MODULE 1

CS010 606L06: ADVANCED SOFTWARE ENVIRONMENTS

STUDY CS ENGG

http://www.studycsengg.gq

WINDOW

Window is a rectangular area on the screen that receives user input and displays output in the form of text and graphics. In other words, a window is an entity which processes messages and has a visible rectangular area.

The most noticeable windows are:

1. Application windows.
2. Dialog box.

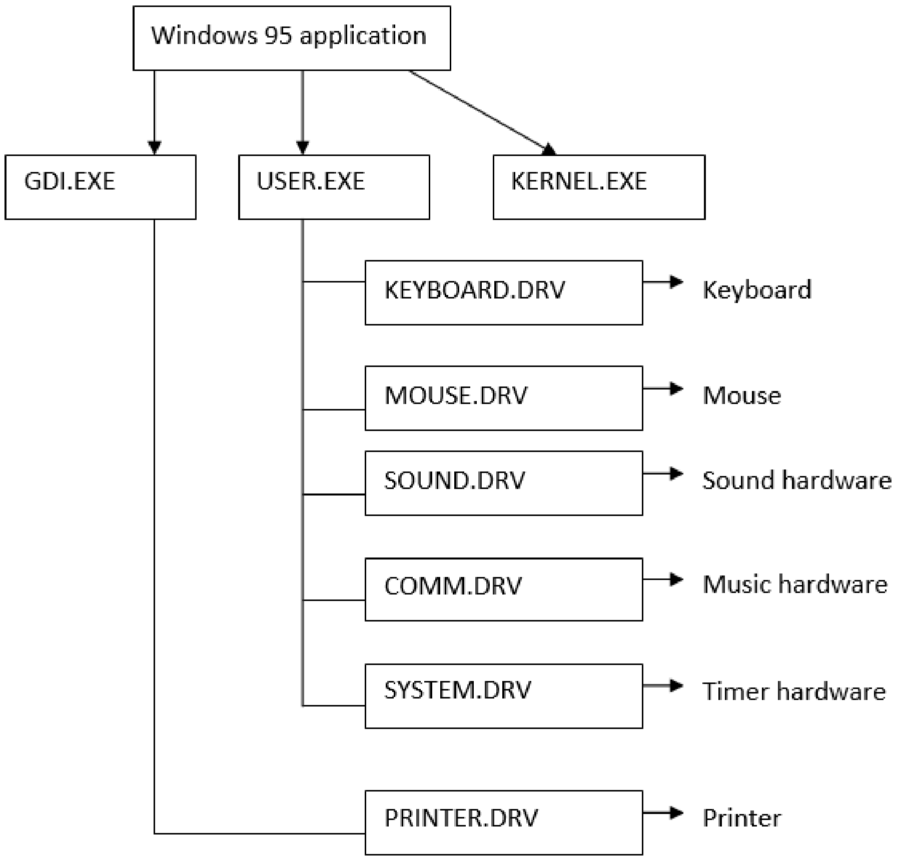
Windows programming

Programming for windows is a type of object oriented programming. Windows API is the name given by the Microsoft to the core set of application programming interfaces available in the Microsoft windows operating system. It is designed for usage by VC++ programs. An application developer can develop an application that calls the API functions.

The main object that we work for in the windows is the window. Window is a rectangular area that a particular program occupies on the screen. Although a single program can create many windows, usually the program has a single top level window. To the user this is the program’s workspace on the screen .The window is what windows uses to divide the screen in to separate programs.

From the programmers perspective a window is an object that receives and process messages .A window receives and processes user inputs. The message that a window receives contains information about an event (for example, a mouse event).

These messages are actually passed to the window’s window function. A window also uses messages to communicate with other windows. To receive and process messages we need a window procedure At this point we are writing a windows program.



Message Box:

Message Box function creates a window of limited flexibility. It has

1. a title bar
2. Close button
3. Optional Icon
4. One or more lines of text
5. Up to 4 buttons

Messages are send by the operating system to the window and vice versa. These messages inform the window about its visible state, its size and many other properties. We can resize windows and whenever there is a change in the size of the window, the window send message to the program indicating the new size. I.e. by invoking a function call in the program, called a window procedure.

