VISUAL & WINDOWS

PROGRAMMING

Win32 Programming

Purbanchal University: BCA IV Semester

S@R0Z

Purbanchal University: VISUAL AND WINDOWS PROGRAMMING

Syllabus

Objective

The main of the course is to promote a good foundation in GUI basics, rational and standards, windows programming techniques etc. through assignments and projects.

Contents:

1. **Introduction to Window concepts:** What is windows, Historical perspective of User Interface, difference Between Window Program and DOS or UNIX program, Windows Programming model, Memory Model, Static and Dynamic Linking. Windows Memory Management

2. **A skeletal Windows application:** The Skeleton Application Source Program, The Winmain Function, Registering the Windows Class, WinMain's Message Loop, Windows Function, About Dialog Function, Components of the Skeleton application.

3. **Displaying text in window:** Device & Display Context (WM-PAINT, Text Out, Logical Coordinates & Device Coordinates), Introduction to Scroll Bar (Parts of a Scroll Bar), Sub-Classing a Window Class (Sub-Classing, Sub-Classing Technique), What is Device Context (DC), Specific Types of Display Context, Display Context (Common Display Context, Less Display Context, Private Display Context & Window Display Context), Attributes of a Display Context (Colors, Defining Color, Bitmap, Brush, Pen & Regional Attributes & Objects)

4. **Graphical Output: (Pixels, Lines & Polygons):** Getting & Setting the Class of a Pixel, Drawing Lines & Pages, Drawing Modes, Drawing Filled Areas- Rectangle & Ellipses, Drawing & Filling Polygons)

5. **Keyboard, Mouse and Timer Input:** **Keyboard**: (Input, Interfaces, Press & Release Massage), Vertical Key Code, Character Massage, Character Sets) **Mouse**: (Mouse Input, Mouse Massages Hit Testing, Capturing the Mouse, Simple Drop & Drag) **Timer**: (Timer Input, Sending WM_Timer Massage, Using Timer)

6. **Using controls:** Overview, Static, Button AND Edit Classes, Creating & Centralize Window, Control Notified Button Class, Check Boxes, Radio Button Edit Class (Style, Massage to Edit Control, Working with Selection),List box, combo box, { ***image list and tree view classes*** }, dialogue boxes.

7. **Dialog Box:** What is Dialog Box, Dialog Function & Cell Book, Function, Creating Model & Modeless Dialog Box)

   Menu and Icons: Menu (Defining & Creating Dropdown & Popup Menu), Icons (Defining, Loading, Displaying)

8. **Printing:** Overview of Printing Process using Default & Installed Printers, Getting the Printers Determining Device Mode Value, Exempla Printing Document)

9. **Memory management:** Dynamic Memory Allocation, (Fixed Movable & Discarded Memory, Block Managing Memory, Block Using the Global Function, Allocating Fix Memory Book Allocating Movable Memory Block, Allocating Discardable Memory Block), Locking & Unlocking Memory Blocks, Reallocating Framing Memory Block.

Lab Exercises:

There shall be above thirty lab exercises covering all the topics.

Textbook:

1. Brent E Rector and Joseph M Newcomer, Win32 programming, Addison Wesley, 1999.

References

1. Charles Petzold, Programming Windows 95. Microsoft Press. 1996.
2. Richard J. Simon. Windows NT, Win32 API Super Bible SAMS. 1997.

Course Plan

[For instructor's purpose only]

1. **Program of Study** — Bachelor of Computer Application [B.C.A]
   Purbanchal University, Kantipur City College

2. **Course Code** — BCA 255 CS
3. **Course Title** — Visual & Windows Programming

4. **Number of Credits** — 3 (3-1-4) (Lecture-Lab-Self Study)

5. **Course Description**

Teaching schedule Hours/Week			Examination Scheme				
Theory	**Tutorial**	**Practical**	**Internal Assessment**		**Final**		**Total**
3	1	3	Theory	Practical*	Theory **	Practical	100
			20	20	60	-	

6. **Course Objectives**

   The main of the course is to promote a good foundation in GUI basics, rational and standards, windows programming techniques etc. through assignments and projects.

7. **Plan**

Week	Topics	Hours		
		Lecture	**Lab**	**Self Study**
1.	**Introduction to Window concepts:** What is windows? Historical perspective of User Interface, difference Between Window Program and DOS or UNIX program, Windows Programming model, Memory Model, Static and Dynamic Linking. Introduction to Windows Memory Management **[Assignment 01]**	**3**	**1**	**4**
2.	**A skeletal Windows application:** The Skeleton Application Source Program, The Winmain Function, Registering the Windows Class,	**3**	**1**	**4**

	Winmain's Message Loop, Windows Function, About Dialog Function, Components of the Skeleton application.			
	UNIT TEST			
3.	**Displaying text in window:** Device & Display Context (WM-PAINT, Text Out, Logical Coordinates & Device Coordinates), **[Assignment 02]**	**3**	**1**	**4**
4.	Introduction to Scroll Bar (Parts of a Scroll Bar), Sub-Classing a Window Class (Sub-Classing, Sub-Classing Technique)	**3**	**1**	**4**
5.	**Examining a Device Context in Depth:** What is Device Context (DC), Specific Types of Display Context, Display Context (Common Display Context, Less Display Context, Private Display Context & Window Display Context) **[Assignment 03]**	**3**	**1**	**4**
	MID TERM EXAMINATION			
6.	**Graphical Output: (Pixels, Lines & Polygons):** Getting & Setting the Class of a Pixel, Drawing Lines & Pages, Drawing Modes, Drawing Filled Areas- Rectangle & Ellipses, Drawing & Filling Polygons)	**3**	**1**	**4**
7.	**Keyboard, Mouse and Timer Input: Keyboard**: (Input, Interfaces, Press & Release Massage), Vertical Key Code, Character Massage, Character Sets) **[Assignment 04]**	**3**	**1**	**4**
8.	**Mouse**: (Mouse Input, Mouse Massages Hit Testing, Capturing the Mouse, Simple Drop & Drag) **Timer**: (Timer Input, Sending WM_Timer Massage, Using Timer)	**3**	**1**	**4**
9.	**Using controls:** Overview, Static, Button AND Edit Classes, Creating & Centralize Window, Control Notified Button Class, Check Boxes, Radio Button Edit Class (Style, Massage to Edit Control, Working	**3**	**1**	**4**

	with Selection)			
10.	**Using controls:** list box, combo box, { ***image list and tree view classes***}	**3**	**1**	**4**
11.	**Dialog Box:** What is Dialog Box, Dialog Function & Cell Book, Function, Creating Model & Modeless Dialog Box) **[Assignment 05]**	**3**	**1**	**4**
12.	**Menu and Icons:** Menu (Defining & Creating Dropdown & Popup Menu), Icons (Defining, Loading, Displaying)	**3**	**1**	**4**
13.	**Printing:** Overview of Printing Process using Default & Installed Printers, Getting the Printers Determining Device Mode Value, Exempla Printing Document) **[Assignment 06]**	**3**	**1**	**4**
14.	**Memory management:** Dynamic Memory Allocation, (Fixed Movable & Discarded Memory, Block Managing Memory, Block Using the Global Function, Allocating Fix Memory Book Allocating Movable Memory Block, Allocating Discardable Memory Block), Locking & Unlocking Memory Blocks, Reallocating Framing Memory Block. **[Assignment 07]**	**3**	**1**	**4**
15.	**Course Revision**	**3**	**1**	**4**
	Total	**45**	**15**	**60**
END TERM EXAMINATION				

Note: Assignments, Labsheets & Questions are not included.

16. Teaching Method (s)

a. Lectures
b. Discussions
c. Labs

17. Teaching Media & Materials

- **a.** LCD and Overhead projector
- **b.** Handouts
- **c.** Text Books
- **d.** Internet Resources

18. Students Evaluation Criteria

		Theory : 20 Marks
S NO	Criteria Name	Weightage %
1.	Class Assignments	35
2.	Class Attendance	10
3.	Class Test	5
4.	Mid Term	20
5.	End Term	30
		Practical : 20 Marks
1.	Lab Attendance	**20**
2.	Lab Work	**30**
3.	Lab Sheet Submission	**50**

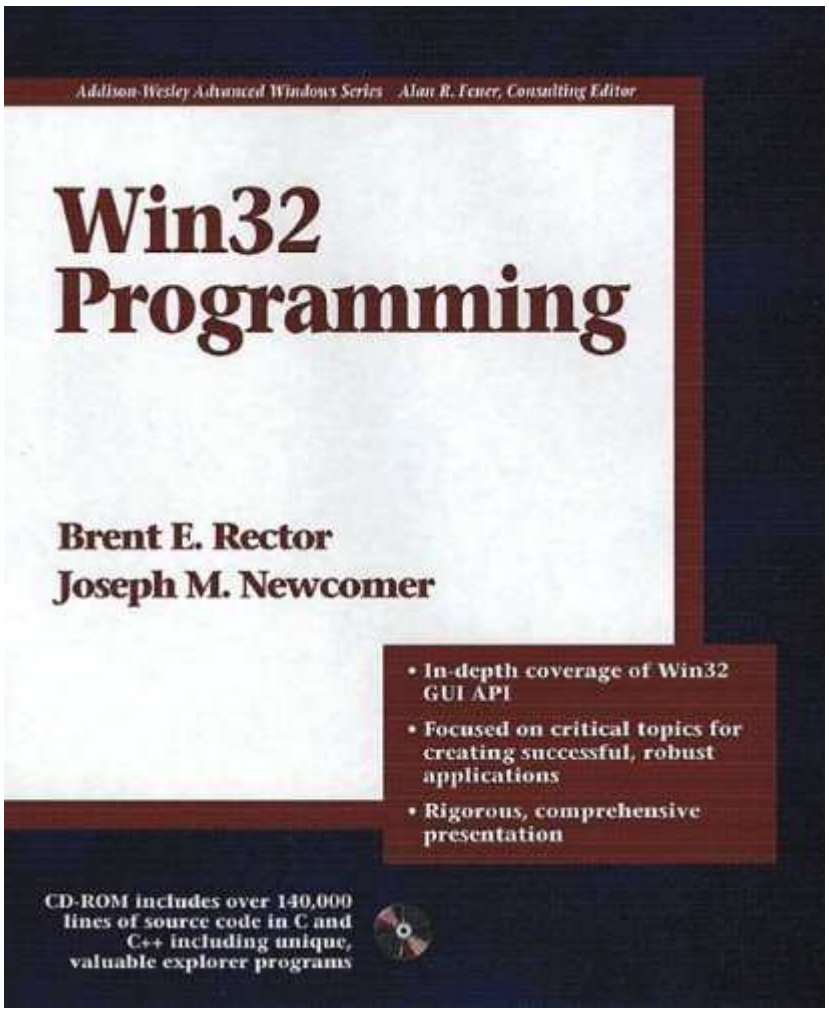
19. Instructor

Mr. Saroj Pandey

Email: sarojpandey@kcc.edu.np

20. References

a. Brent E Rector and Joseph M Newcomer, Win32 programming, Addison Wesley, 1999.
b. Charles Petzold, Programming Windows 95. Microsoft Press. 1996.
c. Richard J. Simon. Windows NT, Win32 API Super Bible SAMS. 1997.
d. Microsoft Documentation on visual programming
e. MSDN Reference Library.


