Programming In Windows

By Minh Nguyen

Table of Contents

****

Overview

Programming in windows seems very difficult because windows provides you with lots of functions that you can use. This text is designed only for understanding the very basics of windows programming and is not intended to cover advance topics. Furthermore, this text will provide many tables and look-up charts that will provide a good reference place when programming. Thus many concepts will be omitted or will only be explained briefly.

Windows Programming Setup

In order to create a windows application, you must create a **Win32** program. Make sure that you choose **GUI** target mode instead of the console target mode. All windows program requires **windows.h** so make sure you include that. Having **windowsx.h** is also a good idea because this file contains other commonly used functions and macros. For now, take out the .RC file and the .DEF file if they were created. It is also a good idea to include this statement in the code as well.

Sample Skeleton Program

```
//Defines
#define WIN32_LEAN_AND_MEAN         //Ensures that unnecessary code are taken out of windows.h

//Include files
#include <windows.h>

//Defines global strings
#define WINDOW_CLASS_NAME "WINCLASS1"

//Globals handles
HWND      main_window_handle = NULL;
HINSTANCE hinstance_app = NULL;

//Processing function of messages
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam)
{
        PAINTSTRUCT ps;
        HDC hdc;
        switch(msg)             //Carries out messages that windows automatically calls
        {
                case WM_CREATE:
                        return(0);
                        break;
                case WM_PAINT:
                        hdc = BeginPaint(hwnd,&ps);
                        EndPaint(hwnd,&ps);
                        return(0);
                        break;
                case WM_DESTROY:
                        PostQuitMessage(0);
                        return(0);
                        break;
                default:
                        break;
        }
        return(DefWindowProc(hwnd,  msg, wparam, lparam));
}
```

```
//The main function for windows
int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hprevinstance, LPSTR lpcmdline, int ncmdshow)
{
	HWND hwnd;		//Handle to windows
	HDC hdc;		//Handle to device context
	WNDCLASSEX winclass;	//Window class type
	MSG msg;		//Holds messages

	//Define window class attributes
	winclass.cbSize = sizeof(WNDCLASSEX);
	winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
	winclass.lpfnWndProc = WindowProc;
	winclass.cbClsExtra = 0;
	winclass.cbWndExtra = 0;
	winclass.hInstance = hinstance;
	winclass.hIcon = LoadIcon(hinstance, IDI_APPLICATION);
	winclass.hCursor = LoadCursor(hinstance, IDC_ARROW);
	winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
	winclass.lpszMenuName = NULL;
	winclass.lpszClassName = WINDOW_CLASS_NAME;
	winclass.hIconSm = LoadIcon(hinstance, IDI_APPLICATION);

	hinstance_app = hinstance;	//Make a duplicate global variable

	if(!RegisterClassEx(&winclass))	//Register the window class
		return(0);

	//Create a window
	if(!(hwnd = CreateWindowEx(	NULL,
					WINDOW_CLASS_NAME, "Graphics App",
					WS_OVERLAPPEDWINDOW | WS_VISIBLE,
					0, 0,
					800, 600,
					NULL,
					NULL,
					hinstance,
					NULL)))
		return(0);

	main_window_handle = hwnd;	//Make a duplicate global variable

	hdc = GetDC(hwnd);		//Get a device context

	//Loop statement
	while(TRUE)
	{
		if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))
		{
			if(msg.message == WM_QUIT)
				break;
			TranslateMessage(&msg);

			DispatchMessage(&msg);
		}
		//Place where programming logic is done
	}

	ReleaseDC(hwnd,hdc);		//Release the device context

	return(msg.wParam);
}
```

That's about it to creating a very simple application. It probably seems complicated for a program that does nothing, but this program will be the core of most windows programs. See if you can understand the code with the comments that have been provided. In the

next section, I will briefly explain what some of the stuff means. If you want, you can just skip that section and move on to using graphics.

In Depth Look At A Basic Program

Okay, the previous code probably will not make sense to a lot of programmers familiar to Dos programming. Most of the programs that you have written, you have been using the **main()** function. In windows, the **main()** function has been replaced with the **WinMain()** function. Let's take a look at the function to see what each part does.

Hungarian Notation

Before proceeding, you should understand the notation used in windows programming. Hungarian notation is the conventional system of naming variables in such a way that programmers could tell what the type of the variable is.

Table 2-1	Hungarian Notation
Common Prefixes	Data types
h	handle
c	char
lp	32-bit pointer
i	int
s	string
p	pointer
msg	message
w	WORD or UINT
dw	DWORD
l	long
by	BYTE or UCHAR
fn	function pointer
b	BOOL

The WinMain Function

```
int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hprevinstance,
                   LPSTR lpcmdline, int ncmdshow)
```

The **WINAPI** (synonymous with **APIENTRY**, **PASCAL**) keyword specify the calling convention of the **WinMain()** function. In standard programming, the default calling convention passes the parameters from right to left. Such calling sequence is used with most standard C and C++ functions. With the WINAPI calling convention, the compiler forces the parameters to be passed from left to right. The differences in the calling sequences lie in the fact that parameters are passed onto a stack. With the right to left calling sequence, allows the function to accept a variable number of parameters. Thus the function cannot clean up the parameters passed to it on the stack because the parameter list can vary. With the left to right calling sequence, the compiler can remove the parameters from the stack easier because it knows during compile time the numbers of variables that a function is expected to accept.

The parameter list includes four parameters, which contain the **handle** to the **instance** of the application, the handle to the previous instance of the application (obsolete), the pointer to the string containing the command lines (if you were using the Run command), and an integer that sets how the application should open the main window respectively.

A **handle** is a unique token or code that identifies an object in windows. A handle is pretty much an integer that is used to distinguish among different objects.

An **instance** of something is a particular of that thing. In this sense, it refers to a copy of the application. When you run a program, windows create an instance of that application. Opening the program again when the program is still running, windows will create another instance of that application.

```
HWND hwnd;              //Handle to windows
HDC hdc;                //Handle to device context
WNDCLASSEX winclass;    //Window class type
MSG msg;                //Holds messages
```

A **device context** (**display context**) is the link between the application, the device driver, and the output device like a printer or plotter. In most cases, think of it as a display surface for the window to write things on.

Programming in windows involves sending **messages**. This is how windows communicate to applications that are running. Messages tell the applications what tasks to do and when to do them. Clicking the mouse, resizing an application, pressing a key, and etc. are all types of actions that involves windows to send specific messages to the application to perform desired tasks.

A **window class** is a template for creating windows that are of members of the class. It contains the common attributes of all windows created based on the class.

```
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(hinstance, IDI_APPLICATION);
winclass.hCursor = LoadCursor(hinstance, IDC_ARROW);
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = WINDOW_CLASS_NAME;
winclass.hIconSm = LoadIcon(hinstance, IDI_APPLICATION);
```

These statements initialize the window class attributes. Note that it is a **struct** data structure and not a **class** data structure. This is beneficial because the member data are all **public,** and therefore, it is easier to set. The structure of the window class follows.