Every window that a program creates has an associated window procedure. A window procedure is a function that is in the program or in the dynamic linked library. Windows sends a message to the window by calling the window procedure. It then does some processing and returns control to windows.

The main object that we work for in the windows is the window. Window is a rectangular area that a particular program occupies on the screen. Although a single program can create many windows, usually the program has a single top level window. To the user this is the program's workspace on the screen .The window is what windows uses to divide the screen in to separate programs.

WINDOWS API

The functionality provided by the Windows API can be grouped into seven categories:

1. **Base Services**
   Provide access to the fundamental resources available to a Windows system like file systems, devices, processes and threads, access to the Windows registry, and error handling. These functions reside in kernel.exe, krnl286.exe or krnl386.exe files on 16-bit Windows, and kernel32.dll and advapi32.dll on 32-bit Windows.

2. **Graphics Device Interface**
   Provide the functionality for outputting graphical content to monitors, printers and other output devices. It resides in gdi.exe on 16-bit Windows, and gdi32.dll on 32-bit Windows.

3. **User Interface**
   Provides the functionality to create and manage screen windows and most basic controls, such as buttons and scrollbars, receive mouse and keyboard input, and other functionality associated with the GUI part of Windows. This functional unit resides in user.exe on 16-bit Windows, and user32.dll on 32-bit Windows. Since Windows XP versions, the basic controls reside in comctl32.dll, together with the common controls (Common Control Library).

4. **Common Dialog Box Library**
   Provides applications the standard dialog boxes for opening and saving files, choosing color and font etc. The library resides in a file called commdlg.dll on 16-bit Windows, and comdlg32.dll on 32-bit Windows. It is grouped under the User Interface category of the API.

5. **Common Control Library**
   Gives applications access to some advanced controls provided by the operating system. These include things like status bars, progress bars, toolbars and tabs. The library resides in a DLL file called commctrl.dll on 16-bit Windows, and comctl32.dll on 32-bit Windows. It is grouped under the User Interface category of the API.

6. **Windows Shell**
   Component of the Windows API allows applications to access the functionality provided by the operating system shell, as well as change and enhance it.

7. **Network Services**
   Give access to the various networking capabilities of the operating system. Its sub-components include NetBIOS, Winsock, NetDDE, RPC and many others.

Web-related APIs:

The Internet Explorer web browser also uses many APIs that are considered as part of the Windows API. Internet Explorer has been an integrated component of the operating system since Windows 98, and provides web related services to applications. The integration will stop with Windows Vista.

It provides:

- An embeddable web browser control, contained in shdocvw.dll and mshtml.dll.
- The URL monitor service, held in urlmon.dll, which provides COM objects to applications for resolving URLs. Applications can also provide their own URL handlers for others to use.
- A library for assisting with multi-language and international text support (mlang.dll).
- DirectX Transforms, a set of image filter components.
- XML support (the MSXML components).
- Access to the Windows Address Book.

Multimedia-related APIs:

Microsoft has provided the DirectX set of APIs as part of every Windows installation since Windows 95 OSR2. DirectX provides a loosely related set of multimedia and gaming services, including:

- Direct3D as an alternative to OpenGL for access to 3D acceleration hardware.
- DirectDraw for hardware accelerated access to the 2D frame buffer. As of DirectX 9, this component has been deprecated in favor of Direct3D, which provides more general high-performance graphics functionality (as 2D rendering is a subset of 3D rendering).
- DirectSound for low level hardware accelerated sound card access.
- DirectInput for communication with input devices such as joysticks and gamepads.
- DirectPlay as a multiplayer gaming infrastructure. This component has been deprecated as of DirectX 9 and Microsoft no longer recommends its use for game development.

- DirectShow which builds and runs generic multimedia pipelines. It is comparable to the GStreamer framework and is often used to render in-game videos and build media players (Windows Media Player is based upon it). DirectShow is no longer recommended for game development.
- DirectMusic

APIs for interaction between programs:

The Windows API is concerned with the interaction between the Operating System and an application. For communication between the different Windows applications among themselves, Microsoft has developed a series of technologies alongside the main Windows API. This includes Dynamic Data Exchange (DDE), which was superseded by Object Linking and Embedding (OLE) and later by the Component Object Model (COM).

