Message Handlers

- How does Windows know what messages for that particular windows (class) are handled by which handler?

- When you register your window's class, you explicitly state that all messages for your main window will be processed by the procedure specified in the **lpfnWndProc** field of the **WNDCLASS** structure.

- A Windows program can contain more than one window procedure. But each window procedure is always associated with a ***particular window class*** which you register by calling **RegisterClass**.

- For example, in the code provided in your notes, we explicitly set

  Wcl.lpfnWndProc = WndProc

  which will result in Windows calling the function **WndProc** when it receives messages in its application queue.

- Note that **WndProc** just handles (in this case) the **WM_DESTROY** message. Everything else is handled by **DefWindowProc**, the Windows default message handler.

- **DefWindowProc** knows how to handle nearly all the default behavior associated with a particular type of window. For instance, it knows how to handle mouse movements that minimize or maximize a window, as well as how to handle movements that expand or shrink a window.

- For example, we can explicitly set the main message handler to **COMTermWndProc** (the program you will be writing for serial communications) as follows:

  Wcl.style = CS_HREDRAW | CS_VREDRAW;
  Wcl.lpfnWndProc = COMTermWndProc;
  Wcl.hInstance = hInst;
  Wcl.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
  Wcl.lpszClassName = Name;
  RegisterClass (&Wcl);

- Note that **COMTermWndProc** becomes the main function for processing all the Windows messages and calling the appropriate functions to perform the various functions for serial communications and terminal emulation.

- In turn, **COMTermWndProc** will then as part of its processing will call **DefWindowProc** to process all default window actions.

- Each message is assigned a numerical value. Windows also provides macro names for all Windows messages.

- The standard names for messages are defined by including ***windows.h*** in your program.

- The following are some common Windows message macros:

WM_CHAR	WM_PAINT	WM_MOVE
WM_CLOSE	WM_LBUTTONUP	WM_LBUTTONDOWN
WM_COMMAND	WM_SCROLL	WM_SIZE

- Note that all functions called by the message processing loop in **WinMain** must be declared using the **CALLBACK** definition.
- The message is comprised of two 32-bit values that contain information related to each message. These are: **wParam** and **lParam**.
- These values hold things such as mouse coordinates or the value of a keypress.
- To provide easy access to the low-order and high-order words of these 32-bit values, Windows provides two macros:

```
x = LOWORD (lParam);
y = HIWORD (lParam);
```

RESOURCES

- Resources are used to add various features to Windows programs.
- This is a fast and easy way to add "plug-and-play" features to your programs with minimal coding.
- There several major resources being used in Windows programs:
    - **Menus** - Usually along the top of a main window which provide iconic options to manipulate a program.
    - **Dialog Boxes** - Special windows containing edit boxes, buttons, radio buttons, check boxes, and other controls used to enter data or select features.
    - **Bitmaps** - These are graphical objects (**.BMP**) that can be painted onto a window or dialog.
    - **Icons** - Like bitmaps but with predefined sizes. These are visible on the taskbar when a program is minimized.
    - **String Table** - A list of strings kept in easy-to-use format. Used as error messages for example.
    - **Fonts** - Character sets used to draw text or symbols on-screen.

    - **Cursors** - Locates the mouse on the screen (Arrow, hourglass, and I-beam), these can also be customized by programmers.

- Resources are objects which are created separately from your program and are linked to the executable at compile time.
- Resources are contained in **Resources Files** which have a **.RC** extension. The filename is usually the same as the name of the executable.
- Resource can be created using standard editors (text resources) or using special resource editors (for icons, etc.).
- The special **script** or **resource language** must be compiled using a resource compiler which creates a **.RES** file which is linked to the main program.

MESSAGE BOXES

- The simplest interface window which displays a message to the user and waits for a response.
- The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.
- The API function for creating a message box is **MessageBox ():**

int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);

    - **hWnd** Identifies the parent window of the message box to be created.
    - **lpText** is a pointer to a null-terminated string containing the message to be displayed inside the message box.
    - **lpCaption** is a pointer to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.
    - **uType** specifies the type of buttons displayed and behavior of the dialog box.

