4th Class

2024 - 2025

Window Programming1

برمجة النوافذ1

أ.د. يسرى حسين

The Components of a Window

Before moving on to specific aspects of Windows programming, a few important terms need to be defined. Figure 1 shows a standard window with each of its elements pointed out.

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. 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.

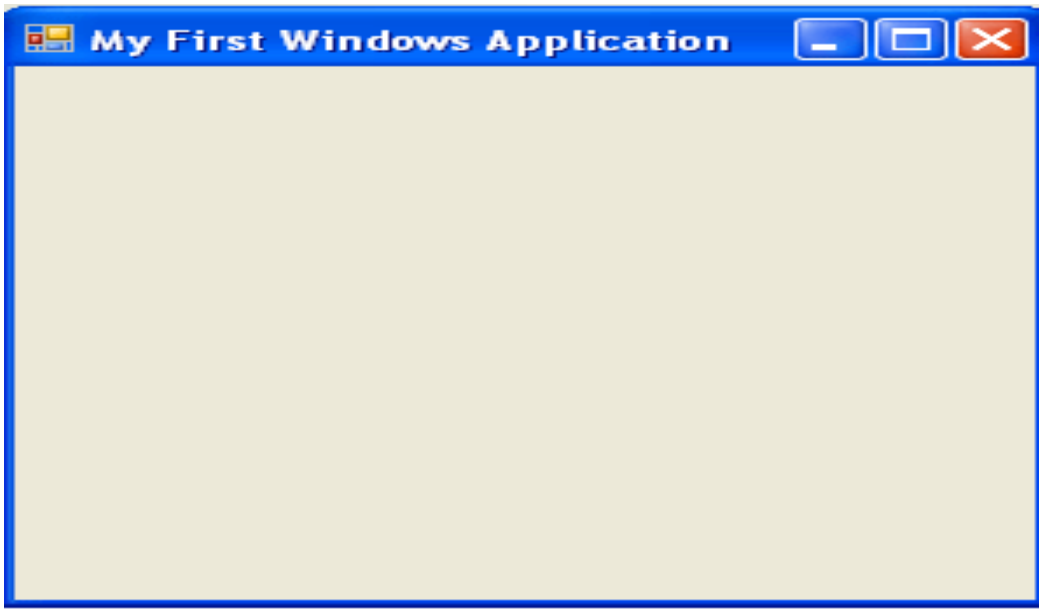


Figure 1: the element of a standard window

Some Windows Application Basics

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

WinMain()

All Windows 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 uses to call **WinMain( ) is WINAPI.** The return type of **WinMain( )** should be **int.**

The Window Procedure

All Windows programs must contain a special function that is *not* called by your program, but is called by Windows. This function is generally called the *window procedure* or *window function.* It is through this function that Windows communicates with your program. The window function is called by Windows 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. 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, 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 will send. For messages that your program doesn't care about, you can let Windows provide default processing. Since there are hundreds of different messages that Windows can generate, it is common for most messages simply to be processed by Windows 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 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 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 communicates with your program by sending it messages. All Windows 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, 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 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 programs do not make extensive use of standard C/C++ data types, such as **int** or **char ***. Instead, all data types used by Windows have been **typdef**ed within the WINDOWS.H file and/or its subordinate files. This file is supplied by Microsoft (and any other company that makes a Windows C/C++ compiler) and must be included in all Windows 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 defines several structures. The two that are needed by the skeleton program arc **MSG** and **WNDCLASSEX**. The **MSG** structure holds a Windows message and **WNDCLASSEX** is a structure that defines a window class. These

structures will be discussed later in this lecture.

A Windows Skeleton

Now that the necessary background information has been covered, it is time to develop a minimal Windows application. As stated, all Windows 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 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.
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 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 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 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 */}
    return msg.wParam; }/* end of WinMain() */
  /* This function is called by Windows 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 process any message not specified  in
the preceding switch statement.*/
  return          DefWindowProc(hwnd,message,wParam,lParam);}
return 0;}/* end WinFunc */
```

Let's go through this program step by step.

First, all Windows 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. 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 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 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, **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, 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 **lpszCIassName,** 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 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 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 **G**et**S**tock**O**bject(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 using the API function **RegisterCIassEx(**), whose prototype is shown here.

ATOM **R**egister**C**lass**E**x(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 **Createindow( )**, whose prototype is shown here

```
HWND CreateWndow (
        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 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 **typedef**ed 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 application that you will create. It essentially tells Windows to send a message to your application that the main window needs to be updated.