Distinction with ordinary programs

The main difference between Windows GUI programming and command line programming is that Windows itself is in control, and not the programmer.

There are three major differences in DOS based and Windows based programming:

- DOS programs have to bother about the details of the hardware they are Running on. Windows programs are not required to bother about the hardware being used. As a result, the programmer can know spend time in writing code for the application rather than for the hardware on which the application is going on to run. Also, the DOS programmers are under the constant fear that if the hardware on which the programs are running changes the program may crash. The Windows programmers are not bothered by the change in hardware because it is the Windows OS itself which takes care of this change.
- A DOS based program does a job a step at a time calling the OS as and when required. As against this, the Windows program waits till it receives a message from the Windows OS about the occurrence of an event, like say, hitting a key from a keyboard or clicking a mouse button.
- DOS never calls a program running under it. Contrary to this, Windows does call a program running under it. This calling is done directly by calling the window procedure which is part of every Windows program. Alternatively, Windows places messages in a queue notifying the program about the occurrence of an event. The messages from the queue are ultimately handled by the window procedure.

EVENT-DRIVEN PROGRAMMING

Event-driven programming or event-based programming is a computer programming paradigm in which the flow of the program is determined by user actions (mouse clicks, key presses) or messages from other programs. In contrast, in batch programming or flow-driven programming the flow is determined by the programmer. Batch programming is the style taught in beginning programming classes while event-driven programming is what is needed in any interactive program. Event-driven programs can be written in any language, although the task is easier in some languages than in others. I.e. instead of waiting for a complete command to make it process information, the system is preprogrammed with an event loop, to look repeatedly for information to process (whether this might be the appearance of a file in a folder, a keyboard or mouse operation, or a timer event) and then perform a trigger function to process it. Programming an event-driven system is thus a matter of rewriting the default trigger functions of the system, to match the required behavior.

Inputs can be polled in the event loop, or interrupt handlers can be registered to react to hardware events (many systems use a combination of both techniques). The preprogrammed algorithm ensures that triggers provided are executed when they are needed, thus providing a software abstraction that emulates an interrupt-driven environment.

Event handlers

Event-driven programs consist of a number of small programs called event handlers (called in response to external events) and a dispatcher (which calls the event handlers).

In many cases, event handlers can trigger events themselves, possibly leading to an event cascade. Graphical user interface programs are typically programmed in an event-driven style.

Computer operating systems are another classic example of event-driven programs on at least two levels. At the lowest level, interrupt handlers act as direct event handlers for hardware events, with the CPU performing the role of dispatcher. Operating systems also act as dispatchers for software processes, passing data and software interrupts to user processes that, in many cases, are programmed as event handlers themselves.

A command line interface can be viewed as a special case of event-driven model in which the system, which is inactive, awaits one very complex event – the entry of a command by user.

WinMain FUNCTION

The WinMain function is the entry point of the application. When a user double clicks, Windows carries out some initialization code then passes control to this function. This is just like main() in C. In this function the application is set up and then enter a loop that will continue until the application is closed. The WinMain function is the conventional name for the user-provided entry point for a Microsoft Windows-based application.

Syntax

```
int WINAPI WinMain(

HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nCmdShow );
```

Parameters of WinMain Function:

- **hInstance**

  Handle to the current instance of the application.

- **hPrevInstance**

  Handle to the previous instance of the application. This parameter is always NULL. If you need to detect whether another instance already exists, create a uniquely named mutex using the CreateMutex function. CreateMutex will succeed even if the mutex already exists, but the function will return ERROR_ALREADY_EXISTS.

- **lpCmdLine**

  Pointer to a null-terminated string specifying the command line for the application, excluding the program name. To retrieve the entire command line, use the GetCommandLine function.