Note: There are many standards that go against what is taught in a professional computer course. For example, the use of global variables are considered bad programming because the scope of the variables will allow easy altering of the variables and this can cause many side effects. Instead, variables should be passed as parameters to reduce side effects. While the reasons are true, global variables are often used in programming because of speed. There is a time penalty for passing parameters because of the function overhead. Encapsulation is another example. Reliability and ease of updating the code is one thing, but speed and use of memory is more important in real world programming. Object oriented programming is a very cool concept, and many programs do use that style of programming for readability, ease of updating, and reliability. However, in real life programming, it is too slow and consumes memory that could be used for other stuff.

```
typedef struct _WNDCLASSEX {
   UINT    style;              //Specifies the class type
   WNDPROC lpfnWndProc;        //Points to the window procedure
   int     cbClsExtra;         //Specifies # of extra bytes to allocate win class structure
   int     cbWndExtra;         // Specifies # of extra bytes to allocate window instance
   HANDLE  hInstance;          //Handle to instance
   HICON   hIcon;              //Handle to icon
   HCURSOR hCursor;            //Handle to cursor
   HBRUSH  hbrBackground;      //Handle to background brush
   LPCTSTR lpszMenuName;       //Handle to resource name of the class menu
   LPCTSTR lpszClassName;      //Specifies the class name
   HICON   hIconSm;            //Handle to the small icon associated with the class
} WNDCLASSEX;
```

Compare this structure declaration with the way it is set.

These are some of the different windows class styles.

Table 2-2 Windows Class Styles

Style	Action
CS_DBLCLKS	Allows double clicks
CS_HREDRAW	Redraw if there is a change in width
CS_OWNDC	Allows unique device context for each windows
CS_VREDRAW	Redraw if there is a change in height

To combine the styles, we must use the **bitwise or** operator (|).

Now that we have define the WNDCLASS and initialize all of the attributes, we must register it to windows and then we can initialize a window instance. Registering the window class lets Windows know about the class and enables us to create windows with the class using only the name of the class.

```
hinstance_app = hinstance;              //Make a duplicate global variable
if(!RegisterClassEx(&winclass))         //Register the window class
        return(0);
//Create a window
```

```
if(!(hwnd = CreateWindowEx(   NULL,
                              WINDOW_CLASS_NAME, "Graphics App",
                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                              0, 0,
                              800, 600,
                              NULL,
                              NULL,
                              hinstance,
                              NULL)))
        return(0);
main_window_handle = hwnd;   //Make a duplicate global variable
hdc = GetDC(hwnd);           //Get a device context
```

Here is the parameter list and a short explanation of each argument of the **CreateWindowEx()** function:

```
HWND CreateWindowEx(
  DWORD  dwExStyle,          // extended window style
  LPCTSTR  lpszClassName,          // address of registered class name
  LPCTSTR  lpszWindowName,         // address of window name
  DWORD  dwStyle,                  // window style
  int  x,                          // horizontal position of window
  int  y,                          // vertical position of window
  int  nWidth,                     // window width
  int  nHeight,                    // window height
  HWND  hwndParent,                // handle of parent or owner window
  HMENU  hmenu,                    // handle of menu, or child-window identifier
  HINSTANCE  hinst,                // handle of application instance
  LPVOID  lpvParam                 // address of window-creation data
 );
```

These are some of the different window styles.

Table 2-2-2 Windows Styles

Style	Action
WS_ OVERLAPPED	A window with title bar and border
WS_ POPUP	Has no controls built
WS_VISIBLE	Specifies that the window is initially visible
WS_OVERLAPPEDWINDOW	A window with title bar, border, and control

Note: To combine the styles, we must use the **bitwise or** operator (|).

Note: All the registering part have error checking (**return(0)** breaks out of the function). This is to ensure the program will stop if something is wrong.

Now we need a message loop to handle events. All it is a loop that retrieves messages sent by windows in the messages queue, translates the messages, and then dispatches the

messages to the event handler. There are two conventional ways of doing this loop, but I have included a way that does not cause a delay when there are no messages.

```
//Loop statement
while(TRUE)
{
        if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))
        {
                if(msg.message == WM_QUIT)
                        break;
                TranslateMessage(&msg);

                DispatchMessage(&msg);
        }

        //Actual coding section
}
```

The section after the if statement is where you would do all the work (writing the actual logic implementation/code). Programming this way is different than DOS programming because you must share the resources. That is after you have done the logic, you must give up the control of the program to Windows in order for Windows to take care of its managements. When Windows is not processing messages, the program is able to perform its actions such as the logic.

Now to end the main loop, you need to release the device context that you have captured. After that, you will return to Windows with the return statement.

```
ReleaseDC(hwnd,hdc);  //Release the device context
return(msg.wParam);
```

Finally, we are done with the WinMain() function. Now we need to take a look at the event handler. The event handler is a function you must write to handle windows messages that you want to deal with.

The Event Handler

This is the function you must write to handle all window messages that you want. Windows will send the application lots of messages, but you can choose which ones you want to work with. Messages that are not handled by your application will be sent to the **DefWindowProc()** where it will be processed.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wparam,
                                        LPARAM lparam)
```

First, **LRESULT** is just a 32-bit integer that is returned by this function. It is used specifically for messages processing functions. **CALLBACK** functions are functions that you write but do not call yourself. Windows call this function when an event occurs.

Remember that events occur when the user makes actions such as moving the mouse, resizing the window, pressing a key, and etc. Events can also be other messages that Windows send to the application when it is created, destroyed, hidden, and etc. The hwnd is the handle to the window, msg is the message, **wparam** and **lparam** are more info on the message. Different messages carry different extra info.

```
PAINTSTRUCT ps;
HDC hdc;
switch(msg)     //Carries out messages that windows automatically calls
{
        case WM_CREATE:
                return(0);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hwnd,&ps);
                EndPaint(hwnd,&ps);
                return(0);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                return(0);
                break;
        default:
                break;
}
```

The window messages, **WM_CREATE**, **WM_PAINT**, **WM_DESTROY**, are the basic window messages. WM_CREATE is sent when the application is first started. WM_PAINT is sent when the application is activated after it has been covered by another application. WM_DESTROY is sent when the application is being closed.

Simple enough, these are the messages that Windows send to the program to see if it needs repainting, initializing, or cleaning up. To check for other messages, all you have to do is add in other cases in the switch statement. You can also do ifs statements but it is neater this way. Here are the most commonly used messages.

Table 2-3	**Messages**
Most Common Messages	Sent When
WM_ACTIVATE	Activated
WM_CLOSE	Closed
WM_CREATE	First created
WM_PAINT	Window needs redrawing
WM_DESTROY	The window is to be destroyed
WM_KEYDOWN	A key has been pressed
WM_KEYUP	A key has been released
WM_MOVE	The window moves
WM_MOUSEMOVE	The mouse moves
WM_SIZE	The window is resized

Finally, last part:

```
return(DefWindowProc(hwnd,  msg, wparam, lparam));
```

The return function will make a call **DefWindowProc()** to resolve messages that was not handled. And we are done with examining the skeleton program. Next section will cover the basics of using graphics.

Windows Programming Basics

Now that you have an understanding of the skeleton program, you can use this structure to create other window programs. Just use the core program as the backbone of any program that you write. Your initialization code should be written before the main event loop and your cleaning up code should be written after the main event loop. In your main event loop is where you should write your programming logic.

Graphics

When using graphics, you must have a device context set. Recall that a device context) is the link between the application, the device driver, and the output device. You set it up as follow:

```
HDC hdc;
hdc = GetDC(hwnd);
```

This gets a handle to the device context for the window area of the current window. The device context can then be used to allow other GDI functions to draw in the client area. After you are finished with a device context, you must release it using the following function.