Addison-Wesley Advanced Windows Series Alan R. Feuer, Consulting Editor
Win32 Programming
Brent E. Rector
Joseph M. Newcomer
• In-depth coverage of Win32 GUI API
• Focused on critical topics for creating successful, robust applications
• Rigorous, comprehensive presentation
CD-ROM includes over 140,000 lines of source code in C and C++ including unique, valuable explorer programs

In pursuit of the sublime…

Table of Contents

Win32 Programming is fundamentally simple but you have to be a genius to understand its simplicity.

Chapter 01: Introduction to Windows

What is Windows?

Windows is a graphical environment that runs multiple applications simultaneously. Each application displays output in a rectangular area of the computer screen called window. The entire computer screen is refereed as Desktop. A User can arrange Windows on the desktop in a manner similar to placing the paper on desk.

For the Programmer, windows provide a rich programming environment that supplies extensive support for developing easy to use user interfaces. Menu, Dialog Boxes, List Boxes, Scroll Bars, Push Buttons and Windows supplies other components of a user interface to the developer. Windows provides the device independent graphics that allows writing the programs without having detailed knowledge of the hardware platform. The programmer can also access the keyboard, mouse, printer, system timer and serial communication ports in the device independent manner.

What is Win32?

Win32 is a family of Windows Programming Interfaces. It is standard programming interface for Windows 95 and Windows NT. The win32 Application programming Interface uses 32bit address space rather than 16bit values used in Windows 3.x and earlier version of windows. Win32 includes many new functions than Win16. The Kernel allows to create powerful programs by using the capability like multithreading.

User Interface

- Interface for the direct interaction with the user
- Display Terminals / CGA, EGA, VGA different video graphics display
- Multitasking (Cooperative multitasking vs. preemptive multitasking)
- Windows have rich set of graphical input controls as the part of GUI which helps on making the program more interactive and user friendly.

E.g. Buttons, Menus, Dialog Box, combo box etc.

Difference between Windows Program and DOS or UNIX program

There are numbers of differences between Windows Program and DOS Program. Those are as follows:

1. **Resources Sharing**

Windows applications are designed to share the resources of the system on which they run. Resources are Memory, Processor, Display, Keyboard, Hard disk etc. To achieve the resources sharing a windows application must interact with the resources of computer only through Windows Application Programming Interfaces (APIs).

DOS Application typically expects all the resources of the system to be available to the application. A DOS application can allocate all available memory without concern to other application. It can directly access the serial and parallel ports. It can even interpret and handle key down/up events directly. So the DOS applications are not designed to share resources.

2. **Graphical User Interface**

Windows application runs with a graphical user interface (GUI). Multiple applications may run concurrently and each application is given access to the graphical display through a window. Windows applications are not restricted to one window per application. A window application may have more than one window. Windows also contains a rich set of input controls (textbox*, radio buttons, buttons, menus etc.*) as a part of GUI.

DOS application does not have such graphical interface it only have command line interface.

3. **Input Facilities**

Windows controls the input devices and distributes input from the devices among each of the multiple concurrent applications as required. A window application is structured to attempt input whenever the user produces it rather than demand it when it is required by the application. An application is notified in each keyboard or mouse events.

This aspect of windows application is quite different from standard DOS/UNIX applications. Those applications force user for input and till then the input devices cannot be used by others.

4. **Memory Management**

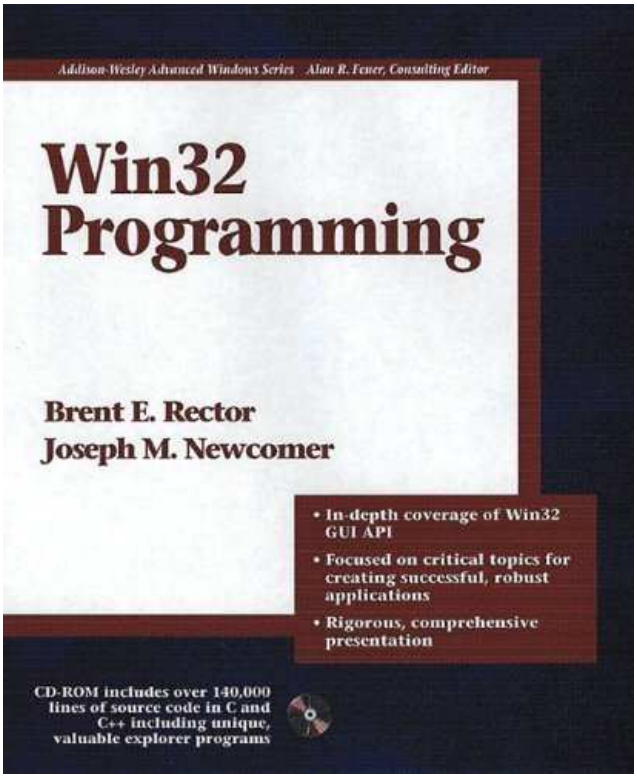
Memory is important resources that all applications share in windows environment. A DOS program often uses entire available computer's memory, whereas multiple windows application must share the memory. Instead of initially reserving all memory required by an application and holding it until the application terminates, a windows application should allocate only the storage required at any given time and free it as soon as it is no longer needed.

5. **Device Independent Graphics**

Windows provides device independent graphical operations. The same graphical operation that produce a pie chart on the displays also draw one on the printer, plotter or any other output devices. But this is not possible in DOS.

Historical perspective of User Interface

Refer: *Win32 Programming,* by Brent E. Rector and John M. Newcomer, Chapter 1.



Windows Programming model [Conceptual View]

Windows applications and the programs that react to different forms of user input and provide the graphical output. The windows application can be called as the collection of objects. The object oriented concepts apply naturally to Windows applications. There are several objects in windows, like Pen, Brush, Menu, Dialog Box etc. The first and most fundamental object is called '***window***'.

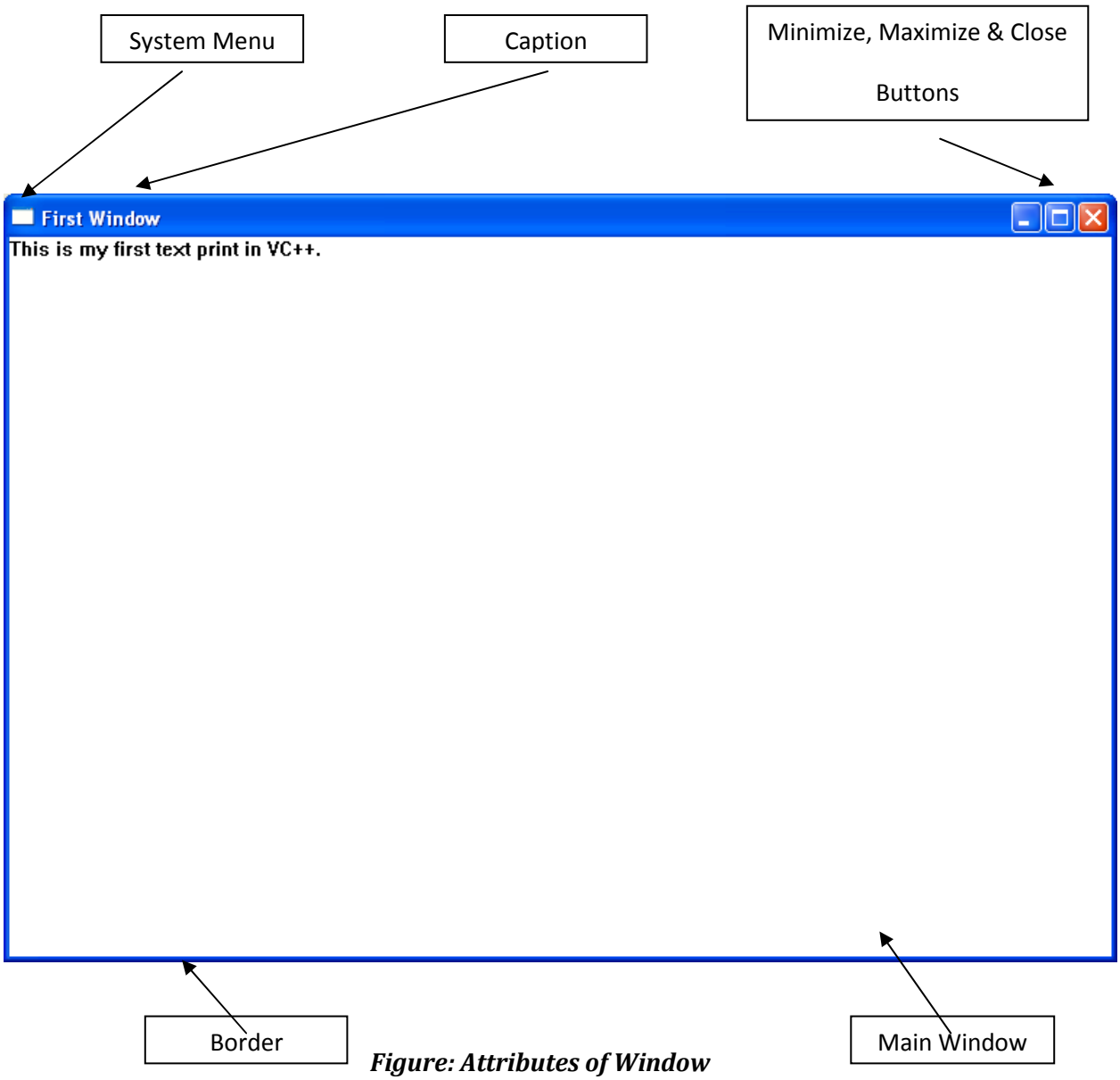


Figure: Attributes of Window

A window has certain data associated with it. It has a Background Color, Caption, Menu etc. A window also contains another window, which is called child window. The child window also has all properties except Menu. If a mouse is clicked in child window, the child window is notified. It may then apply to its parent window. It is often convenient in windows programming to create

a window. We can use just existing window as template for the new ones rather than creating new each time.

Object oriented programming viewpoint says, send the window a notification of the event and let it decide. A window can also contain another window. OOP also encourages the polymorphism ability of an object to take on many different behaviors. For example one window may react in one manner to depressed mouse button, where as another window might react quite differently to the same action.

OOP also provides inheritance. It is often convenient in windows programming to create a window. We can just use existing windows templates for designing new ones, rather than create the window each time. This is important if you are modifying the behaviors of built-in controls, since otherwise we have to program all their functionality in the application. Instead our program only changes the functionality that is you deem important

Windows and associated windows functions

When a program creates a window that window has certain characteristics – its location on screen, a title, a size, a menu and so on. These are physical characteristics. It also has a behavioral characteristic which allows windows to react to the notification in various events. All windows have associated functions called windows function that determines how the window reacts to the notification of various events. That notification is called ***message.*** E.g. Buttons and scroll bar controls, themselves contained within a window, are also windows. So they too have associated window function that controls how the button or scroll bar reacts.

