it is actually displayed. The function returns **NULL** if an error occurs.

Next, the bitmap must be selected into the memory device context using the **SelectObject( )** API function. Its prototype is shown here:

HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hObject);

Here, hdc specifies the device context and hObject is the handle of the object being selected into that context. The function returns the handle of the previously selected object (if one exists), allowing it to be reselected later, if desired.

To actually display the bitmap, use the **BitBlt( )** API function. This function copies a bitmap

from one device context to another. Its prototype is shown here:

BOOL BitBlt(HDC HDest, int X, int Y, int Width, int Height, HDC hSource, int SouceX, int SourceY, DWORD dwHow);

Here, hDcst is the handle of the target device context, and X and Fare the upper left coordinates at which point the bitmap will be drawn. The width and height of the destination region are specified in Width and Height. The hSource parameter contains the handle of the source device context, which in this case will be the memory context obtained using **CreateConipatibleDC( ).** The SourceX and SourceY parameters specify the upper left coordinates within the bitmap at which the copy operation will begin. To begin copying at the upper-left corner of the bitmap, these values must be zero. The value of dwHow determines how the bit-by-bit contents of the bitmap will be drawn on the screen. Some of the most common values are shown here:

Macro	Effect
DSTINVERT	Inverts the bits in the destination bitmap
SRCAND	ANDs bitmap with current destination.
SRCCOPY	Copies bitmap as is, overwriting any preexisting output.
SRCPA1NT	ORs bitmap with current destination.
SRCINVERT	XORs bitmap with current destination.

BitBlt( ) returns nonzero if successful and zero on failure.

In the example, each call to **BitBlt( )** displays the entire bitmap by copying it to the client area of the window.

After the bitmap is displayed, both device contexts are released. In this case, **EndPaint( )** is called to release the device context obtained by calling **BeginPaint( )**. To release the memory device context obtained using **CreateCompatibleDC( ),** you must use **DeleteDC(),** which takes as its parameter the handle of the device context to release. You cannot use **ReleaseDC( )** for this purpose. (Only a device context obtained through a call to **GetDC()** can be released using a call to **ReleaseDC( ).**)

Deleting a Bitmap

A bitmap is a resource that must be removed before your application ends. To do this, your program must call **DeleteObject( )** when the bitmap is no longer needed or when a **WM_DESTROY** message is received. **DeleteObject( )** has this prototype:

BOOL DeleteObject(HGDIOBJ hObj);

Here, hObj is the handle to the object being deleted. The function returns nonzero if successful and zero on failure.