```
ReleaseDC(hwnd, hdc);
```

Sounds simple? Good. Now, in order to start drawing, you need to know a few things.

A **pen** is an object used to draw an outline of objects like a line, the border of a rectangle. A **brush** is an object used to fill the interior of objects like drawing a filled rectangle.

To define them, we create handles to those objects.

```
HPEN pen;
HBRUSH brush;
```

What we have now are two handles. One points to a pen and one points to a brush. Both do not exist, however. Now we must assign the handles to valid pens and brushes.

We can use predefine pens and brushes like so:

```
pen = (HPEN)GetStockObject(WHITE_PEN);
brush = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

Then, we must select them into a device context like so:

```
SelectObject(hdc, pen);
SelectObject(hdc, brush);
```

However, there are not a lot of choices of standard pens and brush to choose from.

Table 3-1	Standard Objects
Value	Meaning
BLACK_BRUSH	Black brush
DKGRAY_BRUSH	Dark gray brush
GRAY_BRUSH	Gray brush
HOLLOW_BRUSH	Hollow brush
LTGRAY_BRUSH	Light gray brush
NULL_BRUSH	Null brush (HOLLOW_BRUSH)
WHITE_BRUSH	White brush
BLACK_PEN	Black pen
NULL_PEN	Null pen
WHITE_PEN	White pen

To create your own custom pen you must use the **CreatePen()** function.

Here is the **CreatePen()** function prototype:

```
HPEN CreatePen(
   int  fnPenStyle,              // pen style
   int  nWidth,                  // pen width
   COLORREF  crColor             // pen color
  );
```

You can use this function as follow, provided that you provide a color type. Don't worry about that right now. Here is how the function call would work.

```
pen = CreatePen(PS_SOLID, 1, COLOR);
```

Table 3-1-2	Basic Pen Styles
Style	Description
PS_SOLID	Pen is solid
PS_DASH	Pen is dashed
PS_DOT	Pen is dotted
PS_DASHDOT	Pen is dashed, dotted…
PS_DASHDOTDOT	Pen is dashed, dotted, dotted…
PS_NULL	Pen is invisible

There is also another way to create a logical pen using **CreatePenIndirect()** function but don't worry about it. Let's move on to creating brushes.

Here is the function prototype for creating a solid brush. All you have to do is specify the color.

```
HBRUSH CreateSolidBrush(
   COLORREF  crColor );        // brush color value
```

There are other types of brushes too, but to use them, you must define a logical brush, which I won't go into now. This is by far, the simplest type of brush you can make. Here is how the function call would work.

```
brush = CreateSolidBrush(COLOR);
```

Remember, we must select them into a device context to use them:

```
SelectObject(hdc, pen);
SelectObject(hdc, brush);
```

This is the **SelectObject()** function prototype:

```
HGDIOBJ SelectObject(
   HDC  hdc,                    // handle of device context
   HGDIOBJ  hgdiobj             // handle of object
  );
```

The return type is the old object. Thus, if you want to save the old pen or brush, you should assign it to the function. That way you can reselect the original after you are done.

After you have used a pen or a brush to draw shapes or objects, you must be sure to delete the pen or brush because they waste memory. Thus, when an application no longer requires a given pen or brush, it should call the **DeleteObject()** function to delete the pen from the device context. This is really important because it will cause memory errors when you run the program.

```
DeleteObject(pen);
DeleteObject(brush);
```

Now, let us move on to colors. Colors are defined using COLORREF data type. You define one as follows:

```
COLORREF color;
```

To set a color, you can manually shift in the red, green, and blue bits or you can use the **RGB()** macro like so:

```
color = RGB(RED,GREEN,BLUE);
```

RED, GREEN, and BLUE have the value range from 0 to 255.

Note: There is another way to specify colors. You can use palettes but this text will not cover that. Using RGB is much easier.

Finally, you are ready to draw. All you need now is the functions to draw simple objects such as a pixel, line, ellipses, and rectangles.

First, I'll show you how to draw a pixel. A pixel is just a small dot on the screen so it will not need a pen or a brush to be defined.

```
COLORREF SetPixel(
  HDC  hdc,                 // handle of device context
  int  X,                   // x-coordinate of pixel
  int  Y,                   // y-coordinate of pixel
  COLORREF  crColor         // pixel color
 );
```

The return type is the color that you have chosen. This should be the same as the pixel color. However, this may differ if the exact color cannot be found. This should provide enough information for you to call the function. Here how if you don't know. Remember, you should be able to figure it out from now on.

```
SetPixel(hdc, x, y, color);
```

Here is the prototype for function to draw a line. Note this requires only a pen.

```
BOOL MoveToEx(
  HDC  hdc,             // handle of device context
  int  X,               // x-coordinate of new current position
  int  Y,               // y-coordinate of new current position
  LPPOINT  lpPoint      // address of old current position
 );

BOOL LineTo(
  HDC  hdc,             // device context handle
  int  nXEnd,           // x-coordinate of line's ending point
  int  nYEnd            // y-coordinate of line's ending point
 );
```

The **MoveToEx()** function is simple enough. It will move to a coordinate, without actually drawing. The address of the old position is optional. If you choose not to store the old position, set the lpPoint parameter to NULL. If you use it, the function will store the old position in the variable.

Here is the point structure:

```
typedef struct _POINT {
  LONG x;
  LONG y;
} POINT;
```

The **LineTo()** function will draw a line from the starting coordinate to the new one specified.