- **nCmdShow**

  Specifies how the window is to be shown. This parameter can be one of the following values:

SW_HIDE	Hides the window and activates another window
SW_MAXIMIZE	Maximizes the specified window
SW_MINIMIZE	Minimizes the specified window and activates the next top-level window in the Z order
SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when restoring a minimized window
SW_SHOW	Activates the window and displays it in its current size and position
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window
SW_SHOWMINIMIZED	Activates the window and displays it as a minimized window
SW_SHOWMINNOACTIVE	Displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED, except that the window is not activated
SW_SHOWNA	Displays the window in its current size and position. This value is similar to SW_SHOW, except the window is not activated
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL, except the window is not made active
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when displaying the window for the first time

Return Value

If the function succeeds, terminating is done when it receives a WM_QUIT message. Then it should return the exit value contained in that message's wParam parameter. If the function terminates before entering the message loop, it should return zero.

Function Information

Header	Declared in Winbase.h, include Windows.h
Import library	None
Minimum operating systems	Windows 95, Windows NT 3.1

CREATING WINDOWS

Every window is a member of a window class. A window class is a template for creating a window. When you write an application, you must register all of the window classes that are used to create windows. To simplify the process of creating windows, Windows CE includes several system-defined window classes; because Windows CE registers these classes automatically, you can create windows with them immediately.

Windows can be created with functions

- CreateWindow
- CreateWindowEx

The only difference between these functions is that CreateWindowEx supports the extended style parameter, dwExStyle, but CreateWindow does not. These functions take a number of parameters that specify the attributes of the window that is being created. In Windows CE, CreateWindow is implemented as a macro that calls CreateWindowEx.

In multithreaded applications, the main thread must create the main application before creating threads for any child windows.

Windows CE includes additional functions for creating special-purpose windows such as dialog boxes and message boxes, such as DialogBox, CreateDialog, and MessageBox.

The CreateWindowEx function uses the following syntax:

```
HWND

CreateWindowsEx(

DWORD   dwExStyle,           //Extended style

LPCWSTR   lpClassName,       //Class name

LPCWSTR   lpWindowName,  //Window name

DWORD   dwStyle              //Style

int   X,                     //Horizontal position

int   Y,                     //Vertical position

int   nWidth,                //Width

int   nHeight,               //Height

HWIND   hwndParent,          //Parent window

HMENU   hMenu,               //Menu

HINSTANCE hInstance,         //Instance handle

LPVOID   lpParam);           //Creation data
```

The CreateWindow function does not include the dwExStyle parameter. The following table shows the window attributes of CreateWindowEx.

Window attribute	**Description**
Extended style	The dwExStyle parameter specifies one or more window extended styles. These styles have their own set of WS_EX_* flags, which should not be confused with the WS_* flags.
Class name	Every window belongs to a window class. Except for built-in classes, such as controls, an application must register a window class before creating any windows of that class. The lpClassNam*e* parameter specifies the name of the class that is used as a template for creating the window.
Window name	The window name, also known as window text, is a text string that is associated with a window. The lpWindowName parameter specifies the window text for the newly created window. Windows use this text in different ways: A main window, dialog box, or message box typically displays its window text in its title bar; a button control, edit control, or static control displays its window text within the rectangle that is occupied by the control; a list box, combo box, or scroll bar control does not display its window name. All windows have the text attribute, even if they do not display the text.
Style	The dwStyle parameter specifies one or more window styles. A window style is a named constant that defines an aspect of the window's appearance and behavior. For example, a window with the WS_BORDER style has a border around it. Some window styles apply to all windows, others apply only to windows of specific window classes.
Horizontal and vertical coordinates	The *x* and *y* parameters specify the horizontal and vertical screen coordinates, respectively, of the upper-left corner of the window.
Width and height coordinates	The nWidth and nHeight parameters determine the width and height, respectively, of the window, in device units.