The Message Loop

The final part of the skeletal **W**in**M**ain( ) 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. 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 **G**et**M**essage(), 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 **G**et**M**essage( ) will pass control back lo Windows.

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 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 using the **DispatchMessage()** API function. Windows 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. 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. 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 via a call to **DefWindowProc( )**, for default processing. This step is necessary because all messages must be dealt with in one fashion or another.

Although the skeleton developed in lecture 2 forms the framework for a Windows 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. The processing of these messages is at the core of all Windows 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 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**

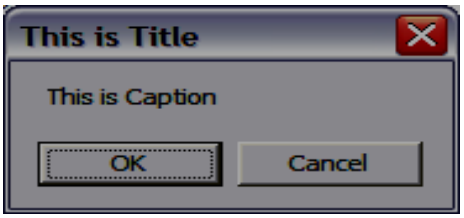
Yes	**IDYES**
Ok	**IDOK**

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 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 Messages

As it relates to Windows, a message is a unique 32-bit integer value. Windows 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 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 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, 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 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 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.
11	Used by Window.
10	Used by Window.
9	Used by Window.
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 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 */
  }return msg. wParam;}
/* This function is called by Windows 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(O); break;
   default: /* Let Windows 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( ):**

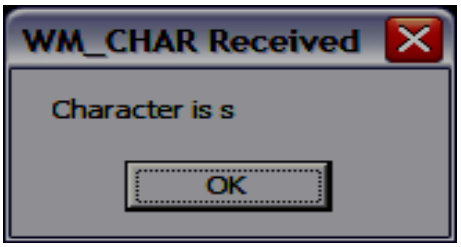


Figure 2-1: Output Program

```
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 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. The codes begin with VK_. For example VK_DOWN corresponding key is Down Arrow.

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; /* 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 and WM_KEYDOWN Messages",
  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 */} return msg. wParam; }
  /* This function is called by Windows 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: /* terminate the program */
                PostQuitMessage (0) ; break;
    default: /* Let Windows process any messages not specified in the preceding switch statement.
 */
        return DefWindowProc(hwnd, message, wParam, lParam); }
return 0; }
```

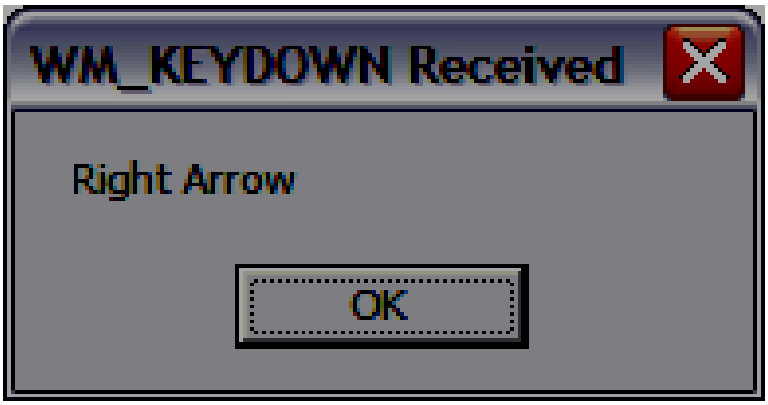


Figure 3-2: Sample output from WM_KEYDOWN program

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***

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 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(O); break;
    default: /* Let Windows 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 **GctDC( )** and **ReleaseDC()** are API functions. Their prototypes are shown here:

HDC GetDC(HWND hwnd);

int ReleaseDC(HWND hwnd, HDC hdc);

GctDC( ) 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 application would use this approach. Later in this lecture, you will learn how to manage text output in a more sophisticated fashion.