Now here are the function prototypes for the other objects.

```
BOOL Rectangle(
   HDC  hdc,            // handle of device context
   int  nLeftRect,      // x-coord. of bounding rect. upper-left corner
   int  nTopRect,       // y-coord. of bounding rect. upper-left corner
   int  nRightRect,     // x-coord. of bounding rect. lower-right corner
   int  nBottomRect     // y-coord. of bounding rect. lower-right corner
  );

BOOL RoundRect(
   HDC  hdc,            // handle of device context
   int  nLeftRect,      // x-coord. of bounding rect. upper-left corner
   int  nTopRect,       // y-coord. of bounding rect. upper-left corner
   int  nRightRect,     // x-coord. of bounding rect. lower-right corner
   int  nBottomRect,    // y-coord. of bounding rect. lower-right corner
   int  nWidth,         // width of ellipse used to draw rounded corners
   int  nHeight         // height of ellipse used to draw rounded corners
  );

BOOL Ellipse(
   HDC  hdc,            // handle of device context
   int  nLeftRect,      // x-coord. of upper-left corner of bounding rect.
   int  nTopRect,       // y-coord. of upper-left corner of bounding rect.
   int  nRightRect,     // x-coord. of lower-right corner of bounding rect.
   int  nBottomRect     // y-coord. of lower-right corner of bounding rect.
  );

BOOL Chord(
   HDC  hdc,            // handle of device context
   int  nLeftRect,      // x-coordinate of the upper-left corner of the bounding
                        // rectangle
   int  nTopRect,       // y-coordinate of the upper-left corner of the bounding
                        // rectangle
   int  nRightRect,     // x-coordinate of the lower-right corner of the bounding
                        // rectangle
   int  nBottomRect,    // y-coordinate of the lower-right corner of the bounding
                        // rectangle
   int  nXRadial1,      // x-coordinate of the first radial's endpoint
   int  nYRadial1,      // y-coordinate of the first radial's endpoint
   int  nXRadial2,      // x-coordinate of the second radial's endpoint
   int  nYRadial2       // y-coordinate of the second radial's endpoint
  );
```

Here are other commonly used functions. It will be listed so that you can use to draw other shapes.

```
int FillRect(
   HDC  hdc,                    // handle of device context
   CONST RECT  *lprc,           // address of structure with rectangle
   HBRUSH  hbr                  // handle of brush
  );

int FrameRect(
   HDC  hdc,                    // handle of device context
   CONST RECT  *lprc,           // address of rectangle coordinates
   HBRUSH  hbr                          // handle of brush
  );

BOOL Pie(
   HDC  hdc,              // handle of device context
   int  nLeftRect,        // x-coord. of bounding rect. upper-left corner
   int  nTopRect,         // y-coord. of bounding rect. upper-left corner
   int  nRightRect,       // x-coord. of bounding rect. lower-right corner
   int  nBottomRect,      // y-coord. of bounding rect. lower-right corner
   int  nXRadial1,        // x-coord. of first radial's endpoint
   int  nYRadial1,        // y-coord. of first radial's endpoint
   int  nXRadial2,        // x-coord. of second radial's endpoint
   int  nYRadial2         // y-coord. of second radial's endpoint
  );

BOOL AngleArc(
   HDC  hdc,                    // handle of device context
   int  X,                      // x-coordinate of circle's center
   int  Y,                      // y-coordinate of circle's center
   DWORD  dwRadius,             // circle's radius
   FLOAT  eStartAngle,          // arc's start angle
   FLOAT  eSweepAngle           // arc's sweep angle
  );

BOOL Arc(
   HDC  hdc,    // handle of device context
   int  nLeftRect,        // x-coordinate of upper-left corner of bounding
                          // rectangle
   int  nTopRect,         // y-coordinate of upper-left corner of bounding
                          // rectangle
   int  nRightRect,       // x-coordinate of lower-right corner of bounding
                          // rectangle
   int  nBottomRect,      // y-coordinate of lower-right corner of bounding
                          // rectangle
   int  nXStartArc,       // first radial ending point
   int  nYStartArc,       // first radial ending point
   int  nXEndArc,         // second radial ending point
   int  nYEndArc          // second radial ending point
  );
```

These are the basic objects that are commonly used to draw in the window area. That's pretty much all there is to using graphics. There are a few more functions that windows provide for you, but right now these basic ones will suffice.

Keyboard Input

Using the keyboard is very easy. There are a few ways to check to see whether or not a key has been pressed. The simplest method is to check for keyboard messages.

The two messages that we can check for in the message queue includes:
WM_KEYDOWN, and **WM_KEYUP**.

Windows provides us the code for each keys pressed also known as a virtual-key code value in the wParam of the WM_KEYDOWN and WM_KEYUP messages. These things are device independent meaning it doesn't not use the hardware-dependent scan code to identify which keys are pressed. Here is a list of the virtual key code list:

Table 3-2	**Represented Virtual Key Code**
Symbolic Name	Description
VK_LBUTTON	Left mouse button
VK_RBUTTON	Right mouse button
VK_MBUTTON	Middle mouse button
VK_CANCEL	Control-break
VK_BACK	Backspace key
VK_TAB	Tab key
VK_CLEAR	#5 on numpad with num lock off
VK_RETURN	Enter key
VK_SHIFT	Shift key
VK_CONTROL	Control key
VK_MENU	Alt key
VK_PAUSE	Pause key
VK_CAPITAL	Caps lock
VK_ESCAPE	Escape key
VK_SPACE	Space bar
VK_PRIOR	Page up
VK_NEXT	Page down
VK_END	End key
VK_HOME	Home key
VK_LEFT	Left arrow
VK_UP	Up arrow
VK_RIGHT	Right arrow
VK_DOWN	Down arrow
VK_SNAPSHOT	Print screen
VK_INSERT	Insert
VK_DELETE	Delete

These are other virtual key codes used for keyboard input. Since the symbolic name is not identified, we can use the hex notation, the decimal equivalence, or the character notation as well.

Here I have listed other virtual key codes using the character notation:

Table 3-2-2 **Represented Virtual Key Code**

Key	Virtual Code (Not implemented)	Description
'0'	VK_0	0 key
'1'	VK_1	1 key
'2'	VK_2	2 key
'3'	VK_3	3 key
'4'	VK_4	4 key
'5'	VK_5	5 key
'6'	VK_6	6 key
'7'	VK_7	7 key
'8'	VK_8	8 key
'9'	VK_9	9 key
'A'	VK_A	A key
'B'	VK_B	B key
'C'	VK_C	C key
'D'	VK_D	D key
'E'	VK_E	E key
'F'	VK_F	F key
'G'	VK_G	G key
'H'	VK_H	H key
'I'	VK_I	I key
'J'	VK_J	J key
'K'	VK_K	K key
'L'	VK_L	L key
'M'	VK_M	M key
'N'	VK_N	N key
'O'	VK_O	O key
'P'	VK_P	P key
'Q'	VK_Q	Q key
'R'	VK_R	R key
'S'	VK_S	S key
'T'	VK_T	T key
'U'	VK_U	U key
'V'	VK_V	V key
'W'	VK_W	W key
'X'	VK_X	X key
'Y'	VK_Y	Y key
'Z'	VK_Z	Z key

Remember that these keys do not have those virtual keys implemented.

Note: The virtual-key codes do not check for lowercase letter. In order to do so you can check the state of the shift key or the caps lock key while the key is pressed. More on checking later.