Parent	The hwndParent parameter specifies the parent window or the owner of a window, depending on the style of the flags that are passed in. If neither the WS_POPUP style nor the WS_CHILD style is specified, the hwndParent parameter might be either a valid window handle or NULL. If the parameter is NULL, the new window is a top-level window without a parent window or owner window. If the parameter is non-NULL, the new window is created as a child window of the specified parent window. If the WS_CHILD style is specified, the hwndParent parameter must be a valid window handle. The new window is created as a child window of the parent window. If the WS_POPUP style is specified, the new window is created as a top-level window, and the hwndParent parameter specifies the owner window. If WS_POPUP is specified and the parameter is NULL, the new window is owned partially by Windows CE. The WS_POPUP style overrides the WS_CHILD style.
Menu	Windows CE does not support menu bars. In Windows CE, you can use the hMenu parameter to identify a child window. Otherwise, it must be NULL.
Instance handle	The hInstance parameter identifies the handle of the specific instance of the application that creates the window.
Creation data	Every window receives a WM_CREATE message when it is created. The lpParam parameter is passed on, as one of the message parameters. Although it can be any value, it is most commonly a pointer to a structure containing data that is required to create a particular window.

The class name for a new window class has to be a Unicode string. We can use the TEXT macro to cast a string as Unicode, as in TEXT("classname"). We can also use the _T macro, as in __T("classname"). Another option is to declare the string as a Long string (L"classname"), which will make the string Unicode only.

Displaying the window:

The system do not display the main window automatically after the system creates the window. Instead, the WinMain function of the application uses the ShowWindow function to display the window. CreateWindow will return an HWND data object, which is a reference to newly created window. The program will pass this handle to the ShowWindow function to make the window appear on the screen.

An application uses the SetWindowText function to change the window text after it creates the window; it uses the GetWindowTextLength and GetWindowText functions to retrieve the window text from a window.

Syntax:

ShowWindow(hwnd , iCmdShow);

hwnd: Handle to the window just created by Create Window

iCmdShow: Value passed as parameter to WinMain. This determines how the window is to be initially displayed on thescreen.ie whether it is normal, minimized or maximized.

MESSAGE LOOP

Messages come in the form of an MSG data type. This data object is passed to the GetMessage() function, which reads a message from the message queue, or waits for a new message from the system. Next, the message is sent to the TranslateMessage() function, which takes care of some simple tasks, such as translating to Unicode or not. Finally, the message is sent to the window for processing with the DispatchMessage() function.

A simple message loop consists of one function call to each of these three functions: GetMessage, TranslateMessage, and DispatchMessage.

If there is an error, GetMessage returns -1, thus the need for the special testing.

```
MSG msg;

BOOL bRet;

while((bRet=GetMessage(&msg,NULL,0,0))!=0)

{

if(bRet==-1)

{

//handle the error and possibly exit

}

else

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}
```

The GetMessage function retrieves a message from the queue and copies it to a structure of type MSG. It returns a nonzero value, unless it encounters the WM_QUIT message, in which case it returns FALSE and ends the loop. In a single-threaded application, ending the message loop is often the first step in closing the application. An application can end its own loop by using the PostQuitMessage function, typically in response to the WM_DESTROY message in the window procedure of the application's main window.

If you specify a window handle as the second parameter of GetMessage, only messages for the specified window are retrieved from the queue. GetMessage can also filter messages in the queue, retrieving only those messages that fall within a specified range. For more information about filtering messages, see Message Filtering.

A thread's message loop must include TranslateMessage if the thread is to receive character input from the keyboard. The system generates virtual-key messages (WM_KEYDOWN and WM_KEYUP) each time the user presses a key. A virtual-key message contains a virtual-key code that identifies which key was pressed, but not its character value. To retrieve this value, the message loop must contain TranslateMessage, which translates the virtual-key message into a character message (WM_CHAR) and places it back into the application message queue. The character message can then be removed upon a subsequent iteration of the message loop and dispatched to a window procedure.

The DispatchMessage function sends a message to the window procedure associated with the window handle specified in the MSG structure. If the window handle is HWND_TOPMOST, DispatchMessage sends the message to the window procedures of all top-level windows in the system. If the window handle is NULL, DispatchMessage does nothing with the message.

An application's main thread starts its message loop after initializing the application and creating at least one window. Once started, the message loop continues to retrieve messages from the thread's message queue and to dispatch them to the appropriate windows. The message loop ends when the GetMessage function removes the WM_QUIT message from the message queue.