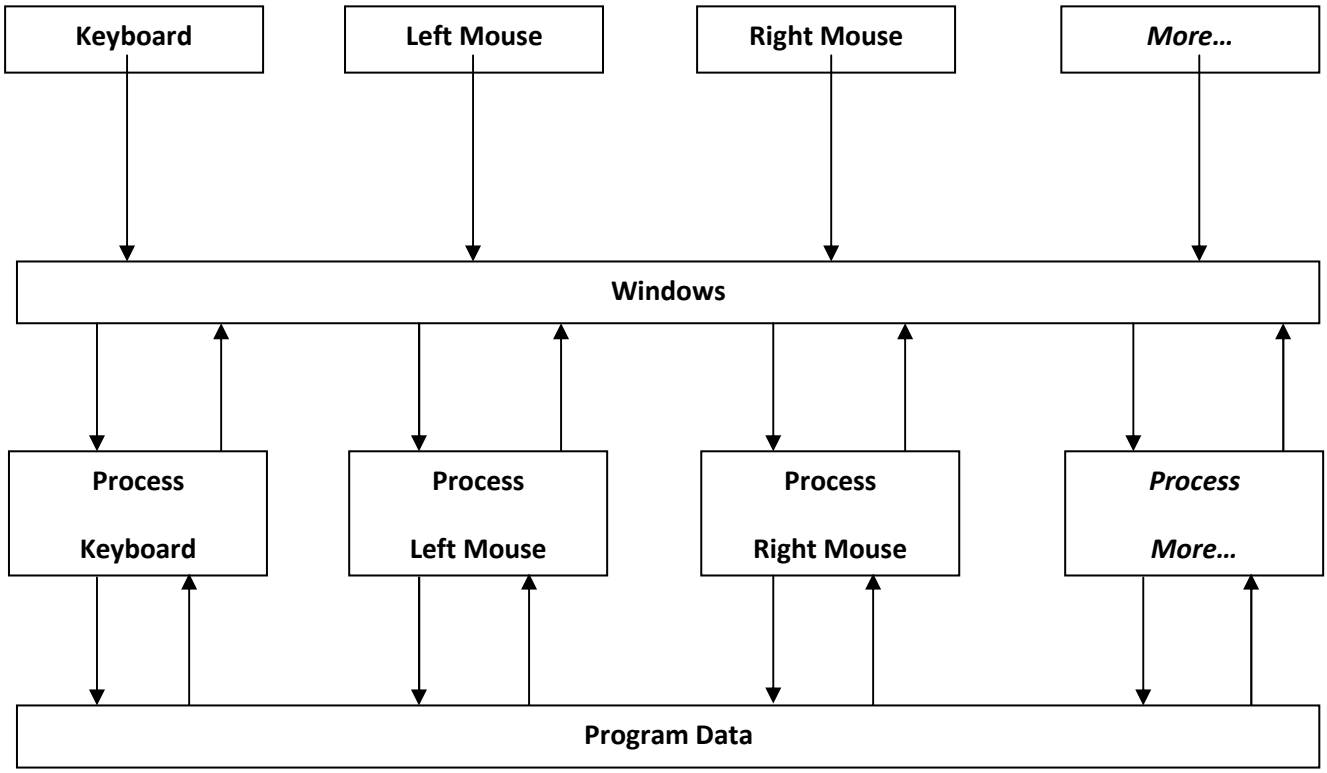
Windows receives a message notifying it of the event. In actually, windows calls the function associated with the window, passing information in the argument of the call that describes the events that have occurred. Windows, their associated window functions and the messages they receive are inter-related.

Where do message come from?

Messages originate from various sources. Although most of the messages originate from the windows operating system, an application may send itself a message from one window to another. A window application uses messages to toll itself to do something in the future. Later when the message arrives the application performs the desired action. A message may ask a

window what its caption is or its status or if it is willing to be destroyed. There are also messages that tell the window to do something, e.g. change the caption, change the background color, or any other appearance.

Fig: WINDOWS EVENTS



Static and Dynamic Linking

If the Linker does find the function its objects code is copied from the library and inserted into the executable file, then this is usual form of linking and is called static Linking. In static linking a fixed links between the program and prewritten modules is provided. In order to use those prewritten modules we need to copy them into final file at compile time in a process called static linking. Static Linking requires the linker to know at link time where the function will resides in memory and to have access to the object code comprising the function.

In dynamic Linking the external library file never gets bound into the final executable file. It remains outside the program as a DLL, in a place where the executable file can find it and send it messages. At run time these messages are function calls requesting certain parts of the DLL code are executed. To Link the executable and DLL we just tell the program where the DLL is and

which bit of code to run from the program. The linker doesn't know at the link time where the function resides.

Some Import Libraries used for linking Windows Program

- ***KERNAL32.dll***
    - *The main DLL for handling memory management, multitasking of the program that are running and the most other functions which directly affects how windows runs.*
- ***USER32.dll***
    - *Contains functions which deal with menu, timers, communications, files and other many areas of windows.*
- ***GDI32.dll***
    - *Graphical Device Interface provides the functions necessary to draw things on the screen, as well as checking which areas of forms need to be redrawn.*
- ***WINMM.dll***
    - *Provides multimedia functions for dealing with sound, music, real time video etc. This is 32 bit DLL equivalent to 16 bit MMSYSTEM.*

Etc. – Refer TEXTBOOK for more...

Windows Memory Model & Memory Management

The purpose of memory management in visual environment is same as that we use in console based C & C++ programming.

Windows program consists of one or more code and data segments. Dynamic Link Libraries may have one or more code and data segments. Multiple windows program can run concurrently and multiple instances of the same program may also run concurrently. To make all those thing possible windows uses many memory management functions and we also need to implement those functions whenever we require in our application.

(Detailed in Chapter 09)

Header files & Windows Data Types

All the windows program must use a header file called <windows.h> which contains three types of statements ***#define, typedef and Functions Prototypes***. <windows.h> and its components are ultimately quite large for an include file and many windows applications do not uses much of the information present <windows.h>.

The Data types supported by Microsoft Windows are used to define function return values, function and message parameters and structure members. Some of those which we use must is shown in following table. Remember the data types are all caps.

Type	Definition
BOOL	Boolean Variables (TRUE or FALSE)
BYTE	Byte – 8 bits
CALLBACK	Calling Convention of functions
CHAR	Character
COLORREF	[Red, Green, Blue] color Value 32 bits
CONST	Constant [Value remains constant during execution]
DWORD	32 bit unsigned integer
FLOAT	Floating Point Variable
HPEN	Handle to the Pen
HBRUSH	Handle to the Brush
HCURSOR	Handle to the cursor
HDC	Handle to the Device context
HDESK	Handle to the desktop
HFILE	Handle to the file
HFONT	Handle to the font
HICON	Handle to the Icon

HWND	Handle to the window
HINSTANCE	Handle to the Instance
HMENU	Handle to the Menu
INT	32 bit signed Integer
LPCSTR	Pointer to a constant Null terminated string
LPBOOL	Pointer to a BOOL
LPBYTE	Pointer to a BYTE
Etc. Refer **TEXTBOOK** for more...	

{! imp Note: Codes of Visual & Windows Programming are fully case sensitive.}

Standard Prefix to windows variables (Only few is here)

Prefix (All Small)	Data Type (All Caps)
b	BYTE
ch	CHAR
d	DOUBLE
dw	DWORD
f	BOOK
H	HANDLE
hwnd	HWND
l	LONG
n	SHORT
rgb	RGB
w	WORD

Functions and variables naming conventions

It's usually best to choose a consistent set of naming conventions for use throughout your code. Naming conventions usually govern things such as how you capitalize your variables, classes, and functions, whether you include a prefix for pointers, static data, or global data, and how you indicate that something is a private field of a class.

Some of the potential benefits that can be obtained by adopting a naming convention include the following:

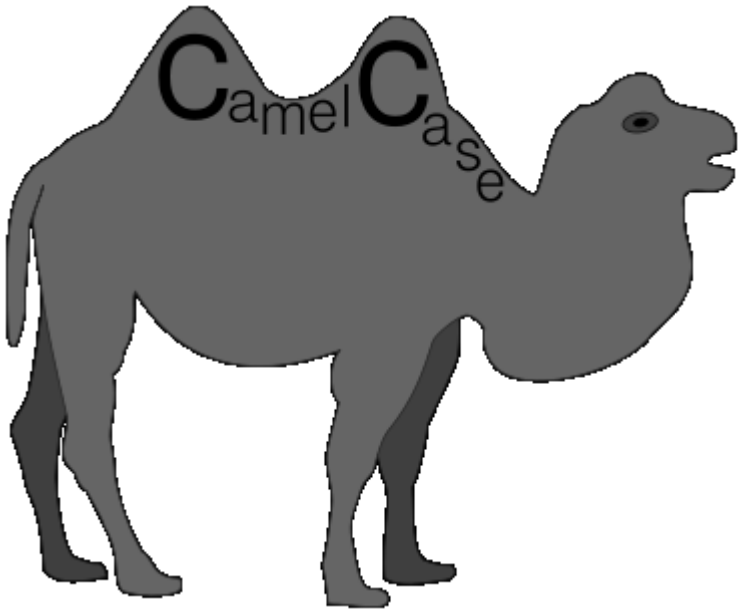
- to provide additional information (i.e., metadata) about the use to which an identifier is put;
- to help formalize expectations and promote consistency within a development team;
- to enable the use of automated refactoring or search and replace tools with minimal potential for error;
- to enhance clarity in cases of potential ambiguity;
- to enhance the aesthetic and professional appearance of work product (for example, by disallowing overly long names, comical or "cute" names, or abbreviations);
- to help avoid "naming collisions" that might occur when the work product of different organizations is combined (see also: namespaces);
- to provide meaningful data to be used in project handovers which require submission of program source code and all relevant documentation and
- to provide better understanding in case of code reuse after a long interval of time.

✓ **Camel Notation**

One popular convention is that leading capital letter camelCase, is used for the names of structs and classes, while normal camelCase is used for the names of functions and variables.

Is the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound and

the first letter is either upper or lower case—as in "CoolSky", BlueColor, "McDonald" or "iPod". The name comes from the uppercase "bumps" in the middle of the compound word, suggestive of the humps of a camel. In programming, it is called Pascal case if the first letter is capitalized and camel case otherwise



- ✓ **Hungarian Notation**

Hungarian notation has commonly been associated with prefixing variables with information about their type--for instance, whether a variable is an integer or a double. This is usually not a useful thing to do because your IDE will tell you the type of a variable, and it can lead to bizarre and complicated looking names. The original idea behind Hungarian notation, however, was more general and useful: to create more abstract "types" that describe how the variable is used rather than how the variable is represented. This can be useful for keeping pointers and integers from intermixing, but it can also be a powerful technique for helping to separate concepts that are often used together, but that should not be mixed.