No Windows 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 (and Windows in general) does not contain a function that simply outputs a character. Instead, Windows 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 3-1 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) ;
/* Create the message   loop.*/
while(GetMessage(&msg, NULL, 0, 0))
        { TranslateMessage(&msg);/*allow use of keyboard */
          DispatchMessage (&msg); /*return control to Windows */ }retur n msg. wParam;}
 /* This function is called by Windows and is passed messages from the message queue. */
 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 process any messages not specified in the preceding switch
statement.*/return DefWindowProc(hwnd, message, wParam, lParam);} return 0;}
```

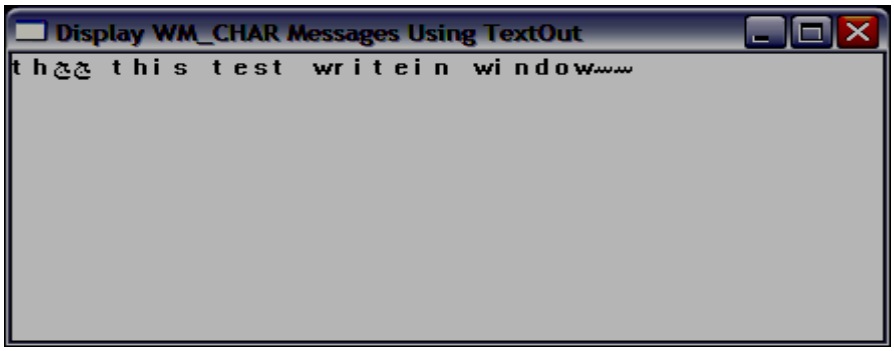


Figure 3-1:Sample window produced using **TextOut**

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 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;

/* Define a window class. */

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 */
```

```
/* Make the window white. */ wc1.hbrBackground = 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, /* name of window class */

"Process WM_PAINT 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 */

   DispatchMessaget&msg); /* return control to Windows */ }return msg. wParam;}

/* This function is called by Windows and is passed messages from the message queue */

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 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;
/* 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(&wc1)) return 0;
/* Now that a window class has been registered, a window can be created. */
hwnd = CreateWindow(
      szWinName, /* name of window class */
      "Routing Output Through WM_PAINT", /* 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 */} 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:
    /* Let Windows process any message not. specified in the preceding switch
statement. */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)

Responding to Mouse Messages

Since Windows is, to a great extent, a mouse-based operating system, all Windows 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 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;
/* 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) ;
  /* Create the message   loop.*/
  while(GetMessage(&msg, NULL, 0, 0))
      { TranslateMessage(&msg);/*allow use of keyboard */
        DispatchMessage (&msg); /*return control to Windows */  }retur n msg. wParam;}
/* This function is called by Windows and is passed messages from the message queue. */
```

```
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;}
```