Only one message loop is needed for a message queue, even if an application contains many windows. DispatchMessage always dispatches the message to the proper window; this is because each message in the queue is an MSG structure that contains the handle of the window to which the message belongs.

You can modify a message loop in a variety of ways. For example, you can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages not specifying a window. You can also direct GetMessage to search for specific messages, leaving other messages in the queue. This is useful if you must temporarily bypass the usual FIFO order of the message queue.

An application that uses accelerator keys must be able to translate keyboard messages into command messages. To do this, the application's message loop must include a call to the TranslateAccelerator function. For more information about accelerator keys, see Keyboard Accelerators.

If a thread uses a modeless dialog box, the message loop must include the IsDialogMessage function so that the dialog box can receive keyboard input.

WINDOW PROCEDURES

Every window has an associated window procedure a function that processes all messages sent or posted to all windows of the class. All aspects of a window's appearance and behavior depend on the window procedure's response to these messages. Each window is a member of a particular window class. The window class determines the default window procedure that an individual window uses to process its messages. All windows belonging to the same class use the same default window procedure. For example, the system defines a window procedure for the combo box class (COMBOBOX); all combo boxes then use that window procedure.

An application typically registers at least one new window class and its associated window procedure. After registering a class, the application can create many windows of that class, all of which use the same window procedure. Because this means several sources could simultaneously call the same piece of code, you must be careful when modifying shared resources from a window procedure. For more information, see Window Classes.

Window procedures for dialog boxes (called dialog box procedures) have a similar structure and function as regular window procedures. All points referring to window procedures in this section also apply to dialog box procedures. This section discusses the following topics.

- Structure of a Window Procedure
- Default Window Procedure
- Window Procedure Subclassing
- Window Procedure Superclassing

Structure of a Window Procedure

A window procedure is a function that has four parameters and returns a signed value. The parameters consist of a window handle, a UINT message identifier, and two message parameters declared with the WPARAM and LPARAM data types.

Message parameters often contain information in both their low-order and high-order words. There are several macros an application can use to extract information from the message parameters. The LOWORD macro, for example, extracts the low-order word (bits 0 through 15) from a message parameter. Other macros include HIWORD, LOBYTE, and HIBYTE.

The interpretation of the return value depends on the particular message. Because it is possible to call a window procedure recursively, it is important to minimize the number of local variables that it uses. When processing individual messages, an application should call functions outside the window procedure to avoid excessive use of local variables, possibly causing the stack to overflow during deep recursion.

Default Window Procedure:

The default window procedure function, DefWindowProc defines certain fundamental behaviour shared by all windows. The default window procedure provides the minimal functionality for a window. An application-defined window procedure should pass any messages that it does not process to the DefWindowProc function for default processing.

Window Procedure Subclassing:

When an application creates a window, the system allocates a block of memory for storing information specific to the window, including the address of the window procedure that processes messages for the window. When the system needs to pass a message to the window, it searches the window-specific information for the address of the window procedure and passes the message to that procedure.

Subclassing is a technique that allows an application to intercept and process messages sent or posted to a particular window before the window has a chance to process them. By subclassing a window, an application can augment, modify, or monitor the behavior of the window. An application can subclass a window belonging to a system global class, such as an edit control or a list box. For example, an application could subclass an edit control to prevent the control from accepting certain characters. However, you cannot subclass a window or class that belongs to another application. All subclassing must be performed within the same process.

An application subclasses a window by replacing the address of the window's original window procedure with the address of a new window procedure, called the subclass procedure. Thereafter, the subclass procedure receives any messages sent or posted to the window.

The subclass procedure can take three actions upon receiving a message:

- it can pass the message to the original window procedure
- modify the message and pass it to the original window procedure
- Process the message and not pass it to the original window procedure.

If the subclass procedure processes a message, it can do so before, after, or both before and after it passes the message to the original window procedure.