[Windows programming model follows HUNGARIAN NOTATION]

Chapter 2: A skeletal windows application

Skeleton is the program in which all basic components of all windows applications. A window application program is designed to illustrate the following concepts:

1. Using the WinMain function
2. Initializing a window program
3. Writing message loop.
4. Writing a window function
5. Writing About dialog box.
6. Terminating the application
7. Creating a resource definition file
8. Building a windows application

WNDCLASSEX Structure

Contains window class information. It is used with the RegisterClassEx and GetClassInfoEx functions. The WNDCLASSEX structure is similar to the WNDCLASS structure. There are two differences. WNDCLASSEX includes the cbSize member, which specifies the size of the structure, and the hIconSm member, which contains a handle to a small icon associated with the window class.

```
typedef struct tagWNDCLASSEX {
  UINT      cbSize;
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCTSTR   lpszMenuName;
  LPCTSTR   lpszClassName;
  HICON     hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

Members

cbSize - The size, in bytes, of this structure. Set this member to sizeof(WNDCLASSEX). Be sure to set this member before calling the GetClassInfoEx function.

Style - The class style(s). This member can be any combination of the Class Styles.

lpfnWndProc - A pointer to the window procedure. You must use the CallWindowProc function to call the window procedure. For more information, see WindowProc.

cbClsExtra - The number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

cbWndExtra - The number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses WNDCLASSEX to register a dialog box created by using the CLASS directive in the resource file, it must set this member to DLGWINDOWEXTRA.

hInstance - A handle to the instance that contains the window procedure for the class.

hIcon - A handle to the class icon. This member must be a handle to an icon resource. If this member is NULL, the system provides a default icon.

hCursor - A handle to the class cursor. This member must be a handle to a cursor resource. If this member is NULL, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

hbrBackground - A handle to the class background brush. This member can be a handle to the physical brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following HBRUSH types:

- COLOR_ACTIVEBORDER
- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE
- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER

- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is unregistered by using UnregisterClass. An application should not delete these brushes. When this member is NULL, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the WM_ERASEBKGND message or test the fErase member of the PAINTSTRUCT structure filled by the BeginPaint function.

lpszMenuName - Pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the MAKEINTRESOURCE macro. If this member is NULL, windows belonging to this class have no default menu.

lpszClassName - A pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be in the low-order word of lpszClassName; the high-order word must be zero.

If lpszClassName is a string, it specifies the window class name. The class name can be any name registered with RegisterClass or RegisterClassEx, or any of the predefined control-class names. The maximum length for lpszClassName is 256. If lpszClassName is greater than the maximum length, the RegisterClassEx function will fail.

hIconSm - A handle to a small icon that is associated with the window class. If this member is NULL, the system searches the icon resource specified by the hIcon member for an icon of the appropriate size to use as the small icon.

! IMP – ***This is base program for all windows based programs.***

The Skeleton Application Source Program

```
#include<windows.h>
LRESULT CALLBACK WindowFunc(HWND,UINT,WPARAM,LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmdMode)
{
        HWND hwnd;
        MSG msg;
        WNDCLASSEX wcl;
        char szWinName[]="Skeleton";

        //Define a window class for our application
        wcl.cbSize=sizeof(WNDCLASSEX);
        wcl.hInstance=hInstance;
        wcl.lpszClassName=szWinName;
        wcl.lpfnWndProc=WindowFunc;
        wcl.style=CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS;
        wcl.hIcon=LoadIcon(NULL,IDI_INFORMATION);//alt tab
        wcl.hIconSm=LoadIcon(NULL,IDI_WINLOGO);//task bar
        wcl.hCursor=LoadCursor(NULL,IDC_ARROW);
        wcl.lpszMenuName=NULL;
        wcl.cbClsExtra=0;
        wcl.cbWndExtra=0;
        wcl.hbrBackground=(HBRUSH) GetStockObject (BLACK_BRUSH);

        //Register the window class
        if(!RegisterClassEx(&wcl))
                return FALSE;

        //Creating a window
        hwnd=CreateWindow(
                szWinName,
                "First Window",
```

```
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                NULL,
                NULL,
                hInstance,
                NULL
                );
        ShowWindow(hwnd,nShowCmdMode);
        UpdateWindow(hwnd);
```

//Windows Message Loop

```
while(GetMessage(&msg,NULL,0,0))
        {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
        }
                return msg.wParam;
        }
```

//Window Function

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,WPARAM wParam,LPARAM lParam)
{
                HDC hdc;
                PAINTSTRUCT ps;
                int x=0,y=0;
                char outputStr[50]="This is my first text print in VC++.";
                switch(message){
                case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                        TextOut(hdc,x,y,outputStr,strlen(outputStr));
```

```
            EndPaint (hwnd,&ps);
            break;
        case WM_CLOSE:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return 0;
}
```

[Note: Write this code in Microsoft Visual C++ 6.0 and view the output. Refer 'Lab sheet 1' for detailed Instructions.]

Components of the Skeleton application

The 'WinMain' Function : Program Entry Point

WinMain() is windows equivalent of main() from DOS or UNIX (C & C++). This is where your program starts execution.The Entry Point of Windows Program is 'WinMain', which is declared in winbase.h header file. Function name, Number of Parameters & its Data types must be same except parameter names. The function declaration is as follows:

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

Parameters

HINSTANCE hInstance : This parameter is handle to the programs executable module (the .exe file in memory). This handle uniquely identifies the program. It is required as an argument to some other Windows function calls. hInstance is used for things like loading resources and any other task which is performed on a per-module basis. A module is either the EXE or a DLL loaded into your program. In early versions of Windows, when you run the same program concurrently more than once, you created multiple instances of that program. All instances of the same application uses shared code and read-only memory.

HINSTANCE hPrevInstance : A program could determine if other instances of itself were running by checking the hPrevInstance parameter. Always NULL for Win32 programs. This is used for backward compatibility only.

LPSTR lpCmdLine : This parameter of WinMain is the command line (*e.g. Typing winword in 'Run' to execute the Microsoft Word*) used to run the program. Some Windows applications use this to load a file into memory when the program is started from command line.

int nCmdShow : This parameter to WinMain indicates how the program should be initially displayed—either normally or maximized to fill the window, or minimized to be displayed in the taskbar. This is an integer value which may be passed to ShowWindow() later on.

Registering the Windows Class

A window is always created based on a window class. The window class identifies the window procedure (window function) that processes messages to the window. More than one window can be created based on a single window class. The window class defines the window procedure and some other characteristics of the windows that are created based on that class. When you create a window, you define additional characteristics of the window that are unique to that window. Before you create an application window, you must register a window class by calling RegisterClassEx. This function requires a single parameter, which is a pointer to a structure of type WNDCLASSEx.

```
Char ClassName[] = "myWindowClass";
WNDCLASSEX wcl;
```

// Size of the structure

```
wcl.cbSize=sizeof(WNDCLASSEX);
```

// The instance handle of the program (which is one of the parameters to WinMain)

```
wcl.hInstance=hInstance;
```

//Name of the class which is used in CreateWindow function.

```
wcl.lpszClassName=szWinName;
```

// Name of the Window Function which is used to process all the events [messages] generated in the application.

```
wcl.lpfnWndProc=WindowFunc;
```

// These identifiers used indicate that all windows created based on this class are to be completely repainted whenever the horizontal window size (CS_HREDRAW) or the

vertical window size (CS_VREDRAW) changes. If you resize the window, the text string is redrawn.

```
wcl.style=CS_HREDRAW|CS_VREDRAW;
```

// Sets an icon for all windows created based on this window class. The icon is a small bitmap picture that represents the program to the user. When the program is running, the icon appears in the Windows taskbar and at the left side of the program window's title bar.

To obtain a handle to a predefined icon, you call LoadIcon with the first argument set to NULL. When you're loading your own customized icons that are stored in your program's .EXE file on disk, this argument would be set to hInstance, the instance handle of the program.

The second argument identifies the icon. For the predefined icons, this argument is an identifier beginning with the prefix IDI ("ID for an Icon") defined in WINUSER.H. The IDI_APPLICATION icon is simply a little picture of a window. The LoadIcon function returns a handle to this icon on success.

//alt + tab Icon

```
wcl.hIcon=LoadIcon(NULL,IDI_INFORMATION);
```

//Taskbar

```
wcl.hIconSm=LoadIcon(NULL,IDI_WINLOGO);
```

// Loads a predefined mouse cursor known as IDC_ARROW and returns a handle to the cursor. This handle is assigned to the hCursor field of the WNDCLASS structure. When the mouse cursor appears over the client area of a window that is created based on this class, the cursor becomes a small arrow.

```
wcl.hCursor=LoadCursor(NULL,IDC_ARROW);
```

//Menu name for the application. If the application has no menu NULL is given.

```
wcl.lpszMenuName=NULL;
```

// these 2 fields are used to reserve some extra space in the class structure and the window structure that Windows maintains internally:

```
wcl.cbClsExtra=0;
wcl.cbWndExtra=0;
```

// Background color of the client area of windows created based on this class. The hbr prefix of the hbrBackground field name stands for "handle to a brush." A brush is a graphics term that refers to a colored pattern of pixels used to fill an area. Windows has several standard, or "stock," brushes. The GetStockObject call shown here returns a handle to a white brush:

```
wcl.hbrBackground=(HBRUSH) GetStockObject (BLACK_BRUSH);
```

// After filling the structure values we must register those values to the windows. This is notification to the windows about the window we want to create.

```
if(!RegisterClassEx(&wcl))
        return FALSE;
```

//Registering the class with the filled values of the structure return TRUE on success FALSE on failure

```
if(!RegisterClassEx(&wcl))
{
  MessageBox(NULL, "Window Registration Failed!", "Error!",
    MB_ICONEXCLAMATION | MB_OK);
  return 0;
}
```

// Creating Window:

HWND hwnd;

hwnd = **CreateWindow**(Window Class Name, The title/caption of window, Window Style, x-position, y-Position, Width, height, Parent window's Handle, handle of the Menu, Programs Instance handle, Creation parameter pointer-normally set to NULL. This parameter can be used to point to some data that we might later want to reference in our program.);

CreateWindow

Creates an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu.

```
HWND WINAPI CreateWindow(
        LPCTSTR lpClassName,
        LPCTSTR lpWindowName,
```

```
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

Parameters

lpClassName - A null-terminated string or a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be in the low-order word of lpClassName; the high-order word must be zero. If lpClassName is a string, it specifies the window class name. The class name can be any name registered with RegisterClass or RegisterClassEx, provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined system class names. For a list of system class names, see the Remarks section.

lpWindowName - The window name. If the window style specifies a title bar, the window title pointed to by lpWindowName is displayed in the title bar. When using CreateWindow to create controls, such as buttons, check boxes, and static controls, use lpWindowName to specify the text of the control. When creating a static control with the SS_ICON style, use lpWindowName to specify the icon name or identifier. To specify an identifier, use the syntax "#num".
dwStyle - The style of the window being created. This parameter can be a combination of the window style values, plus the control styles indicated in the Remarks section.

x - The initial horizontal position of the window. For an overlapped or pop-up window, the x parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, x is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area. If this parameter is set to CW_USEDEFAULT, the system selects the default position for the window's upper-left corner and ignores the y parameter. CW_USEDEFAULT is valid only for overlapped windows; if it is specified for a pop-up or child window, the x and y parameters are set to zero.

y - The initial vertical position of the window. For an overlapped or pop-up window, the y parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, y is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box, y is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

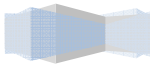
If an overlapped window is created with the WS_VISIBLE style bit set and the x parameter is set to CW_USEDEFAULT, then the y parameter determines how the window is shown. If the y parameter is CW_USEDEFAULT, then the window manager calls ShowWindow with the SW_SHOW flag after the window has been created. If the y parameter is some other value, then the window manager calls ShowWindow with that value as the nCmdShow parameter.

nWidth - The width, in device units, of the window. For overlapped windows, nWidth is either the window's width, in screen coordinates, or CW_USEDEFAULT. If nWidth is CW_USEDEFAULT, the system selects a default width and height for the window; the default width extends from the initial x-coordinate to the right edge of the screen, and the default height extends from the initial y-coordinate to the top of the icon area. CW_USEDEFAULT is valid only for overlapped windows; if CW_USEDEFAULT is specified for a pop-up or child window, nWidth and nHeight are set to zero.

nHeight - The height, in device units, of the window. For overlapped windows, nHeight is the window's height, in screen coordinates. If nWidth is set to CW_USEDEFAULT, the system ignores nHeight.

hWndParent - A handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows. To create a message-only window, supply HWND_MESSAGE or a handle to an existing message-only window.

hMenu - A handle to a menu, or specifies a child-window identifier depending on the window style. For an overlapped or pop-up window, hMenu identifies the menu to be used with the window; it can be NULL if the class menu is to be used. For a child window, hMenu specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about



events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

hInstance - A handle to the instance of the module to be associated with the window.

lpParam - A pointer to a value to be passed to the window through the CREATESTRUCT structure (lpCreateParams member) pointed to by the lParam param of the WM_CREATE message. This message is sent to the created window by this function before it returns.

If an application calls CreateWindow to create a MDI client window, lpParam should point to a CLIENTCREATESTRUCT structure. If an MDI client window calls CreateWindow to create an MDI child window, lpParam should point to a MDICREATESTRUCT structure. lpParam may be NULL if no additional data is needed.

Return Value

If the function succeeds, the return value is a handle to the new window. If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- if one of the controls in the dialog template is not registered, or its window window procedure fails WM_CREATE or WM_NCCREATE

CreateWindowEx