The Complete Bitmap Example Program

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc (HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "MyWin"; /* name of window class */
HBITMAP hbit; /* handle of bitmap */
HINSTANCE hInst; /* handle to this instance */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
  LPSTR IpszArgs, int nWinMode)
{HWND hwnd; MSG msg; WNDCLASSEX wcl;
/* Define a window class. */
 wcl.cbSize = sizeof(WNDCLASSEX);
wcl.hInstance = hThisInst; /* handle to this instance */
wcl.lpszClassName = szWinName; /* window class name */
wcl.lpfnWndProc = WindowFunc; /* window function */
wcl.style = 0; /* default style */
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* standard icon */
wcl.hIconSm = LoadIcon(NULL, IDI_APPLICATION); /* small icon */
wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /*cursor style */
wcl.lpszMenuName = NULL; /* no main menu */
wcl.cbClsExtra = 0; /* no extra */
wcl.cbWndExtra =0; /* information needed */
/* Make the window white. */
 wcl.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
/* Register the window class. */
 if(!RegisterClassEx(&wcl)) return 0;
hInst = hThisInst; /* save instance handle */
/* Now that a window class has been registered, a window can be created. */
hwnd = CreateWindow(
```

szWinName, /* name of window class */
"Displaying a Bitmap", /* title */
WS_OVERLAPPEDWINDOW, /* window style - normal */
CW_USEDEFAULT, /* X coordinate - let Windows decide */
CW_USEDEFAULT, /* Y coordinate - let Windows decide */
CW_USEDEFAULT, /* width - let Windows decide */
CW_USEDEFAULT, /* height - let Windows decide */
HWND_DESKTOP, /* no parent window */
NULL, /* no override of class menu */
hThisInst, /* handle of this instance of the program */
NULL /* no additional arguments */);
/* Display the window. */ ShowWindow(hwnd, nWinMode); UpdateWindow(hwnd) ;
/* Create the message loop. */
while (GetMessage(&msg, NULL, 0, 0))
{TranslateMessage(&msg) ; /* translate keyboard messages */
DispatchMessage(&msg) ; /* return control to Windows NT */}
return msg.wParam;}

LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)

{HDC hdc, memdc; PAINTSTRUCT ps;
switch (message) { case WM_CREATE: /* load the bitmap */
hbit = LoadBitmap (hlnst, "MyBP"); /* load bitmap */ break;
case WM_PAINT: hdc = BeginPaint (hwnd, &ps); /*get device context */
memdc= CreateCompatibleDC(hdc); /* create compatible DC */
SelectObject(memdc, hbit); /* select bitmap */
BitBlt(hdc,10,10,256,128,memdc, 0, 0, SRCCOPY); /* display image */
BitBlt(hdc,300,100,256,128,memdc, 0, 0, SRCCOPY); /* display image */
EndPaint(hwnd, &ps); /* release DC */
DeleteDC(memdc); /* free the memory context */ break;
case WM_DESTROY: /* terminate the program */
DeleteObject(hbit); /* remove the bitmap */ PostQuitMessage(0); break;
default: return DefWindowProc(hwnd, message, wParam, lParam);} return 0;}

You might want to experiment with the bitmap program before continuing. For example, try using different copy options with BitBlt(). Also, try bitmaps of differing sizes.

In Depth:

XORing an Image to a Window

As explained, **BltBlt( )** can copy the bitmap contained in one device context into another device context a number of different ways. For example, if you specify **SRCPAINT**, the image is ORed with the destination. Using **SRSCAND** causes the bitmap to be ANDed with the destination. Perhaps the most interesting way to copy the contents of one DC to another uses **SRCINVERT**. This method XORs the source with the destination. There are two reasons this is particularly valuable.

First, XORing an image onto a window guarantees that the image will be visible. It doesn't matter what color or colors the source image or the destination uses; an XORed image is always visible. Second, XORing an image to the same destination twice removes the image and restores the destination to its original condition. As you might guess, XORing is an efficient way to temporarily display and then remove an image from a window without disturbing its original contents.

To see the effects of XORing animage to a window, insert the following cases into **WindowFunc( )** in the first bitmap program.

