WINDOWS PROGRAMMING

Windows Programming Concepts

- There are two models that multitasking environments use to share CPU resources:
    - **Pre-emptive model**: The operating system has the power to suspend one application's use of CPU time in anticipation that another application may require the CPU resource. In the case of Win32, each active process or thread is assigned a CPU time slice.
    - **Non pre-emptive model**: The active program (application) has full control of the CPU and system resources and will not release either until the end of the program execution. (DOS)
- There are two ways in which we can write programs for Windows.
- The first is to utilize the Win32 API functions defined by Windows.
- The second is to use a C++ class library that encapsulates the API.
- A very popular class library is MFC (Microsoft Foundation Classes).
- There are two key terms that need to be understood first: **Event** and **Message**.
- An **Event** in Windows is an occurrence in the system, or a change of state in the system. Examples are clicking on a mouse button or any system interrupts such as the arrival of a character into the serial port.
- A **Message** is information about an event that is generated by Windows when an event occurs. Every event has its own message. For example, the message that is created by Windows, **WM_LBUTTONDOWN**, when the left mouse button is pressed, might contain information such as the x and y co-ordinates, the active application and the system time.
- Programming windows applications is generally a task of writing procedures to respond to messages generated by Windows.
- Event driven programming is quite different from the sequential (DOS) programming:
    - Programmers must be aware of over 100 messages and as many as 50 system functions to program a simple application.
    - The flow of the program is dynamic and user driven.
    - The programmer must anticipate the messages that Windows generates by programming separate code for each possible message.
- Functions known as **Message Handlers** are called by the application program to handle messages forwarded by windows.

- An application can either process a received message or pass it back to Windows for default processing.

- The message is retrieved by the **WinMain** message loop. **WinMain** is responsible for translating incoming messages and dispatching them to the application's message procedure.

- The message procedure, often called **WndProc** is called by windows to pick up the message, identify it by its number **(Handle**) and call a message handler to process it.

- A **Handle** is an integer that contains a number assigned to each object. If the object type and number are known, the instance of the code object (active application) can be identified. Note that in Windows we can have several objects (applications) open at any time but only one is active.

- Once the object's Handle has been identified, the object's properties can be accessed.

- For example, assume a user selects a pull down menu item from application's window by pointing to the menu with the mouse and dragging down to select.

- Because Windows treats the pull down menu as an object that is related to your window, it sends a message containing a handle tied to the window that has the pull down menu item you selected.

- Thus, an event is tied to a message, a message is tied to a handle, and a handle is tied to an object.

- The diagram shown illustrates how messages are processed in Windows.

- Any device function must first pass through Windows.

- For example, Windows will respond to a mouse event as follows:

    1. The user points to an application and clicks the mouse button.
    2. Windows determines to which window the mouse pointed.
    3. Windows generates a mouse click event message and posts it to your application's **WinMain()** procedure message queue.
    4. **WinMain()** takes the message and instructs Windows to send it to the procedure that handles events for the Window selected.

       More specifically, the message is sent to the message loop that passes the message on to the **WndProc()**, which decides what to do with the message.

    5. The application function that handles messages for that window then processes the message and responds according to the menu item selected.

Basic Concepts

- Thus there are two main parts in a program. The first is the **WinMain()** procedure; the second is the **WndProc()** procedure.
- **WinMain()** has some special properties that differentiate it from other functions:
    - It must be compiled using the **WINAPI** calling convention.
    - The return type for **WinMain** must be **int**.
- The **WinMain** procedure has three parts. The first ***registers*** the Window, the second ***creates*** the window, and the third sends ***messages*** to the window.
- Before Windows can execute a Win32 program it must **define** and **register** a window class ("class" here is a style or type rather than a class in the C++ sense).
- Registering a window class sets a form and function of the window.
- All Win32 applications must establish a message loop inside the **WinMain()** function.
- This loop reads pending messages from the application's message queue and dispatched them to the program's window function **(WndProc** in our example).
- Any messages sent to a window pass through **WndProc**. This window function can explicitly handle the messages, or pass them on to **DefWindowProc**, which is the default message handler.
- This window function (and all Window functions) must be declared as returning type **LRESULT CALLBACK**.
- The **CALLBACK** calling convention is used with those functions that will be called by Windows. Therefore such functions are referred to as ***callback*** functions.