```
HWND WINAPI CreateWindowEx(
        DWORD dwExStyle,
        LPCTSTR lpClassName,
        LPCTSTR lpWindowName,
        DWORD dwStyle,
        int x,
        int y,
        int nWidth,
        int nHeight,
        HWND hWndParent,
```

```
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

Parameters

dwExStyle - The extended window style of the window being created.

Some Extended Window Styles.

- **WS_EX_ACCEPTFILES**: The window accepts drag-drop files.
- **WS_EX_APPWINDOW**: Forces a top-level window onto the taskbar when the window is visible.
- **WS_EX_CLIENTEDGE**: The window has a border with a sunken edge.
- **WS_EX_CONTROLPARENT**: The window itself contains child windows that should take part in dialog box navigation. If this style is specified, the dialog manager recurses into children of this window when performing navigation operations such as handling the TAB key, an arrow key, or a keyboard mnemonic.
- **WS_EX_DLGMODALFRAME**: The window has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.
- WS_EX_LEFT: The window has generic left-aligned properties. This is the default.
- **WS_EX_LEFTSCROLLBAR:** If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the vertical scroll bar (if present) is to the left of the client area. For other languages, the style is ignored.
- **WS_EX_LTRREADING:** The window text is displayed using left-to-right reading-order properties. This is the default.
- **WS_EX_MDICHILD**: The window is a MDI child window.
- **WS_EX_NOPARENTNOTIFY:** The child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.
- **WS_EX_OVERLAPPEDWINDOW:** The window is an overlapped window.
- **WS_EX_TOOLWINDOW: T**he window is intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB. If a tool window has a

system menu, its icon is not displayed on the title bar. However, you can display the system menu by right-clicking or by typing ALT+SPACE.

- ***More remaining…***

Other parameters are same as that of **CreateWindow** function.

```
// Return the handle of window on success and NULL on failure
  if(hwnd == NULL)
  {
    MessageBox(NULL, "Window Creation Failed !!!", "Error!!!", MB_ICONEXCLAMATION | MB_OK);
     return 0;
  }
```

Window Style

The mostly used window style is WS_OVERLAPPEDWINDOW. It will have a title bar; a system menu button to the left of the title bar; a thick window-sizing border; and minimize, maximize, and close buttons to the right of the title bar. That's a standard style for windows, which appears as the "window style" parameter in CreateWindow. This style is a combination of several bit flags:

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED |
                             WS_CAPTION |
                             WS_SYSMENU |
                             WS_THICKFRAME |
                             WS_MINIMIZEBOX|
                             WS_MAXIMIZEBOX)
```

Displaying the Window

After the CreateWindow call returns, the window has been created internally in Windows. Windows has allocated a block of memory to hold all the information about the window, plus some other information, all of which Windows can find later based on the window handle. However, the window does not yet appear on the video display. Two more calls are needed.

The first is:

ShowWindow (hwnd, CmdShow);

The first argument is the handle to the window. The second argument is the CmdShow value passed as a parameter to WinMain. This determines how the window is to be initially displayed on the screen, whether it's normal, minimized, or maximized. The value you receive from WinMain and pass to ShowWindow is SW_SHOWNORMAL if the window is displayed normally, SW_SHOWMAXIMIZED if the window is to be maximized, and SW_SHOWMINNOACTIVE if the window is just to be displayed in the taskbar.

The ShowWindow function puts the window on the display.

Normally the client area of a window is updated when Windows has nothing else to do. When no more messaged await an application and the window needs to be redrawn, Windows sends the window function a message requesting that it update the client area of the window. We can also force this update explicitly at any time by calling the following function:

UpdateWindow (hwnd);

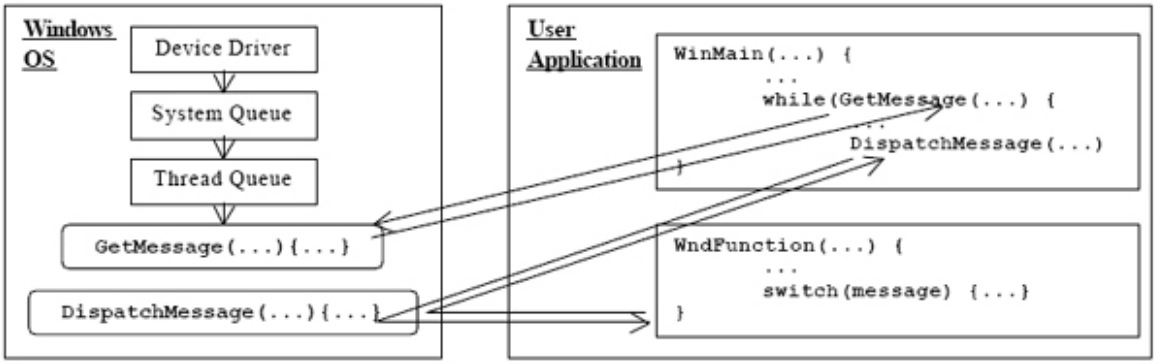
It redraws the content of the client area by sending the window procedure a WM_PAINT message.

Winmain's Message Loop

After the 'UpdateWindow' call, the window is fully visible on the video display. The program must now make itself ready to get input from the user. Windows maintains a "message queue" for each Windows program currently running under Windows. When an input event occurs, Windows translates the event into a "message" that it places in the program's message queue. A program retrieves these messages from the message queue by executing a block of code known as the "***message loop***".

The main message loop for an application is at the bottom of the WinMain() function. The GetMessage() function fetches the message from the message queue, its followed by TranslateMessage() and DispatchMessage(). TranslateMessage() translates the events to windows readable format. DispatchMessage() sends the message data to the WndProc() which then handles the message and act accordingly.

```
while(GetMessage(&Msg, NULL, 0, 0))
  {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
  }
```



The msg variable is a of the type MSG, which is structure & defined in the WINUSER.H header file like this:

```
struct  MSG
{
        HWND hwnd;     //the message is for this window
        UINT message; //this is the actual message number
        WPARAM wParam;      //additional info
        LPARAM lParam          //additional info
        DWORD time; // the time at which the message was posted
        POINT pt ; // the position of the cursor when the msg was posted
}
```

The POINT data type is yet another structure, defined in the WINDEF.H header file like this:

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


In the beginning of the message loop the GetMessage retrieves a message from the message

queue. This call passes to Windows a pointer to a MSG structure named msg. The second parameter specifies the handle of the window for which to retrieve a message. If it is NULL, then it retrieves messages for any window of the application calling this function. The third and the fourth arguments specify the lowest and highest integer message number to retrieve. E.g. if only the mouse messages are to be retrieved, then these values are set as WM_MOUSEFIRST and WM_MOUSELAST. If both are 0, then every kind of messages are retrieved.

- ***hwnd*** - The handle to the window which the message is directed to.
- ***message*** - The message identifier. This is a number that identifies the message. For each message, there is a corresponding identifier defined that begins with the identifier WM ("Window Message"). E.g. WM_KEYDOWN, WM_PAINT, WM_COMMAND, WM_LBUTTONDOWN.
- ***wParam*** - A 32-bit "message parameter," the meaning and value of which depend on the particular message.
- ***lParam*** Another 32-bit message parameter dependent on the message.
- ***time*** - The time the message was placed in the message queue.
- ***pt*** - The mouse coordinates at the time the message was placed in the message queue.

If the message field of the message retrieved from the message queue is anything except WM_QUIT, GetMessage returns a nonzero value. A WM_QUIT message causes GetMessage to return 0. When the GetMessage return 0 the message loop will be terminated and the termination of the message is the termination of the program.

The statement TranslateMessage(&msg); passes the msg structure back to Windows for translation.

The statement DispatchMessage(&msg); again passes the msg structure back to Windows. Windows then sends the message to the appropriate window procedure for processing. After Windows Procedure processes the message, it returns control to Windows. When Windows returns to the program the message loop continues with the next GetMessage call until the WM_QUIT message is posted.

[*Note: All the calls to Window procedure are in the form of messages. Everything that happens to a window is relayed to the window procedure in the form of a message. The window procedure then responds to this message or passes the message to DefWindowProc for default processing.*]**

Overview of Windows Messages

Message processing is at the heart of what makes the windows application work. Many messages in windows originate from devices. Depressing and releasing a key on the keyboard generates interrupt that are handled by the keyboard device driver. Moving the mouse and clicking mouse buttons generates interrupts that are handled by the mouse device driver. These device drivers call windows to translate the hardware event into a message. The resulting message is then placed unto the windows system queue.

There are 2 types of Queues in Windows

1. System Queue
2. Thread Queue

There is only one System Queue. Hardware events that are converted into messages are placed into the system queue.

Each running windows application has its own unique thread queue. Windows transfers the messages in the system queue to the appropriate thread queue. Each of a program's thread queues holds all messages for all windows running in a particular thread. Windows implements the sharing of the shared resources (such as the mouse and the keyboard) by using the system queue. When any event occurs a message is placed into the system queue. Windows then must in effect decide which thread queue should receive the message. How windows does this can vary on depending on the event. Windows uses the concept of input focus to decide which thread queue should receive the message. The input focus is an attribute processed by only one window at a time. Keyboard messages are moved from the system queue in to the thread queue for the thread with a window which currently has input focus. As the input focus moves from window to window, windows moves keyboard messages from the system queue to the proper thread queue.

Thread (Application)

- All the messages go through system queue to the corresponding thread queues
- A program can have multiple threads

- Each thread of execution has a unique thread queue
- The messages in a thread queue are processed in FIFO order
- The system queue decides the receiving thread queue by the input focus of the shared resources (e.g. keyboard, mouse, timer)
- In mouse messages, some app's may capture the mouse
- A program can call the Windows OS so as to place a message in its thread queue
- The application programs retrieves the messages from the OS using a Message Loop
- The message loop continuously retrieves messages from the application queue and dispatches them to the corresponding window function

---

How windows works: Windows, Events & Messages

The simplified version of the working of windows involves three key concepts: windows, events and messages. Window is a simple rectangular region with its own boundaries. We are aware of many types of windows: Explorer Window, Document window of Word Processor, Dialog Box. There are other many types of windows. A command button is also a window, Icons, Text Boxes, Menu; Scrollbars etc. are all referred as window.

Every window that we create in visual programming has a unique identifier called handle. A handle is a 32 bit integer value assigned to a window. Microsoft Windows Operating system manages many windows by assigning a unique ID number, that is 'Handle'. The system continually monitors each of those windows for different activities or events. Events can occur through user actions such as mouse click or key press, through programmatic control or even as a result of another window's actions. Every window has message handlers which process messages from specific window.

Each time an event occurs, it causes a message to be sent to the OS. The system processes the message and broadcasts it to all other windows. Each window can then take the appropriate action based on its own instructions for dealing with the particular message (e.g. repainting itself when it has been uncovered by another window.)

A message handler is actually very simple. The handler receives a message and responds to it or discards it, depending on the design of your application. A message handler accepts a message and either responds to it or pass it on another handler. Message is just a command to tell the window to do something. When window receives a message it has option to respond or ignore. It depends upon the programming you have done on it.

Message Flow

As we have covered about different types of Queues in windows. It will be easier to understand the message flow. Windows generates message for every hardware event – such as when the user presses a key on a keyboard or moves a mouse. It passes these messages to the appropriate thread message queue. Each thread in the system processes the messages in its own message queue. If a message is destined for specific thread, the message is placed in the thread's message queue. Some messages are system-wide or are destined for multiple threads. These messages are placed in the queue of the appropriate threads.

Contd. ...

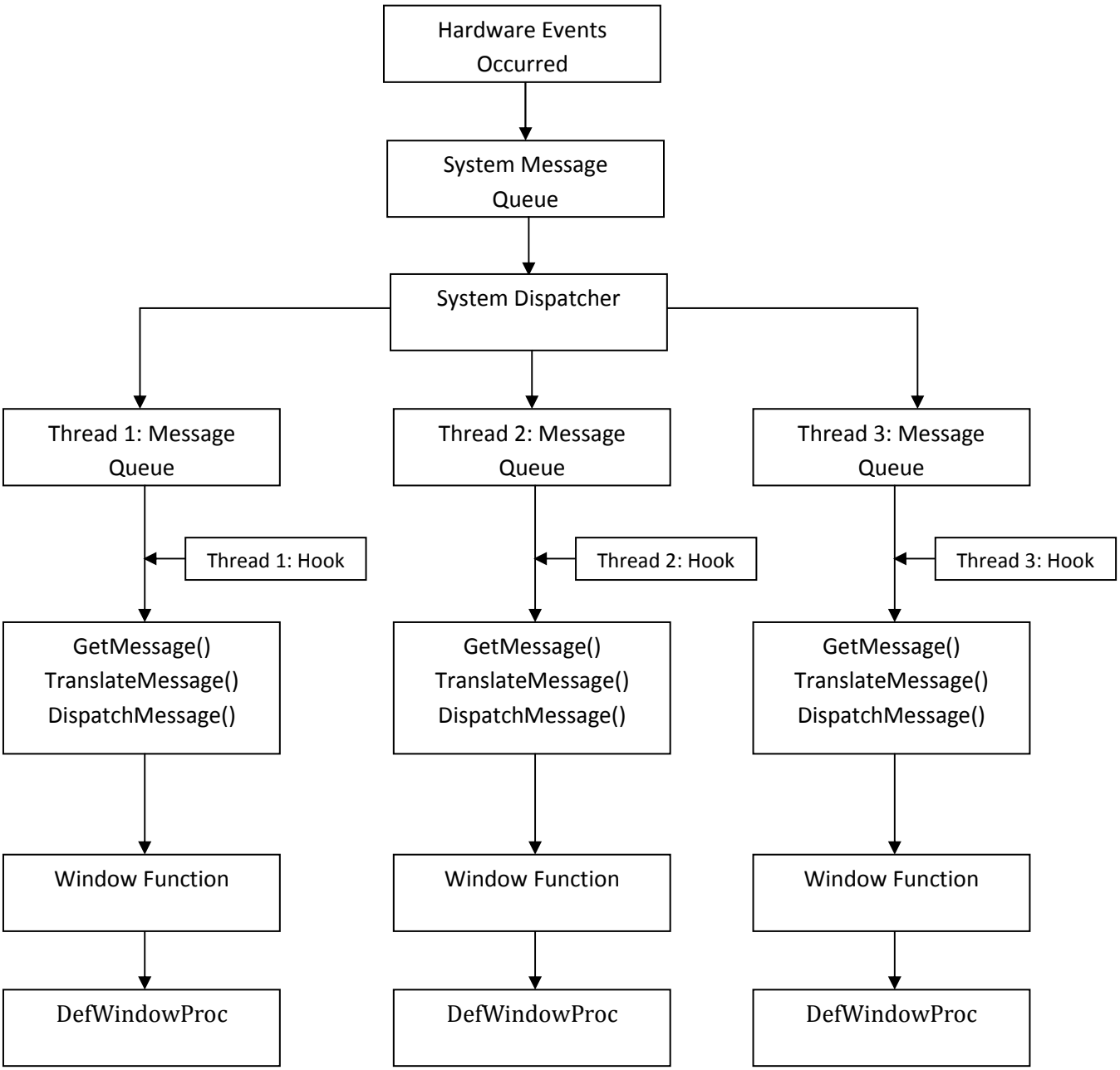


Fig: Windows Message Flow Diagram

The message data contains the window handle (hwnd), the coded message type (msg), the wParam and lParam data that will be passed to the WndProc(...) function, the time when the message is sent (in milliseconds after windows started) and the POINT (pt) structure which contains the x and y coordinates of the mouse cursor when the message was sent.

Some Messages:

- ✓ WM_CHAR
  - o This message is sent to a control when a key is pressed on the keyboard.

- ✓ WM_CLEAR
  - This message is sent to an EDIT control to instruct it to clear the current selection
- ✓ WM_KEYDOWN
  - This message is posted to window when a non-system key is pressed.
- ✓ WM_KEYUP
  - This message is received when a key is released on the keyboard.
- ✓ WM_QUIT
  - This message is sent to window to request that it is to be destroyed.

Windows Procedure

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  switch(msg)
  {
    case WM_CLOSE:
      DestroyWindow(hwnd);
    break;
    case WM_DESTROY:
      PostQuitMessage(0); // When 'Message Loop' gets 0 or WM_QUIT or similar message then the
                          window will be terminated.
    break;
    default:
      return DefWindowProc(hwnd, msg, wParam, lParam);
  }
  return 0;
}
```

Chapter 03: Device Context

Introduction to Device and Display Context

Windows supports a number of different types of devices for both input and output. The two most common devices are the display screen and the printer, although windows also support other devices such as plotter and image acquisition devices.

The basic tool that windows use to provide device independent for an application is called a device context of DC. The DC is an internal structure that windows use to maintain information about device. Instead of sending output directly to hardware, an application sends it to the DC, and then windows send it to the hardware.

A Device context is like the canvas of an artist. It is the actual object you draw on windows. DC's can be created for display as well as printers. A device context is block of memory managed by GDI. It is the data structure that contains information that GDI uses to determine how to perform the drawing.

All **GDI* f**unctions require a device context as a parameter. A device context is a link between our application, a **device driver*** and essential output device, such as Printer of Plotter.GDI functions are called to request certain output operations. GDI forwards those requests (which are **device independent***) to the particular device driver associated with the device.

---

* ***Graphical Device Interface [GDI]*** is a Microsoft Windows application programming interface and core operating system component responsible for representing graphical objects and transmitting them to output devices such as monitors and printers. GDI is responsible for tasks such as drawing lines and curves, rendering fonts and handling palettes. It is not directly responsible for drawing windows, menus, etc.; that task is reserved for the user subsystem, which resides in user32.dll.)

---

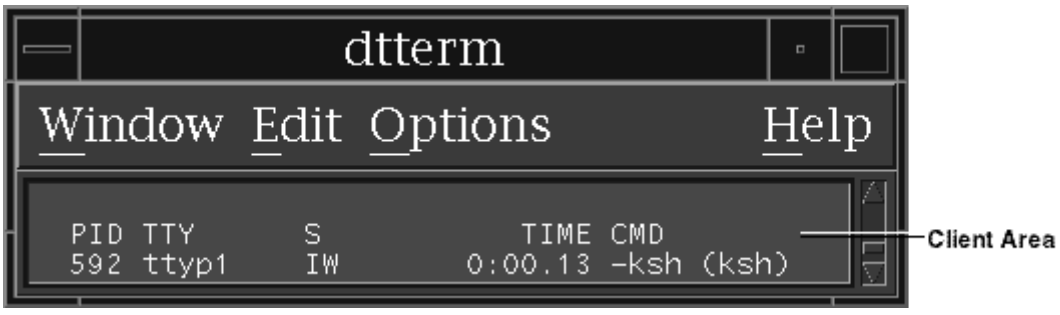
* ***Device Driver****:* a device driver or software driver is a computer program allowing higher-level computer programs to interact with a hardware device. This is a program that controls a device. Every device, whether it is a printer, disk drive, or keyboard, must have a driver program. Many drivers, such as the keyboard driver, come with the operating system. For other devices, you may need to load a new driver when you

connect the device to your computer. In DOS systems, drivers are files with .SYS extension. In Windows environments, drivers often have .DRV extension. Or also come as exe files. A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

*** Device independent:** A program is said to be device independent when its function is universal on different types of device. This refers to programs that work with a variety of peripheral devices. This generally means that it is written in a framework that can be read by devices from any companies. A program that was not originally written for a certain environment can be ported, i.e. the code can be adapted for a certain platform and compiled for the platform it will be functioning in. Unfortunately, this can lead to confusion if the user interface still resembles the one for the platform it was initially designed for.

Device independence is the process of making a software application be able to function on a wide variety of devices regardless of the local hardware on which the software is used.

*** Client Area:** The client area is the inside area of window excluding the title bar, toolbars, status bar, scroll bars. For example in case of Internet Explorer the area in which the web pages are displayed.



A DC for graphical display allows to write anywhere on the device, even on the top of another application's window. Instead we generally use a **Display Context;** it treats each window as a separate display surface. An application using a display context for a window can write anything it wants in that window. It cannot, however, access or draw over any part of the system display that lies outside the window.

A device context handle is like all other handles in windows. It is simply a token that windows give, that represents an object; in this case a Display Context. We need to obtain display context handle before writing anything. Display contexts generally should be allocated as needed and released as quickly as possible.

The device context provides the mechanism by which clipping and coordinate transformations take place. That is the Black-Box, graphics and text output commands go in one side, are processed and transferred according to the attributes that are selected for the device context and then are sent to the actual window or device.

In any type of drawing on a graphics output device, we must first obtain a handle to a device context (or DC). The device context contains many "attributes" that determine how the GDI functions work on the device. When you call TextOut function, you need to specify in the function only the device context handle, the starting coordinates, the text, and the length of the text. You don't need to specify the font, the color of the text, the color of the background behind the text, or the character spacing. These are all attributes that are part of the device context.