```
case WM_LBUTTONDOWN: hdc = GetDC(hwnd) ;
rnemdc = CreateCompatibleDC (hdc) ,- /* create compatible DC */
 SelectObjoct (memdc, hbit); /* select bitmap */
```

/* XOR image onto the window */

BitBlt(hdc, LOWORD (lParam) , HIWORD (lParam) ,256,128,memdc, 0, 0, SRCINVERT); ReleaseDC(hwnd, hdc); DeleteDC (memdc) ; break;

case WM_LBUTTONUP :hdc = GetDC(hwnd) ;

memdc = CreateCompatibleDC (hdc) ; /* create compatible DC *?

SelectObject (memdc, hbit); /* select bitmap */

/* XOR image onto the window a second time */

BitBlt(hdc,LOWORD(lParam) , HIWORD(lParam),256, meradc,0,0, SRCINVERT) ;

ReleaseDC(hwnd, hdc); De1eteDC (memdc) ; break;

The code works like this: Each time the left mouse button is pressed, the bitmap is XORed to the window starting at the location of the mouse pointer. This causes an inverted image of the bitmap to be displayed. When the left mouse pointer is released, the image is XORed a second time, causing the bitmap to be removed and the previous contents to be restored. Be careful not to move the mouse while you are holding down the left button. If you do, then the second XOR copy will not take place directly over the top of the first and the original contents of the window will not be restored.

Creating a Custom Icon and Cursor

To conclude this chapter we will examine the creation and use of custom icons and cursors. As you know, all Windows NT applications first create a window class, which defines the attributes of the window, including the shape of the application's icon and mouse cursor. The handles to the icons and the mouse cursor are stored in the hIcon, hIconSm, and hCursor fields of the WNDCLASSEX structure. So far, we have been using the built-in icons and cursors supplied by Windows NT. However, it is possible to define your own.

Defining Icons and Cursors

To use a custom icon and mouse cursor, you must first define their images, using an image editor. Remember, you will need to make both a small and a standard-size icon. Actually, icons come in three sizes: small, standard, and large. The small icon is 16*16, the standard icon is 32*32, and the large icon is 48*48. However, the large icon is seldom used. In fact, most programmers mean the 32*32 icon when they use the term "large icon".

All three sizes of icons are defined within a single icon file. Of course, you don't need to define the large icon. If one is ever needed, Windows will automatically enlarge the standard icon. All cursors are the same size, 32*32.

For the examples that follow, you should call the file that holds your icons ICON.ICO. Be sure to create both the 32 x 32 and the 16 x 16 icons. (The 48 * 48 icon is not needed.) Call the file that holds your cursor CURSOR.CUR.

Once you have defined the icon and cursor images, you will need to add an ICON and a **CURSOR** statement to your program's resource file. These statements have these general forms: **IconName ICON filename**

CursorName CURSOR filename

Here, IconName is the name that identifies the icon and CursorName is the name that identifies the cursor. These names are used by your program to refer to the icon and cursor. The filename specifies the file that holds the custom icon or cursor.

For the example program, you will need a resource file that contains the following statements: **MyCursor CURSOR CURSOR.CUR**

Mylcon ICON YAHOO.ICO

Loading Your Icons and Cursor

To use your custom icons and cursor, you must load them and assign their handles to the appropriate fields in the **WNDCLASSEX** structure before the window class is registered. To accomplish this you must use the API functions **Loadlcon( ) and LoadCursor( ),** which you learned about in Chapter 2. For example, the following loads the icons identified as **Mylcon** and the cursor called **MyCursor** and stores their handles In the appropriate fields of **WNDCLASSEX**.

wcl.hIcon=Loadlcon (hThisInst, "Mylcon"); /* standard icon */

wcl.hIconSm = NULL; /* use small icon in Mylcon */

wcl.hCursor = LoadCursor(hThisInst, "MyCursor"); /* load cursor */

Here, **hThisInst** is the handle of the current instance of the program. In the previous programs in this book, these functions have been used to load default icons and cursors. Here, they will be used to load your custom icons and cursor.

You are probably wondering why **hIconSm** is assigned **NULL**. As you should recall, in previous programs the handle of the small icon is assigned to the hIconSm field of the

WNDCLASSEX structure. However, if this value is **NULL**, then the program automatically uses the 16x16 pixel icon defined in the file that holds the standard icon. Of course, you are free to specify a different icon resource for this icon, if you like.

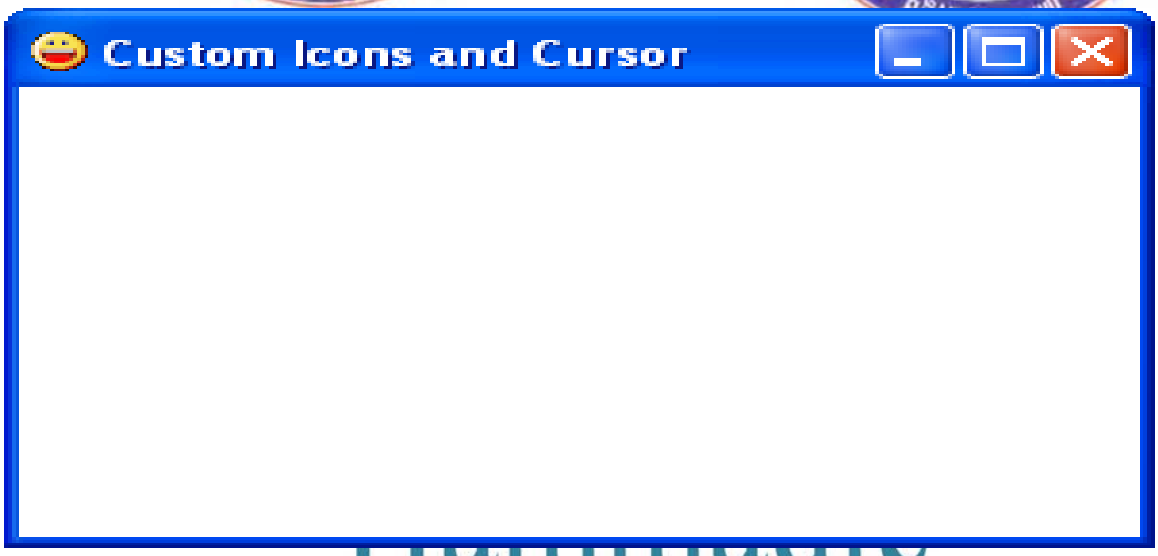
A Sample Program that Demonstrates a-Custom Icon and Cursor

The following program uses the custom icons and cursor. The small icon is displayed in the main window's system menu box and in the program's entry in the task bar. The standard icon is displayed when you move your program to the desktop. The cursor will be used when the mouse pointer is over the window. That is, the shape of the mouse cursor will automatically change to the one defined by your program when the mouse mow* hi program's window. It will automatically revert to its default shape when it moves off the program's window.

Remember, before you try to compile this program, you musl define custom icons and cursor using an image editor and then add uicim |f to the resource file associated with the program.