More virtual-key codes that are defined in **WINDOWS.H**:

Table 3-2-3	Represented Virtual Key Code
Symbolic Name	Description
VK_NUMPAD0	Num lock on #0
VK_NUMPAD1	Num lock on #1
VK_NUMPAD2	Num lock on #2
VK_NUMPAD3	Num lock on #3
VK_NUMPAD4	Num lock on #4
VK_NUMPAD5	Num lock on #5
VK_NUMPAD6	Num lock on #6
VK_NUMPAD7	Num lock on #7
VK_NUMPAD8	Num lock on #8
VK_NUMPAD9	Num lock on #9
VK_MULTIPLY	Multiply key
VK_ADD	Add key
VK_SUBSTRACT	Subtract key
VK_DECIMAL	Decimal key
VK_DIVIDE	Divide key
VK_F1	F1 key
VK_F2	F2 key
VK_F3	F3 key
VK_F4	F4 key
VK_F5	F5 key
VK_F6	F6 key
VK_F7	F7 key
VK_F8	F8 key
VK_F9	F9 key
VK_F10	F10 key
VK_F11	F11 key
VK_F12	F12 key
VK_NUMLOCK	Num Lock key

These are the most basic keys used in most application. Thus I will not show u how to use special characters for inputting. This is more than enough to work with keyboard input. As a general rule, you can always use the character notation of each special character to check for the key pressed.

Now that you know the keyboard virtual-key code, it's time you know how to use them in checking.

First you must add the **case WM_KEYDOWN**: in your message handle switch statement. Then you must typecast the **wparam** parameter values to int in order to check the message info with the virtual-key code.

Here is an example:

```
//All of this should be in the switch msg structure that we have created
case WM_KEYDOWN:
        if((int)wparam == VK_ESCAPE)        //Typecast wparam and compare
                ;       //Do action here
        if((int)wparam == VK_PAUSE)         //Typecast wparam and compare
                ;       Do action here
```

If you want to check for when a key is up you would use the WM_KEYUP window message.

Supposed we do not want to check the key states inside our message handle function using messages. We need to use another approach.

The second way to check for a key pressed is to use **GetKeyState()** function. This is especially useful for determining the state of the shift, control, num lock, and caps lock keys. Here is how you would use this function.

```
int keystate;     //int type

//Check for the state of the key
//The high order bit is 1 when key is down, 0 if up
keystate = GetKeyState(VK_SHIFT);
```

So how do we check the value of the high order bit? We can check by using the **bitwise and** operator (&). Thus:

```
BOOL state;
state = (GetKeyState(VK_SHIFT) & 0x8000);
```

The **GetKeyState()** function returns the state of the virtual key as it was at the time the last message was retrieved by the **GetMessage()** function. This means that the key state retrieved doesn't necessary have to be the state of the key at the current time. Therefore windows have another function we can use, the **GetAsyncKeyState()** function.

Note: The **GetAsyncKeyState()** function as well as the **GetKeyState()** function will return a value that will signify if the virtual key is currently down or up. The high-order bit will determine that. If it is a 1 then the key is currently up. If it is a 0 then the key is currently down. By checking the return value's low-order bit we can determine if the key was toggled. If it is a 1 then the key was toggled. If it is a 0 then the key was not toggled.

Note: To check the high-order bit (0x8000) you must use the **bitwise and** (&) plus the high order bit. The same is true for the low-order bit (0x0000).
Windows can also let us check the whole state of the keyboard. However, such an approach will not be covered here.

Now we are done. Well, almost. Using the checking method every time we need to check a key can be long and hard to read. Thus we should define a macro to do this for us. Here are two macros that will return the state of the keys. A return value of 1 means the function is true and false otherwise.

```
#define KEYDWN(vk_code)  ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEYUP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)
```

Note: Macros should be on one line only.

Now, all we need to do is use these macros to determine the key state of any key we wish. Remember to place these macros at the beginning of your program before using them. Just in case you would use them like so:

```
if(KEYDWN(VK_SHIFT))
        ;                       //Action
if(KEYUP(VK_SHIFT))
        ;                       //Action
```

Now we are officially done with the keyboard input section. Let's move on to using mouse input.

Mouse Input

Mouse input is pretty simple. It arrives in the form of messages, just like keyboard input. There is a lot of mouse messages you can process in your event handler but you only need to worry about a few commonly used ones. Well here are the messages:

Table 3-3	**Messages**
Most Common Messages	Sent When
WM_LBUTTONDOWN	Left button pressed
WM_LBUTTONUP	Left button released
WM_LBUTTONDBLCLK	Left button double-clicked
WM_RBUTTONDOWN	Right button pressed
WM_RBUTTONUP	Right button released
WM_RBUTTONDBLCLK	Right button double-clicked
WM_MOUSEMOVE	The mouse moves

Mouse messages, unlike keyboard messages can be lost. This is due to the mouse hardware, the rate that Windows receives and processes the messages, and the rate at which the mouse is moving.

Note: Since mouse messages can be lost be clicking in one window and releasing it in another, it is probably a good idea to make the window the size of the whole screen. This way you do not need process those types of messages.

Note: Using WM_RBUTTONDBLCLK and WM_LBUTTONDBLCLK require having CS_DBLCLKS style define in your **winclass** (see **window class styles** attributes). If this is not included, double click messages are interpreted as pressing the button down.

Now we should learn how to process those messages. First let's begin by finding the position of the mouse at the time when a message is generated.

Before we begin, let's review the point structure.

```
typedef struct _POINT {
   LONG x;
   LONG y;
} POINT;
```

Mouse messages return the **lparam** that holds the cursor position. In order to extract that information we can use the **MAKEPOINT()** macro. Thus we can use these like so:

```
POINT pt;                    //POINT structure
pt = MAKEPOINT(lparam);      //Macro to convert the lparam value into a point
```

Of course you can always do it by typecasting the **lparam** to HIWORD and LOWORD to extract the information.

```
POINT pt;                    //POINT structure
pt.x = HIWORD (lparam);
pt.y = LOWORD (lparam);
```

Simple enough. Now let's move on to processing mouse inputs while pressing other modifiers. These include holding down other mouse keys, the shift key, or the control key. These keys are stored in the **wparam** parameter.

Table 3-3-2	**Messages**
Mouse key values	Description
MK_CONTROL	Control key is down
MK_LBUTTON	Left button is down
MK_MBUTTON	Middle button is down
MK_RBUTTON	Right button is down
MK_SHIFT	Shift key is down

Note: Checking these things are confusing. It is best to use the **bitwise and** operator (&) with each of the ones necessary using a nested loop. Bitwise operators can be confusing to use so you must be extremely careful. They sort of do not work in the same way as

logical operators. There will be an example later to show you how to use multiple conditions checking.

Double clicks messages are kind of confusing. They are not isolated messages. Remember that windows will only process them if you have CS_DBLCLKS style defined in your **win class styles** attribute. Windows sends a double click message only on the second mouse button down message, which must be within a short time period called **double-click speed**. Because of this, it is sometimes not a good idea to process messages like left mouse button down and double click messages with totally different behaviors. When we receive a mouse button down message, we do not know whether or not it is a single mouse button down event or the first of a double click message. Unless, that is of course, we introduce a delay. But this approach is confusing, so you don't need to worry. Instead, you can make it so that a double-click message be a part of a continuation of a single mouse button down message.