Windows provides several methods for obtaining a device context handle.

Device Context API Functions

BeginPaint	Obtain the client area DC in a WM_PAINT handler.
CreateDC	Create a device context for the specified device. It is most often used to create a DC for printer.
CreateCompatibleDC	Creates a memory device context that is compatible to a source DC. A memory device context can be considered a simulation of a device in a memory. Selecting a bitmap into the device allows creation of a memory image that is compatible with the device.
DeleteDC	Deletes a created device context. This should be used to free device contexts created using the **CreateDC** & **CreateCompatibleDC** function.
EndPaint	Release the DC obtained by BeginPaint function.

GetDC	Obtain the client are DC other than in a WM_PAINT handler.
GetWindowDC	This function is similar to GetDC, except that it retrieves a device context for the entire window rectangle (not only the client area of the window)
ReleaseDC	Release the DC obtained by GetDC or GetWindowDC.

The most common method for obtaining a device context handle and then releasing it involves using the ***BeginPaint*** and ***EndPaint*** calls when processing the WM_PAINT message:

```
HDC hdc;
PAINTSTRUCT ps;
hdc = BeginPaint (hwnd, &ps) ;
        // [Other program lines]
EndPaint (hwnd, &ps) ;
```

The variable ps is a structure of type PAINTSTRUCT. The hdc field of this structure is the handle to the device context that BeginPaint returns. The PAINSTRUCT structure also contains a RECT (rectangle) structure named rcPaint that defines a rectangle encompassing the region of the window's client area. With the device context handle obtained from BeginPaint you can draw only within this region.

PAINTSTRUCT Structure

```
typedef struct tagPAINTSTRUCT
{
        HDC hdc;
        BOOL fErase;
        RECT rcPaint;
        BOOL fRestore;
        BOOLfIncUpdate;
        BYTE rgpReserved[32];
} PAINTSTRUCT;
```

The PAINTSTRUCT structure contains information that can be used to paint the client area of the window.

Members:

hdc: Identifies the DC to be used for painting.

fErase: Specifies whether the background need to be redrawn. It is not 0 if the application should redraw the background. The application is responsible for drawing the background if a window-class is created without a background brush.

***rcPaint*:** specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.

RECT Structure

```
typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

Members

left: x-coordinate of the upper-left corner of a rectangle

top: y-coordinate of the upper-left corner of a rectangle

right: x-coordinate of the lower-right corner of a rectangle

bottom: y-coordinate of the lower-right corner of a rectangle

***fRestore*:** Reserved internally by windows.

***fIncUpdate*:** Reserved internally by windows.

***rgbReserved*[32]:** Memory Block reserved internally by windows .

BeginPaint function updates the PAINTSTRUCT structure **ps**; with information about

the paint request. It also returns the handle to a display context for the window. The **EndPaint** function ends the paint request and releases the display context back to windows.

Windows programs can also obtain a handle to a device context with another function:

```
hdc = GetDC (hwnd) ;
        // [Other program lines]
ReleaseDC (hwnd, hdc) ;
```

This device context applies to the whole client area of the window whose handle is hwnd. You can draw on your entire client area with the handle returned from GetDC. But with if the DC is obtained from BeginPaint function then the drawing is only the area which the RECT represents.

A Windows program can also obtain a handle to a device context that applies to the entire window (Client and Non-Client Area *(window title bar, menu, scroll bars, and frame)*).

```
hdc = GetWindowDC (hwnd) ;
        [other program lines]
ReleaseDC (hwnd, hdc) ;
```

Displaying output upon receipt of a WM_PAINT Message

WM_PAINT message is used to carry out the drawing necessary for displaying information. The system sends WP_PAINT messages to the application when the application window needs to be updated. This can be also called explicitly.

Windows sends a WM_ERASEBKGND message to the window function when processing a WP_PAINT message. The DefWindowProc function process the WM_ERASEBKGND message by filling the affected area using the class background brush. This erases any output that may have drawn previously. Windows places a WM_PAINT message in an application's queue whenever the client area of the window needs to be repainted.

A window can receive a WM_PAINT message whenever one of the following events occurs:

- A previously obscured area of the window becomes visible.
- The user resizes the window.
- The contents of the client area of the window are explicitly invalidated via a call to either the **InvalidateRect** or **InvalidateRgn.**

A well designed windows application concentrates all display output in the WM_PAINT message processing rather than sprinkling output statements through the program. When any part of the program wishes to redraw a portion of a window, it should simply notify windows that the particular area of the window needs to be updated. As a result of this notification, windows will eventually send a WM_PAINT message to the window function. The WP_PAINT message processing logic than redraw the areas in need of update.

The system is not the only source of WM_PAINT messages. The **InvalidateRect** or **InvalidateRgn** function can indirectly generate WP_PAINT messages in application. These functions mark all or part of the client area as invalid that must be drawn.

Drawing in the Client Area

BeginPaint and EndPaint functions are used to prepare for and complete the drawing in the client area. Beginpaint returns a handle to the display device context used for drawing in the client area; EndPaint ends the paint request and releases the device context.

To make sure the string is visible when the window is first created, the WinMain function calls UpdateWindow immediately after creating and showing the window. This causes a WM_PAINT message to be sent immediately to the window procedure.

An update region defines the part of the client area that needs to be repainted on the next WM_PAINT message, we should call either **InvalidateRect** or **InvalidateRgn** function to remove a given rectangle or region from the update region. If there is no update region remaining, windows will remove the WM_PAINT message from the queue.

NOTE:*If you decide to update a specific part of the window before receiving a WM_PAINT message, you should call the* **InvalidateRect** or **InvalidateRgn** *functions to remove a given rectangle or region from the update region. If there is no update region remaining, windows will remove the WM_PAINT message from the queue.*

Windows does not actually place a WM_PAINT message in the application's message queue; instead it sets a flag indicating that WM_PAINT message is needs to be sent to the appropriate window function. Message can be posted to the message queue after windows sets the WM_PAINT message pending flag. The GetMessage() function will retrieve any message waiting in the message queue. When there are non, the GetMessage() function looks in the system queue for pending mouse and keyboard message for the application. Finally windows checks to see if any timer event that requires a WM_PAINT message has occurred. Only if no message is found with the GetMessage() function checks the pending WM_PAINT message flag set. The GetMessage() function synthesizes a WM_PAINT message and returns it. This means that WM_PAINT is the lowest-priority message in the system.

[***Note**: You can have your application redraw the entire contents of the client area whenever the window changes size by setting the **CS_HREDRAW and CS_VREDRAW** styles for the window class. Applications that adjust the size of the drawing based on the size of the window use these styles to ensure that they start with a completely empty client area when drawing.*]

- ✓ **ValidateRect**

The **ValidateRect** function validates the client area within a rectangle by removing the rectangle from the update region of the specified window.

```
BOOL ValidateRect
(
        HWND hwnd,          // Handle of a window
        CONST RECT *lpRect  // Address of the validation rectangle coordinates.
);
```

hwnd: Handle to the window whose update region is to be modified. If this parameter is NULL, the system invalidates and redraws all windows and sends the WM_ERASEBKGND and WM_NCPAINT messages to the window procedure before the function returns.

lpRect: Pointer to a RECT structure that contains the client coordinates of the rectangle to be removed from the update region. If this parameter is NULL, the entire client area is removed.

Return Values

If the function succeeds the return value is nonzero. If fails then return value is zero.

- ✓ **ValidateRgn**

The ValidateRgn function validates the client area within a region by removing the region from the current update region of the specified window.

```
BOOL ValidateRgn(
        HWND hwnd,          // Handle of Window
        HRGN hRgn           // Handle of valid region
);
```

hwnd: Handle to the window whose update region is to be modified.

hRgn: Handle to a region that defines the area to be removed from the update region. If this parameter is NULL then the entire client area is removed.

Return Values

If the function succeeds the return value is nonzero. If fails then return value is zero.

- ✓ **InvalidateRgn**

The InvalidRgn function invalidates the client area within the specified region by adding it to the current update region of a window. The invalidated region, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs.

```
BOOL InvalidateRgn(
        HWND hwnd,   // Handle to window with changed update region
        HRGN hRgn,   // Handle of the region to add
        BOOL bErase  // Erase Background Flag
        );
```

hwnd: Handle to the window with an update region that is to be modified

hRgn: Handle the region to be added to the update region. The region is assumed to have client coordinates. If this parameter is NULL, the entire client area is added to the update region.

bErase: Specifies whether the background within the update region should be erased when the update region is processed. If this parameter is TRUE, the background is erased when the BeginPaint() is called. If the parameter is FALSE, the background remains unchanged.

Return Values

The return value is always non zero.

- ✓ **InvalidateRect**

The InvalidateRect function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn.

```
BOOL InvalidateRect(
        HWND hwnd,
        CONST RECT *lpRect,
        BOOL bErase
        );
```

hwnd: Handle to the window whose update region has changed. If this parameter is NULL, the system invalidates and redraws all windows and sends the WM_ERASEBKGND and WM_NCPAINT message to the window procedure before the function returns.

lpRect: Pointer to the RECT structure that contains the client coordinates of the rectangle to be added to the update region. If this parameter is NULL, the entire client area is added to the update region.

bErase: Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased when the BeginPaint function is called. If this parameter is FALSE, the background remains unchanged.

Return Values

If the function succeeds the return value is nonzero. If fails then return value is zero.

Logical Coordinates and Device Coordinates

Within limits, we can define a logical coordinate system to be any coordinate system that is convenient for the application. We can specify the size of a unit in the logical coordinate system. We can specify the size in term of physical dimension, pixels or scaling factors. We can also specify the orientation of the axes of the logical coordinate system. The coordinate values are

limited from -32, 768 to + 32, 767. Windows NT allows using full 32 bit coordinates.

The ability to draw figures using logical coordinates is extremely helpful to a windows programmer. An application intended for drawing building plans would naturally use a physical measurement coordinate system.

To draw on the device, GDI must convert the logical coordinates into device coordinates. Device coordinates aren't nearly as flexible as logical coordinates. The device coordinate system often depends on the actual device and sometimes even the operating mode of the device.
All display device and most printers use the following device coordinate system:

- The origin is located at the upper-left corner of the display.
- Device units are equivalent to pixels, increasing horizontal device coordinates proceed from left to right across the device surface.
- Increasing vertical device coordinate proceed from the top of the device surface to the bottom (downwards).

GDI translate (maps) logical coordinates into device coordinates by using the mapping mode attribute of a device context. Windows provides a number of mapping modes that we can use to specify how a logical coordinate to be mapped to its equivalent device coordinates.

GDI Functions for Information Display in Window

- **DrawText**
    - The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines etc.).

```
int DrawText(
        HDC hdc,            // Handle to DC
        LPCSTR lpString,    // Text to Draw
        int ncount;         // Text Length
        LPRECT lpRect,      // Formatting Dimensions
        UINT uFormat        // Text- Drawing options
        );
```

uFormat parameter options:

Specifies the method of formatting the text. This parameter can have one or more of the following values:

Value	Description
DT_BOTTOM	Justifies the text to the bottom of the rectangle. This value is used only with the DT_SINGLELINE value.
DT_CENTER	Centers the text horizontally in the rectangle.
DT_LEFT	Aligns the text to left.
DT_EDITCONTROL	Duplicates the text displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for and edit control and the function does not display a partially visible last line.
More ... **Please refer text book for more parameters. !IMP**	

- **TextOut**
  - The TextOut function writes a character string at the specified location, using the currently selected font, background color and text color.