```
/* Demonstrate custom icons and mouse cursor. */
#include <windows.h>
#include <string.h>
#include <stdio.h>
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
 char szWinName[]="MyWin"; /* name of window class. */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR
lpszArgs,int nWinMode) {HWND hwnd; MSG msg;WNDCLASSEX wcl;
wcl.cbSize = sizeof(WNDCLASSEX);wcl.hInstance=hThisInst;
wcl.lpszClassName=szWinName;wcl.lpfnWndProc=WindowFunc;
wcl.style = 0;wcl.hIcon=LoadIcon(hThisInst, "MyIcon");
wcl.hIconSm=NULL;
wcl.hCursor=LoadCursor(hThisInst, "MyCursor");
wcl.lpszMenuName=NULL;wcl.cbClsExtra =0; wcl.cbWndExtra=0;
wcl.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
/* Register the window class. */if(!RegisterClassEx(&wcl)) return 0;
hwnd=CreateWindow(szWinName,"Custom Icons and Cursor",
 WS_OVERLAPPEDWINDOW, 10, 20, 300, 400,
 HWND_DESKTOP, NULL, hThisInst, NULL);
```

```
ShowWindow (hwnd, nWinMode);UpdateWindow(hwnd);
while (GetMessage(&msg, NULL, 0, 0))
{TranslateMessage(&msg); DispatchMessage(&msg); }
return msg.wParam;}
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam,
LPARAM lParam)
{switch(message)
{case WM_DESTROY: PostQuitMessage(0); break;
default: return DefWindowProc(hwnd, message, wParam, lParam); } return 0;}
```



Of course, your custom icon may look different. The custom mouse cursor will appear when you move the mouse over the window. (Try this before continuing.)

One last point about custom icons: When you create custom icons for your application, you will usually want all sizes of icons to display the same general image since it is this image that is associated with your program.