As you already know, to process the mouse messages you must add a case in your event handler.

```
//All of this should be in the switch msg structure that we have created
case WM_LBUTTONDOWN:
        ;       //Do action here, which can include stuff like checking for current
                //position, conjunction keys pressed, etc.
```

And we would add other cases to check for the other conditions.

To clear up any confusion, here is a short example of checking for mouse input. Assume that the user has pressed down the left and right mouse buttons while holding down the shift key.

```
//All of this should be in the switch msg structure that we have created
case WM_LBUTTONDOWN:
        POINT pt;                       //POINT structure
        pt = MAKEPOINT(lparam);         //Macro to convert the lparam value into a point

        if(wparam & MK_RBUTTON)
                if(wparam & MK_SHIFT)
                        //The only way to get in here is when the left mouse button,
                        //the right mouse button, and shift key are down
                        ;       //Do action here
```

Finally, we are done with processing input messages.

Displaying Text

Displaying text is really simple. All you really need to know is a simple **TextOut()** function. Here is the prototype:

```
BOOL TextOut(
  HDC  hdc,                     // handle of device context
  int  nXStart,                 // x-coordinate of starting position
```

```
  int  nYStart,                 // y-coordinate of starting position
  LPCTSTR  lpString,            // address of string
  int  cbString                 // number of characters in string
);
```

Get it? Okay, I will explain briefly. Basically, you supply the function with the device context, the position that the text is supposed to be displayed, the text, and the numbers of characters in the text. The function will return a 1 if successful and a 0 if failed. Here is an example:

```
TextOut(hdc, 100, 100, "Hello World!", 13);
```

You can also store text in a string and then use this function to display them. Here is an example.

```
char buffer[80];                    //Create a character array
buffer = "Hello World!";            //Store something in to an array
TextOut(hdc, 100, 100, buffer, sizeof(buffer));    //Use the sizeof operator
```

It's simple as that for basic displaying text.

Message Box

A message box is a window that contains a small piece of caption that asks the user to respond by clicking pushbuttons. Simple message boxes are easy to generate. The function **MessageBox()** returns a value that identifies the button pressed. Here is the prototype:

```
int MessageBox(
  HWND  hwndOwner,          // handle of owner window
  LPCTSTR  lpszText,        // address of text in message box
  LPCTSTR  lpszTitle,       // address of title of message box
  UINT  fuStyle             // style of message box
);
```

The **hwndOwner** parameter identifies the window that owns the message box window. Normally, you would set this to the handle of the window making the function call. The **lpszText** and the **lpszTitle** are null-terminated strings used for the caption and title of your message box window. The style of the message box deals with the icons and options you wish to include in the message box. Thus the **fuStyle** can be any of these:

These bit flags set the push buttons that would appear in a message box.

Table 3-5	Message Box Window Styles
Style	Description
MB_YESNO	MB contains yes no
MB_YESNOCANCEL	MB contains yes no cancel

MB_RETRYCANCEL	MB contains retry cancel
MB_OKCANCEL	MB contains ok cancel
MB_OK	MB contains ok
MB_ABORTRETRYIGNORE	MB contains abort retry ignore

These bit flags set the different icons that can be associated with your message box:

Table 3-5-2	**Message Box Window Styles**
Icons	Description
MB_ICONINFORMATION	Icon with letter i
MB_ICONEXCLAMATION	Icon with !
MB_ICONQUESTION	Icon with ?
MB_ICONSTOP	Icon with stop sign

Finally, these bit flags sets the default selection of the message box:

Table 3-5-3	**Message Box Window Styles**
Default Selection	Description
MB_DEFBUTTON1	Makes first button default
MB_DEFBUTTON2	Makes second button default
MB_DEFBUTTON3	Makes third button default

Note: You can combine these styles from the tables in any way. However, you must not combine two elements in the same table. To combine the styles, you must use the **bitwise or** operator (|).

Now let's look at the return values of the **MessageBox()** function. They can be:

Table 3-5-4	**Message Box Window Styles**
Selection ID	Description
IDABORT	Abort was selected
IDCANCEL	Cancel was selected
IDIGNORE	Ignore was selected
IDNO	No was selected
IDOK	Ok was selected
IDRETRY	Retry was selected
IDYES	Yes was selected

Note: Pressing the escape key acts as choosing the cancel option if that push button has been declared in the message box styles attribute parameter.

Simple? Let's try to make a message box of our own. It will be a short exit message box, with ok and cancel push buttons, default button is the cancel button, and will have a question mark. Anyway here is the way to set it up:

```
int result;        //Variable to hold return value of the MessageBox function call
```

```
result = MessageBox(hwnd, "Exit?","Exit Message Box", MB_OKCANCEL |
                    MB_ICONQUESTION | MB_DEFBUTTON2);
//Check which button was pressed
if(result == IDCANCEL)
        ;       //Perform Action
if(result == IDOK)
        ;       //Perform Action
```

This is the basic approach to generating an extremely simple type of dialog box.

This entire chapter has covered a lot of ground. If you feel uneasy, it's normal. Don't freak out. Just glance back to understand the basic material covered. It is not expected of you to remember all the values of the tables that have been listed. All you need to understand are the basic concepts.

With those things out of the way, let's take a look at some system basics in the next chapter.

System Basics

In this chapter, we are going to be working some advanced system stuff. These stuff deals with retrieving the systems information.

System Metrics

Many times we want to be able to check information about the Windows environment. Windows provide us with two functions: **GetSystemMetrics()** and **GetDeviceCaps()**. **GetSystemMetrics()** retrieves Windows metrics and **GetDeviceCaps()** returns information about specific device context capabilities. **GetDeviceCaps()** will not be covered in this text.

Since **GetSystemMetrics()** is the easiest to use, we will only use **GetSystemMetrics()** to retrieve many Windows parameters and metrics. For instance we can retrieve the height of a window caption, size of cursors and icons, width and height of window borders, etc.

Here is the function prototype:

```
int GetSystemMetrics(int);
```

This by far is very simple to use. All we need to do is pass a constant symbolic name in the parameter and the function will return the specified metric. Measurements are returned in units of pixels or on and off flags.

Here is a table of some common system metric indices. These will return various widths associated with the Windows environment.

Table 4-1 System Metric Indices

Symbolic names	Description
SM_CXBORDER	Width of a window frame that can't be sized
SM_CXCURSOR	Width of cursor
SM_CXDLGFRAME	Width of a dialog frame
SM_CXFRAME	Width of a window frame that can be sized
SM_CXFULLSCREEN	Width of a full screen window
SM_CXHSCROLL	Width of the arrow bitmap on H-scroll bar
SM_CXHTHUMB	Width of the thumb box on the H-scroll bar
SM_CXICON	Width of an icon
SM_CXMIN	Minimum width of a window
SM_CXMINTRACK	Minimum tracking width of a window
SM_CXSCREEN	Width of the screen
SM_CXSIZE	Width of the title bar bitmaps
SM_CXVSCROLL	Width of the arrow bitmap on V-scroll bar

These are the basic constants associated with the various screen widths. You can uses these things to calculate the borders of a window, to determine the client area of a screen and determine where to draw on the screen, etc.