The system provides two types of subclassing: instance and global. In instance subclassing, an application replaces the window procedure address of a single instance of a window. An application must use instance subclassing to subclass an existing window. In global subclassing, an application replaces the address of the window procedure in the WNDCLASS structure of a window class. All subsequent windows created with the class have the address of the subclass procedure, but existing windows of the class are not affected.

Window Procedure Superclassing:

Superclassing is a technique that allows an application to create a new window class with the basic functionality of the existing class, plus enhancements provided by the application. A superclass is based on an existing window class called the base class. Frequently, the base class is a system global window class such as an edit control, but it can be any window class.

A superclass has its own window procedure, called the superclass procedure. The superclass procedure can take three actions upon receiving a message: It can pass the message to the original window procedure, modify the message and pass it to the original window procedure, or process the message and not pass it to the original window procedure. If the superclass procedure processes a message, it can do so before, after, or both before and after it passes the message to the original window procedure.

Unlike a subclass procedure, a superclass procedure can process window creation messages (WM_NCCREATE, WM_CREATE, and so on), but it must also pass them to the original base-class window procedure so that the base-class window procedure can perform its initialization procedure.

To superclass a window class, an application first calls the GetClassInfo function to retrieve information about the base class. GetClassInfo fills a WNDCLASS structure with the values from the WNDCLASS structure of the base class. Next, the application copies its own instance handle into the hInstance member of the WNDCLASS structure and copies the name of the superclass into the lpszClassName member. If the base class has a menu, the application must provide a new menu with the same menu identifiers and copy the menu name into the lpszMenuName member. If the superclass procedure processes the WM_COMMAND message and does not pass it to the window procedure of the base class, the menu need not have corresponding identifiers. GetClassInfo does not return the lpszMenuName, lpszClassName, or hInstance member of the WNDCLASS structure.

MENUS

A menu is the most important part of the consistent user interface that windows program offer. A menu is a list of application commands. Each command is replaced by a string or bitmap called a menu item. Whenever the user chooses a menu item, windows sends the application a command message that indicates which menu item the user choose. Menu items I pop ups can be enabled, disabled or grayed.

Menu has a hierarchical structure .A menu bar resides at the uppermost level of hierarchy, immediately below the caption bar. The top level menu has a menu handle. Each popup menu within a top level menu has its own handle. Selecting an item from the menu bar activates the popup menu.

Example:
Selecting a file item in the menu bar activates a popup menu with the items such as the New, Open, Save and so on.

Each item in a menu is defined by three characteristics which include:

- What appears in the menu (either a text string or a bitmap).
- An ID number that WINDOWS sends to the program.
- The handle to the top up menu.

Menus and messages:

Windows sends several different messages to the window procedure when the user selects a menu item.

Examples:

- WM_MENUSELECT – A menu tracking message.
  A program can receive many WM_MENUSELECT messages as the user moves the cursor or mouse among the menu items

- WM_COMMAND – Indicates that user has chosen an enabled menu item from the window's menu.

- WM_SYSCOMMAND - Indicates that user has chosen an enabled menu item from the system menu.

- WM_MENUCHAR - this message is issued in 2 circumstances. That is, when user press the
  - ALT and a character that does not correspond to a menu item.
  - Character key that does not correspond to an item in pop up.

Menu functions:

1) Append Menu: Adds a new menu item to the end of a menu
2) CheckMenuItem: Checks or unchecks the menu item
3) CreatePopupMenu: Creates a popup menu
4) CreateMenu: Creates a new empty menu
5) DeleteMenu: Removes an item from a menu
6) DestroyMenu: Removes a menu from the memory
7) DrawMenuBar: Forces a window's menu bar to be repainted

8) EnableMenuItem: Changes a menu item to/from enabled.
9) GetMenu: Retrieves a handle to the window's menu.
10) GetMenuItemCount: Gets the number of menu items in a menu.
11) GetMenuItemID: Retrieves the ID value associated with the menu item.
12) GetMenuState: Finds the number of items in a menu or the status of an item.
13) GetMenuString: Retrieves the label displayed in a menu item.
14) GetSubMenu: Retrieves a handle to a popup menu.
15) GetSystemMenu: Retrieves a handle to the system menu.
16) InsertMenu: Inserts a new menu item to an existing menu.
17) ModifyMenu: Changes the properties of a menu item.