- The table below shows some common values of the **uType** parameter:

VALUE	EFFECT
MB_ABORTRETRYIGNORE	The message box contains three push buttons: **Abort**, **Retry**, and **Ignore**.
MB_CANCELRETRYCONTINUE	The message box contains three push buttons: **Cancel**, **Retry**, and **Continue**.
MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
MB_ICONHAND	A stop-sign icon appears in the message box.
MB_ICONERROR	Displays a stop sign icon.
MB_ICONSTOP	Same as MB_ICONHAND.
MB_ICONINFORMATION	An icon consisting of a lowercase letter “i” in a circle appears in the message box.
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_OK	The message box contains one push button: OK.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.

- If the **MessageBox** function succeeds, the return value is one of the following menu-item values returned by the dialog box:

Value	Meaning
IDABORT	Abort button was pressed.
IDRETRY	Retry button was pressed.
IDCANCEL	Cancel button was pressed.
IDIGNORE	Ignore button was pressed.
IDCONTINUE	Continue button was pressed.
IDNO	No button was pressed.
IDOK	OK button was pressed.
IDRETRYAGAIN	Retry Again button was pressed.
IDYES	Yes button was pressed.

- The code provided (**winmenu.cpp**) illustrates the use of the **MessageBox()** function.

- The **WndProc** function is now a little more involved in the processing of left and right mouse buttons.

- Pressing the left mouse button results in a message box with two choices: **Yes** and **No**.

- The response to the choices results in another message box being displayed together with the selected choice.

- Note that the response from the final message box is ignored resulting in no action and a continuation of program execution.

MENUS

- The appearance of the menu (its contents) are defined in a resource file as follows:

```
MenuName     MENU  [options]
{
        menu items
}
```

- **MenuName** is an arbitrary name assigned to the menu.

- The keyword **MENU** tells the resource compiler that a menu is being created.

- The keywords **MENUITEM** and **POPUP** are used to define the menu items within the menu.

- **POPUP** specifies a standard drop-down menu window below the menu item selected.
- **MENUITEM** specifies a single entry in a menu window. The string associated with **MENUITEM** is displayed in the menu entry.

- **POPUP** is an entire window (a pop-up list) whereas **MENUITEM** is a single entry in the list.

- A sample resource file (**winmenu2.rc**) is provided.

- The menu, called **MYMENU** has three top-level menu bar options: **TX/RX**, **Settings**, and **Help**.

- Selecting option "**TX/RX**" results in a pop-up list consisting of two more options: **TX** and **RX**.

- Option "**Settings**" has four more options in its pop-up list, and within the list items "**Port**" and "**Speed**" have their own pop-up lists with further options.

- In this way we can design very elaborate menus quickly and with minimal coding.

- The **&** symbol in a menu item causes the letter it precedes to become the shortcut key associated with that option.
- Note that there is a file called **winmenu2.h** included in the resource file.
- This file contains the macro definitions of the menu ID values. These are the various values that are returned as the menu items are selected.
- These ID values can be any integer values. Do not select values lower than 100 since they may have a different significance in a Windows program.
- Do not use a number twice unless you are certain you know what you are doing. Using the same number for two different menu items is bound to cause problems.
- The final example code provided (**winmenu2.cpp**) illustrates the use of menus and message boxes together.
- The first step is to specify the menu within the window's class structure.
- When a particular menu item is selected by the user, the constant associated with that menu item is sent to the program's **WndProc**.
- Both the menu item and its associated constant arrive packaged in the form of a **WM_COMMAND** messages.
- The value of **LOWORD (wParam)** corresponds to the menu item's ID constant specified in the appropriate **resource** and **.h** file.
- It now becomes simply a matter of incorporating a **switch** statement to determine which menu item was selected and take the specified action.
- Note that in an actual working application, the action associated with each menu item would be a function call which would carry out the required task.
- **Reading Assignment**: Read the section on "**Accelerator Keys**" in your textbook.