Windows Data Types

- Win32 API functions in general do not make use of the standard C/C++ data types. Instead they make use of many data types that have been ***typdef***'ed within the windows.h file
- Some common types are:
    - **HANDLE** (32-bit integer used to identify a resource)
    - **HWND** (32-bit integer used as a window handle)
    - **BYTE** (8-bit unsigned integer)
    - **WORD** (16-bit unsigned short integer)
    - **DWORD** (32-bit unsigned integer)
    - **UINT** (32-bit unsigned integer – left over from old 16-bit Windows)
    - **LONG** (32-bit signed integer)
    - **BOOL** (integer)

- **LPSTR** (pointer to a string)
- **LPCSTR** (const pointer to a string)

- In addition, Win2K defines several structures:

  - **MSG** (holds a Win2K message)
  - **WNDCLASSEX** (defines a window class)

- Study the code provided. The following lines of code highlight some of the key issues:

  Lines 18 -26:

  This is where the window is registered.

  A pointer to the structure of type **WNDCLASS** is created.

  Default values are assigned to the **WNDCLASS** structure.

  The structure is passed to Windows with the Windows **RegisterClass()** function.

  Lines 29 - 45:

  This is where the window is created and displayed.

  The first step is to call the **CreateWindow()** function which uses the base information from the registered class and returns a handle to the window being created.

  The second step is call the functions **ShowWindow()** and **UpdateWindow().**

  ShowWindow() shows or displays the window on the screen. It can also be used to hide a window. It requires two parameters, the handle to the window, and a parameter specifying what to do with the window.

  UpdateWindow() forces Windows to send an update window message (**WM_PAINT**) to the application. This allows you to repaint any screen information that was destroyed when the window was hidden.

  Lines 48 - 52:

  This is the central message loop which keeps repeating throughout the life of the program.

  When the user strikes a key or clicks on the mouse buttons, messages are sent to this message loop via a message queue.

  Recall that Windows delivers messages to the routines you have assigned to a particular window.

Windows gives your application the option of preprocessing the message before Windows calls your message-processing routine.

The **GetMessage()** function contains a *WHILE* loop and is the application's means of receiving raw messages from Windows. It reads the messages from the message queue.

TranslateMessage() is a function invoked by the application to take the raw message received by Windows and convert the "key-board pressed" (abstract form) to "character entered" (easy to understand) form.

DispatchMessage() function posts the translated message to the application's window-processing procedure.

The **DispatchMessage()** function tells Windows to send the message to the procedure that you assigned to handle messages for the window involved. In this case it happens to be **WndProc()**.

Lines 57 - 69:

This is the procedure specified to handle messages for this window.

This program only responds explicitly to **WM_DESTROY** messages. All the other messages are passed on to **DefWindowProc()**.

The first argument to the function is the handle to the window affected.

The second argument contains the message being sent. These are all Windows predefined statements defined in the header file **windows.h**.

The third and fourth arguments are short and long integers respectively, which contain additional information depending on the message sent.

All messages that a window procedure chooses not to process must be passed to a Windows function named **DefWindowProc()**.

The **DefWindowProc()** merely handles the default behavior associated with a window.

In this example **WndProc()** chooses to process only the **WM_DESTROY** message. It calls **PostQuitMessage()** and exits with a value 0.

All other messages are handled by **DefWindowProc()**. It is essential to call **DefWindowProc()** for default processing of all messages that your window procedure does not process.

Hungarian Notation

- Named after Microsoft programmer Charles Simonyi.
    - This variable naming convention helps avoid coding error with data types and makes it easier to read Windows code.
    - This is basically a convention adopted by programmers to allow them to remember the context of variable names long after coding.
    - For example, in C, a variable name given to a string containing an error message might be **message666**.
    - In Hungarian notation, the same variable might be called **lpszSerialPortErrorMsg**. This is translated as follows:

      lpsz is the data type (long pointer to a string terminated by a zero).

      SerialPortErrorMsg is a description of the contents of the string (long descriptive names are in!).

- The value of this notation has somewhat been diminished by the **STRICT** preprocessor directive. It is still nevertheless an established part of the Windows coding tradition.
- The table below shows the Hungarian definitions in **windows.h**

Prefix	Type	Windows Type
b	int	BOOL
by	unsigned char	BYTE
c	char	
dw	unsigned long	DWORD
fn	function	
h	unsigned int (UINT)	HANDLE
i	int	
l	long	LONG
lp	long pointer	
n	int or short	
p		
pt		
S	string	
sz	null-terminated string	
w	unsigned short	WORD
lpsz	Long pointer to null-terminated string	

- The third column shows the various new types that are frequently used in Windows programs.