BUTTONS

Push buttons:
A push button is a rectangle enclosing text specified in the window text parameter of the CreateWindow call. The push button controls are used mostly to trigger an immediate action without retaining any type of on/off indication The two types of push button controls have window styles called BS_PUSHBUTTON and BS_DEFPUSHBUTTON .The DEF stands for default .Pressing the push button causes the button to be repainted .Releasing the mouse button restores the original appearance and sends a WM_COMMAND message to the parent window with the notification code BN_CLICKED.

Checkbox:
A check box is a square box with text to the right .The most common styles for a checkbox are BS_CHECKBOX and BS_AUTOCHECKBOX. The state of a check box can be set by a message BS_SETCHECK message. The state of the checkbox can be retrieved by using the BS_GETCHECK message.

Radio Button:
The radio button looks very much like a check box except that it contains a little circle rather than a box .A dot within the circle indicates that the button is checked. The radio button has the 3 window styles BS_RADIOBUTTON or BS_AUTORADIOBUTTON

Group box:
Group boxes are used to enclose other button controls. The group box is a rectangular outline with its window text at the top. It neither processes mouse or keyboard input nor sends WM_COMMAND message to its parents.

Changing the button text:
We can change the text in a button by calling SetWindowText.

SetWindowText (hwnd, pszString); where hwnd is a handle to the window whose text is being changed and the pszString is a pointer to a null terminated string .For a normal window this text is the text of the caption bar. For a button control, it is the text displayed with the button.

The Button Colors:
Each of the different Buttons requires multiple colors.
COLOR_BTNFACE is used for the main surface color of the push buttons and the back ground color of others.
COLOR_BTNSHADOW - for suggesting a shadow at the right and bottom sides of the push buttons and the insides of the checkbox squares and radio button circles.
COLOR_BTNTEXT - For text color

DRAWING ON WINDOWS

The subsystem of Microsoft Windows responsible for displaying graphics on video displays and printers is known as the Graphics Device Interface (GDI). GDI is an extremely important part of Windows. Not only do the applications we write for Windows use GDI for the display of visual information, but windows itself uses GDI for the visual display of user interfaces. GDI consist of several hundred function calls and some associated data types, macros and structures. Windows uses the device context to store "attributes" that govern how the GDI functions operate on the display

1) SetPixel(hdc,x,y,crColor);
2) crColor=GetPixel(hdc,x,y);
3) LineTo(hdc,x,y);-Draws straight line
4) Polyline and PolylineTo –draws a series of connected straight lines
5) PolyPolyline –Draws multiple polylines.
6) Arc-Draws elliptical lines
7) PolyBezier and PolBezierTo –Draw Bezier splines
8) ArcTo and AngleArc –Draw elliptical lines
9) PolyDraw –Draws a series of connected straight lines and Bezier splines
10) Rectangle – Draws a rectangle
11) Ellipse – Draws an ellipse
12) RoundRect –Draws a rectangle with rounded corners.
13) Pie- Draws a part of an ellipse that looks like a pie slice
14) Chord –Draws part of an ellipse formed by a chord

To draw a straight line, you must call two functions. The first function specifies the point at which the line begins, and the second function specifies the end point of the line:

```
MoveToEx (hdc, xBeg, yBeg, NULL)

LineTo (hdc, xEnd, yEnd);
```

MoveToEx doesn't actually draw anything instead, it sets the attribute of the device context known as the "currentposition”. LineTo function then draws a straight line from the current position to the point specified in the LineTo function.

Rectangle, Ellipse, RoundRect, Chord, and Pie functions are not strictly line-drawing algorithms, but they draw lines and also fill the enclosed area with the current area-filling brush

1. Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
2. Ellipse (hdc, xLeft, yTop, xRight, yBottom) ;
3. RoundRect (hdc, xLeft, yTop, xRight, yBottom, xCornerEllipse, yCornerEllipse);
4. Arc (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
5. Chord (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
6. Pie (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);