The next table will include the common system metric indices indicating the various heights associated with the Windows environment.

Table 4-1-2	System Metric Indices
Symbolic names	Description
SM_CYBORDER	Height of window frame that can't be sized
SM_CYCAPTION	Height of a window caption
SM_CYCURSOR	Height of cursor
SM_CYDLGFRAME	Height of a dialog frame
SM_CYFRAME	Height of window frame that can be sized
SM_CYFULLSCREEN	Height of a full screen window
SM_CYHSCROLL	Height of the arrow bitmap on H-scroll bar
SM_CYICON	Height of an icon
SM_CYMENU	Height of the menu
SM_CYMIN	Minimum height of a window
SM_CYMINTRACK	Minimum tracking height of a window
SM_CYSCREEN	Height of the screen
SM_CYSIZE	Height of the title bar bitmaps
SM_CYVSCROLL	Height of the arrow bitmap on V-scroll bar
SM_SYVTHUMB	Height of the thumb box on V-scroll bar

Here are other miscellaneous symbolic names associated with the **GetSystemMetrics()** function:

Table 4-1-3	System Metric Indices
Symbolic names	Description
SM_DEBUG	Non-zero flag indicating Windows debug mode
SM_MOUSEPRESENT	Non-zero flag indicating if mouse exist
SM_SWAPBUTTON	Non-zero flag indicating if left and right mouse buttons are swapped
SM_CMETRICS	Count of system metrics

Anyway here is how you would use it. Simply call the function with the desired system metric index and store the return value in an int data type.

```
int cyFullScreen = GetSystemMetrics(SM_CYFULLSCREEN);
```

That's all there is to using the **GetSystemMetrics()** function. There are few ways we can use these information to customize our applications. For instance, we can use this to set our window size.

Full Screen Apps

We can use the system metrics in order to create full screen applications. First let's revisit the **CreateWindowEX()** function, and also our **CreateWindowEX()** function call.

```
HWND CreateWindowEx(
   DWORD  dwExStyle,              // extended window style
   LPCTSTR  lpszClassName,        // address of registered class name
   LPCTSTR  lpszWindowName,       // address of window name
   DWORD  dwStyle,                // window style
   int  x,                        // horizontal position of window
   int  y,                        // vertical position of window
   int  nWidth,                   // window width
   int  nHeight,                  // window height
   HWND  hwndParent,              // handle of parent or owner window
   HMENU  hmenu,                  // handle of menu, or child-window identifier
  HINSTANCE  hinst,               // handle of application instance
  LPVOID  lpvParam                // address of window-creation data
  );

hwnd = CreateWindowEx(      NULL,
                            WINDOW_CLASS_NAME, "Graphics App",
                            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                            0, 0,
                            800, 600,
                            NULL,
                            NULL,
                            hinstance,
                            NULL);
```

In order to create a full screen window, we must specify a valid width and height for the screen resolution. Therefore, we can replace the width and height with:

```
width = GetSystemMetrics(SM_CXSCREEN);
height = GetSystemMetrics(SM_CYSCREEN);
```

We must also change the extended window styles to use WS_POPUP and WS_VISIBLE instead of WS_OVERLAPPEDWINDOW. This creates a window without borders and controls and thus looks like a blank screen.

The result: the desired effect of a full screen application.

Sending Messages

Sending your own messages is very important in Windows. Recall that a message is sent automatically when an event occurs such as clicking the mouse, pressing a key, etc. You,

however, can send your own messages and allow it to be processed in the event queue by Windows. There are two ways to do this: **SendMessage()** and **PostMessage()** functions.

SendMessage() sends a message to window immediately for processing. This function will return only after **WinProc()** has finished processing the message.

PostMessage() sends a message to windows through the event queue and returns immediately. The message has a low priority.

Here are the prototypes of the functions:

```
LRESULT SendMessage(
   HWND  hwnd,              // handle of destination window
   UINT  uMsg,              // message to send
  WPARAM  wParam,           // first message parameter
  LPARAM  lParam            // second message parameter
 );

BOOL PostMessage(
   HWND  hwnd,              // handle of destination window
   UINT  uMsg,              // message to post
  WPARAM  wParam,           // first message parameter
 LPARAM  lParam             // second message parameter
);
```

If **PostMessage()** is successful it return a nonzero value. **SendMessage()** actually calls the **WinProc()** function so it returns the actually value of the **WinProc()** function.

Just like having to check for the event messages in the event handler, we can call those things ourselves with these functions. For instance:

```
SendMessage(hwnd, WM_DESTROY,0,0);
```

This will cause a WM_DESTROY message to be executed by the program, and thus end the program. In a better context, we can use something like this to detect when a key has been pressed and carry out a specific action. Recall the **KEYDOWN()** macro:

```
#define KEYDWN(vk_code)  ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)

if(KEYDWN(VK_ESCAPE))
        SendMessage(hwnd, WM_CLOSE,0,0);
```

Finally, the basics of Windows systems have been covered.

Using Resources

Resources are simple pieces of data that are combined with the program during linking and can be loaded during runtime by the program. Resources can be icon files, menus, bitmaps, sounds, etc.

Why use resources? There are lots of reasons as to why resources are used. For instance, we can create a single .EXE that contain both the code and the data. This makes it more organized and easier to manage. Another thing is that it doesn't let other people access your files and modify them or copy them.

In order to use them, you will need a resource compiler. The file .RC is the resource file made. During compilation of the resources, a .RES file is created that is contains all the binary data making up all the resources. If you link this file with your .CPP, .H, .OBJ, .LIB, etc you would create your .EXE. Most compilers have linking and compiling of resources built in. I would recommend using Visual C++ because it contains built in resource editors as well. You can use, any program you want as long as it supports the export format.

Icons

Icons are just bitmaps, usually 32 by 32 pixels. However it can range from 16 x 16 to 64 x 64 pixels. It can also support 16 colors to 256 colors.

First create the resource. Use any editor you want as long as .ICO files can be created. Supposedly, you can just rename the extension of .BMP created in a paint program, but some compilers will not accept the format. Besides, for others, you get more functionality if you use the resource editor provided by the compiler.

In the .RC script file, you would place the following:

```
iconName ICON filename.ICO
```

Thus, these are valid statements:

```
icon_name  ICON   cross.ICO
winicon    ICON   arrow.ICO
```

The compiler will assume that you are defining a string and thus will refer to your icon as "icon_name". If you want it to be treated as an integer ID, you must define it before hand. It is important to put this in an .H file, as you must also include this in your main program.

```
#define icon_name 100  //Numbers are arbitrary
icon_name ICON   cross.ICO
```

Note: This will set icon_name to be associated as a integer ID and not as "icon_name". Without the defines, it will be associated as a string. This means that in the program, you would refer to it as "icon_name" instead of icon_name.