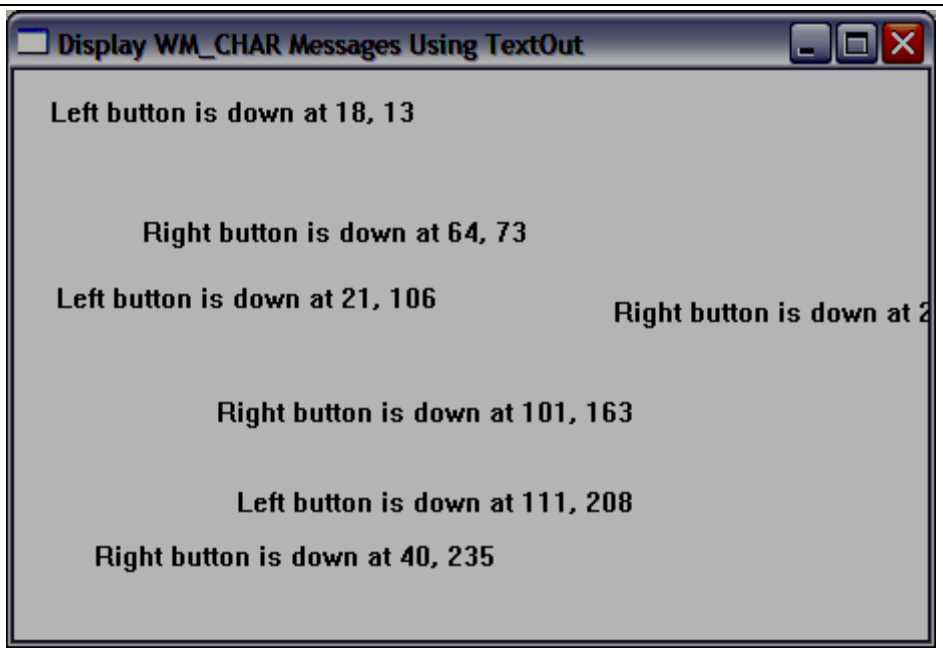


Figure 4-1: Sample output from the mouse message program

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* **G**et**D**ouble**C**lick**T**ime(*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"; /*name of window class*/  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;
  /* 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 = CS_DBLCLKS; /* enable double clicks */
  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 */ );
/* save original double-click time interval */ OrgDblClkTime=GetDoubleClickTime( );
/* 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 */ }retur n msg. wParam;}
/* This function is called by Windows and is passed messages from the message queue. */
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:/*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(1Param),HIWORD(lParam));
TextOut(hdc, LOWORD(IParam), HIWORD(IParam), str, strlen(str) ) ;
ReleaseDC(hwnd, hdc); /* Release DC */break;
      case WM_LBUTTONDBLCLK: /* process  left  button  double-click*/
  interval  = GetDoubleClickTime ( ) ;
  sprintf(str,"Left ButtonXnlnterval  is %u milliseconds",  interval);
  MessageBox(hwnd, str, "DoubleClick", MB_OK); break;
     case WM_RBUTTONDBLCLK:  /*process  right  button  double-click*/
  interval  =  GetDoubleClickTime ();
  sprintf(str,  "Right Button\nlnterval  is %u milliseconds",  interval);
  MessageBoxfhwnd,  str,  "Double Click",  MB_OK); break;
     case WM_DESTROY: /* terminate the program */
SetDoubleClickTime(OrgDblClkTime); /*restore interval*/ PostQuitMessage(0);break;
      default:  return DefWindowProc(hwnd, message, wParam, IParam); } return 0;}
```

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 …!?

This lesson begins our exploration of Windows '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. 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 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 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 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 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 WIN.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: **MENUITEM**s and **POPUP**s**.** A **MENUITEM** specifies a final selection. A **POPUP** specifies a pop-up submenu, which may contain other **MENUITEM**s or **POPUP**s**.** 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;
/* Define a window class. */ 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.cbClsExtra = 0; wc1.cbWndExtra =0;
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 );
    /* Display the window. */  ShowWindow(hwnd, nWinMode); UpdateWindow(hwnd);
   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)  { 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;}
```

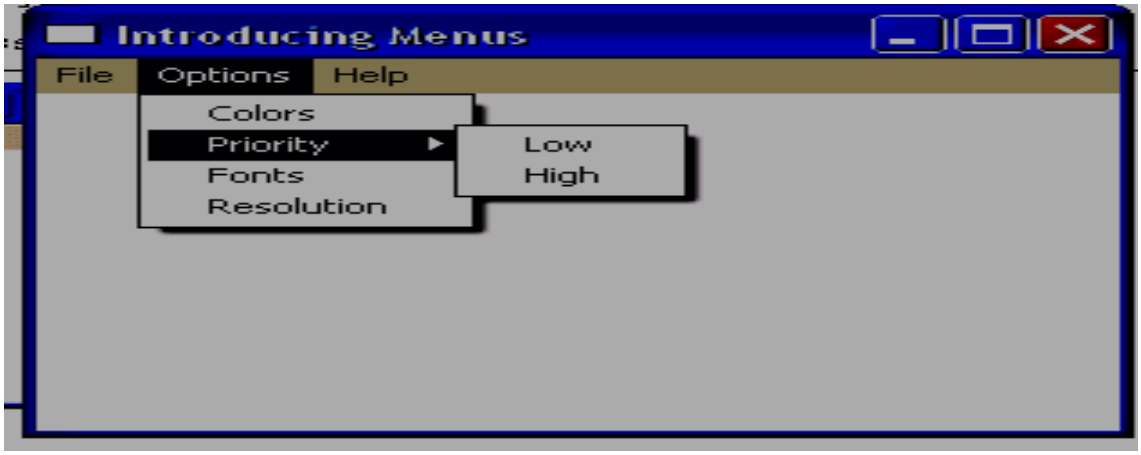


Figure 5.2: output of below program

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 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\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 }
```

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 (HlNSTANCE 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 TransIateAcccIerator(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 DispatchMessagc()** when such a translation takes place. When using **TranslateAccelcrator(**), your message loop should look like this: *while(**G**et**M**essage(&msg, NULL, 0, 0))*

*{if(!**T**ranslate**A**ccelerator(hwnd, hAccel, &msg))*

*{**T**ranslate**M**essage(&msg);* /*allow use of keyboard */

***D**ispatch**M**essage(&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 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.