Confusing enough! Now back to loading a simple icon. Here is the example:

In the .RC file you should have something like this:

```
icon_name ICON filename.ICO
```

In the program code, during the loading of the icon part of initializing the window class attributes:

```
winclass.hIcon = LoadIcon(hinstance, "icon_name");
winclass.hIconSm = LoadIcon(hinstance, "icon_name");
```

And if you want to use the integer ID way to load the icon:

In an .H file you should have something like this:

```
#define icon_ID1        100
#define icon_ID2        101
```

In the .RC you should have something like this:

```
#include "headerfile.H"          //Header file with the defines

icon_ID1 ICON filename.ICO
icon_ID2 ICON filename2.ICO
```

In the program code, during the loading of the icon part of initializing the window class attributes:

```
#include "headerfile.H"          //Header file with the defines

winclass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(icon_ID1));
winclass.hIconSm = LoadIcon(hinstance, MAKEINTRESOURCE(icon_ID2));
```

Simple right? Just remember that when using integer ID's be sure to use the **MAKEINTRESOURCE()** macro to convert the integer into a string pointer.

These things aren't limited to that either. In fact, you can use this icon object you created for other functions too. The small icon is what appears on the title bar as well as in the start menu. The big icon appears in the window view in folders or on the desktop.

Make sure you have all these files saved and put them all into the project to compile.

Cursors

Cursors are very much similar with icons. They are usually 32 by 32 pixels. However it can range from 16 x 16 to 64 x 64 pixels. It can also support 16 colors to 256 colors. Some are even animated.

First create the resource. Use any editor you want as long as .CUR files can be created. You can use the compiler's own resource editor to create your own cursor files because you can define the hot spot of the cursor and so on.

Anyway, after creating the cursor, you need to define the cursor in the .RC file using the CURSOR keyword.

```
cursor_name CURSOR filename.CUR
```

Same as icons, you can use integer ID's to create cursors.

Let's take a look at how to load the cursors during initialization.

In the .RC file you should have something like this:

```
cursor_name CURSOR filename.CUR
```

In the program code, during the loading of the cursor part of initializing the window class attributes:

```
winclass.hCursor = LoadCursor(hinstance, cursor_name);
```

And if you want to use the integer ID way to load the cursor:

In an .H file you should have something like this:

```
#define cursor_ID1      100
#define cursor_ID2      101
```

In the .RC you should have something like this:

```
#include "headerfile.H"          //Header file with the defines

cursor_ID1 CURSOR filename1.CUR
cursor_ID2 CURSOR filename2.CUR
```

In the program code, during the loading of the cursor part of initializing the window class attributes:

```
#include "headerfile.H"          //Header file with the defines
winclass.hCursor = LoadCursor(hinstance, MAKEINTRESOURCE(cursor_ID1));
```

Simple right? Just remember that when using integer ID's be sure to use the **MAKEINTRESOURCE()** macro to convert the integer into a string pointer.

We can also mess with the cursors at the window level. Here are the function prototypes for two functions we can use:

```
HCURSOR LoadCursor(
   HINSTANCE  hinst,  // handle of application instance
   LPCTSTR  lpszCursor // name string or cursor resource identifier
  );

HCURSOR SetCursor(
   HCURSOR  hcur                // handle of cursor
  );
```

Create a cursor by doing something like so:

```
HCURSOR myCursor = LoadCursor(hinstance, "cursor_name");
```

Then, use the newly created cursors with the **SetCursor()** function.

```
HCURSOR oldCursor;
oldCursor = SetCursor(myCursor);
```

By the way, let us take a look at some defaults cursors:

Table 5-2 System Cursors

Symbolic names	Description
IDC_APPSTARTING	Standard arrow and small hourglass
IDC_ARROW	Standard arrow
IDC_CROSS	Crosshair
IDC_IBEAM	Text I-beam
IDC_ICON	Empty icon
IDC_NO	Slashed circle
IDC_SIZE	Four-pointed arrow
IDC_SIZENESW	Double-pointed arrow pointing northeast and southwest
IDC_SIZENS	Double-pointed arrow pointing north and south
IDC_SIZENWSE	Double-pointed arrow pointing northwest and southeast
IDC_SIZEWE	Double-pointed arrow pointing west and east
IDC_UPARROW	Vertical arrow
IDC_WAIT	Hourglass

That's all there is to making simple cursors and loading them into the program.

Sounds

Windows only support .WAV formats so that is all I can cover. Custom resources can be made but this is sufficient for this text.

Once again, we can create WAVE formats just like icons and cursors.

```
wave_name WAVE filename.WAV
```

Same as icons, you can use integer ID's to create wave objects.

In the .RC file you should have something like this:

```
wave_name WAVE filename.WAV
```

This is using the string name method. To use the integer ID you would do something like so:

In an .H file you should have something like this:

```
#define wave_ID1        100
#define wave_ID2        101
```

In the .RC you should have something like this:

```
#include "headerfile.H"          //Header file with the defines

wave_ID1 WAVE filename1.WAV
wave_ID2 WAVE filename2.WAV
```

There is a simple function that will load the wave object for us:

```
BOOL PlaySound(
   LPCTSTR lpszName,          // sound string
   HANDLE hModule,           // sound resource
   DWORD fdwSound             // sound type
  );
```

Note: lpszName is the string name or integer ID of the wave object. It can also be the filename on the disk. Remember that when using integer ID's be sure to use the **MAKEINTRESOURCE()** macro to convert the integer into a string pointer. hModule is simply the instance of the application to load the resource from. fdwSound sets the options of how the sound is played.

The table will describe some of the most common options in configuring how the sound is played:

Table 5-2	System Cursors
Symbolic names	Description
SND_RESOURCE	Set as resource
SND_SYNC	Synchronous playback
SND_ASYNC	Asynchronous playback
SND_LOOP	Loop playback
SND_PURGE	End all sound

Note: To combine the styles, we must use the **bitwise or** operator (|).

Note: You should not combine some together because it is one or the other. For instance, SND_ASYNC should not be combined with SND_SYNC.

Here are examples of using the **PlaySound()** function:

```
PlaySound("wave_name", hinstance, SND_ASYNC | SND_RESOURCE);
PlaySound(MAKEINTRESOURCE(wave_ID1), hinstance,
                SND_ASYNC | SND_LOOP | SND_RESOURCE);
```

That's all there is to using simple sounds.

Conclusion

Well, that's about all that I feel is necessary to talk about. Sorry if I omit too many stuff. It is really hard to decide which stuff is important to add and which part is too difficult to discuss. I wanted to add stuff like adding menus, and the use of bitmaps and fonts, key accelerators, dialog boxes, and lots more but those issues require extremely long and detailed explanations and length code. The complexity of these things can be overwhelming (even in the simplest reference guides and help files these things require 20-50 pages to discuss). I really wished that I could have added such awesome features, but those things will make this text extremely lengthy and defeats the purpose of this text all together as a simple reference and starter guide.

I can only hope that this text actually helps others. I hope that you enjoyed reading this text and I also hoped that this text have inspired you to creating cool programs in Windows.