```
BOOL TextOut(
        HDC hdc,            // Handle to DC
        int nXstart         // x-Coordinate of starting position
        int nYstart,        // y-coordinate of starting position
        LPCTSTR lpString,   // Character String
        Int cbString        // Number of Characters
        );
```

- **SetTextColor**
  - The SetTextColor function sets the text color for the specified device context to the specified color.

```
COLORREF SetTextColor(
```

```
HDC hdc,            // Handle to DC
COLORREF crColor    // Text Color
);
```

Return Values

If the function succeeds, the return value is color reference for the previous text color as a COLORREF value. If the function fails, the return value is CLR_INVALID.

- **SetTextAlign**

The SetTextAlign function sets the text-alignment flags for the specified device context.

```
UINT SetTextAlign(
    HDC hdc,        // Handle to DC
    UINT fMode      // Text alignment Option
    );
```

fMode: Specifies the text alignment by using a mask of the values in the following list. Only one flag can be chosen from those that affect horizontal and vertical alignment.

Values	Meaning
TA_BASELINE	The reference point will be on the base line of the text.
TA_BOTTOM	The reference point will be on the bottom edge of the bounding rectangle.
TA_TOP	The reference point will be on the top edge of the bounding rectangle.
TA_CENTER	The reference point will be aligned horizontally with the center of the bounding rectangle.
TA_LEFT	The reference point will be on the left edge of the bounding rectangle.
TA_RIGHT	The reference point will be on the right edge of the bounding rectangle.

TA_NOUPDATE	The current position is not updated after each text output call. The reference point is passed to the text output function.
TA_RTLREADING	**Middle East Language edition of Windows:** The text is laid out in right to left reading order, as opposed to the default left to right order. This applies only when the font selected into the device context is either Hebrew or Arabic
TA_UPDATECP	The current position is updated after each text output call. The current position us used as the reference point.

When the current font has a vertical default base line, as Kanji (Japanese), the following values must be used instead of TA_BASELINE and TA_CENTER.

Value	Meaning
VTA_BASELINE	The reference point will be on the base line of the text.
VTA_UPDATECP	The reference point will be aligned vertically with the center of the bounding rectangle.

Default Values are TA_LEFT, TA_TOP & TA_NOUPDATECP.

Return Values

If the function succeeds, the return value is previous text-alignment setting. If the function fails, the return value is GDI_ERROR.

Types of display context

A display context has one these general forms.

- A display context for the client area of the window.
- A display context for a window (which includes non client area such as the title bar and frame)
- A display context for the desktop window (the display surface of the display device)

Display context for the window client area are the most commonly used; to support rich variety of options to maximize programmer's performance. Windows supports four types of display context:

1. Common Display Context
2. Class Display Context
3. Private Display Context
4. Window Display Context

1. Common Display Context

The common display context is the default type of display context given to a window. If display context type is not specified in a window's class style parameter, windows assign a common display context to the window. This is the easiest way to get a display context for a window. A common display context is the most frequently used form of display context.

When using a common display context, windows resets common display context to default values each time it retrieves a common display context. However this requires you to explicitly set each attribute of a common display context that you want each time you retrieve the display context.

Attribute	Default Value
Background Color	RGB(255, 255, 255) – White
Background Mode	OPAQUE
Bitmap	No default value. Applies only to memory DCs.
Brush	WHITE
Brush Origin	(0, 0)
Clipping Region	Windows sets the clipping region to the entire client area. Update region is clipped if necessary. Child and Pop-Up windows overlaying also be clipped.
Color-Palette	DEFAULT-PALETTE
Current Pen Position	(0, 0)
Device Origin	Upper-Left corner of the client area.
Drawing Mode	R2-COPYPEN

Font	SYSTEM-FONT
Graphics Mode	GM_COMPATIBLE
InterCharacter Spacing	0
Mapping mode	MM_TEXT
Miter Limit	10.0
Pen	BLACK_PEN
Polygon filling mode	ALTERNATE
Relative-Absolute Flag	ABSOLUTE
Text Color	RGB (0, 0, 0) – Black
Text Alignment	TA_TOP | TA_LEFT | TA_NOUPDATE
Viewport extent	(1, 1)
Viewport Origin	(0, 0)
Window Extent	(1, 1)
Window Origin	(0, 0)

2. Class Display Context

We can indicate that a window uses a class display context by specifying the CS_CLASSDC class style we register the window class. Windows creates a class display context, initialized to the default values as shown above.

A Class display context is not created fresh each time. It is allocated and initialized precisely once for use by all windows of the class. Because a class display context is not a common display context, the requirement to release the display context after using it does not apply to class display context.

You must retrieve a class display context before using it. A class display context is retrieved by using the GetDC or BeginPaint function.

Retrieving a class display context sets the device origin and clipping attributes of the display context to the appropriate values for the window that retrieves it. Basically, the act of retrieving the class display associates it with the window. These attributes remain unchanged as long as no other window of the same class retrieves the display context.

When a second window of the class retrieves the class display context, the device origin and clipping attributes are changed to appropriate values for the second window. The original window will need to retrieve the class display context once more before using it again in order to switch the device origin and clipping attributes back to the proper values for that window.

You don't need to retrieve a class display context each time it's used if you can guarantee that no other window of the same class has retrieved the display context since the first window retrieved it.

A change made to the class display context by one window affects all the other windows that subsequently use that display context.

3. Private Display Context

A private display context is a display context permanently associated with only one window. You can indicate that a window uses a private display context by specifying the CS_OWNDC class style when you register the window class. Because each window has its own separate display context, you need to retrieve private display context only once. Windows automatically updates the device origin and clipping region attributes of the context any time the window is moved or resized.

With this class style the display context exists for the life of the window. The application still uses GetDC() & BeginPaint() functions to retrieve a handle to display context but doesn't need to call ReleaseDC() or EndPaint() function after using the device context.

A Private Display Context is never retrieved after retrieving it once. A private display context is given same default values as that of Common display context. Subsequent changes to a private display context are preserved until they are explicitly changed.

4. Window Display Context

A window display context allows to draw anywhere in the window [client + non-client area (Title Bar, Menus, Scroll Bars)]. A window display context has its origin at the upper-left corner of the window not the upper-left corner of the client area. Window Display is retrieved by calling the **GetWindowDC**() function.

Windows retrieves a window display context the same as it does for a common display context. As with a common display context, display context must be released with **ReleaseDC()** when all GDI calls are finished.

A attributes of a window display context are set to the default values as Common display context. The CS_OWNDC and CS_CLASSDC class styles apply only to display context for the client area.

Attributes of a Device Context

Windows uses these attributes when you draw on an output device using the device context.

- Proper drawing object to use (Pen and Brushes)
- The proper font in which to display
- The foreground and background color of the text
- The orientation and scaling of the coordinate system used to specify an object's position and size.
- The area on the output device to which drawing is permitted (the clipping region)
- The color palette to use.

Color Attributes

Background and foreground color of a window can be changes with GDI calls. GDI uses background color attribute as the color used to fill the empty space around text in character cell. Background color of a window is specified when we register the window class by specifying color value, as follows:

```
wcl.hbrBackground = (HBRUSH)(COLOR_WINDOW+1)
        or
wcl.hbrBackground=GetStockBrush(WHITE_BRUSH)
```

Windows uses the class background brush when erasing the invalid area of a window. It uses the background color attribute of a device context when drawing with styled lines and hatched brushes and displaying text when the background mode is set to OPAQUE.

- **SetBkColor**

The SetBkColor function sets the current background color to the specified color value, or to the nearest physical color if the device cannot represent the specified color value.

```
COLORREF SetBkColor(
        HDC hdc,
        COLORREF crColor
        );
```

- **GetBkColor**

This function returns the current background color for the specified device context.

```
COLORREF GetBkColor(
        HDC hdc,
        );
```

- **SetBkMode**

This function sets the background mix mode of the specified device context. The background mix mode is used with text, hatched brushes, and with non-solid pen styles.

```
int SetBkMode(
        HDC hdc,
        int iBkMode
        );
```

hdc: Handle to device context

iBkMode: Specifies the background mode. It is either of the following values

- **OPAQUE**
- **TRANSPARENT**

Return values

If the function succeeds, the return value specifies the previous background mode. If the function fails, the return value is zero. To get the error information, GetLastError() function is used.

E.g. ***SetBkMode(hdc, TRANSPARENT);***

- **GetBkMode**

  The GetBkMode function returns the current background mix mode for a specified device context. The background mix mode of a device context affects text, hatched brushes, and pen styles that are not solid lines.

  ```
  int GetBkMode(
          HDC hdc
  );
  ```

  Parameter

  hdc - Handle to the device context whose background mode is to be returned.

  Return Value

  - If the function succeeds, the return value specifies the current background mix mode, either OPAQUE or TRANSPARENT.
  - If the function fails, the return value is zero.

Defining a color

Windows provides palettes to allow applications to display pictures with different colors than the default colors that windows provides. It is common for video cards to have at least 256 colors that windows provide. However, this is not enough color to display full color pictures with reasonable accuracy. Full color pictures can require as many as 16 million colors for an accurate representation. Palettes allow an application to change the default colors to provide the best possible set of colors that will represent the picture most accurately.

Color can be specified in three ways:

1. **As an explicit RGB value**
2. **As an index to the appropriate logical palette entry**
3. **As a palette-relative RGB value.**

Contd. ...

COLORREF Layout

Macro	31 24	23 16	15 8	7 0
RGB (R, G, B)	0	b	g	r
PALETTEINDEX (n)	1	0	n	
PALETERGB(R, G, B)	2	b	g	r

1. Explicit RGB Color Value

For explicitly defining color value should use the RGB macro defined by **windows.h.** Since each intensity occupies 1 byte, intensity can vary from 0 to 255. Therefore the hexadecimal value 0x00FF0000 specifies an explicit RGB value of blue at its maximum intensity, with no green and no red.

```
COLORREF crColor;
crColor=RGB(0, 0, 255);
SetTextColor(hdc, crColor)
        Or
SetTextColor(hdc, RGB(0, 0, 255))
```

The GetRValue, GetGValue and GetBValue macros extract the Red, Green or Blue intensity values respectively from the supplied RGB color value.

The SetTextColor() and SetBkColor() function pass an explicit RGB value directly to the device driver for the device associated with the specified device context. The DC actually uses that what is nearest available color on the device; it might be exact also.

If we required knowing what is nearest color to an explicit **RGB** device can represent, you use the **GetNearestColor()** function. It returns an RGB color value indentifying the device's solid color that is closest to the specified color.

E.g. crColor=GetNearestColor(hdc, RGB(128, 64, 192));

2. Palette – Index Color Values

Color can also be specified by indicating which entry in a palette of colors contains the color to be used. Many devices can display large number of colors. Typical maximum values are 262,

144 or 16, 777, 216. However only relatively small set of the colors can be displayed simultaneously – 256 colors at one time is a frequent limit. You create a logical color palette that lists the colors you want to use chosen from the colors available on the device. Then, rather than specify an explicit RGB value for a color, you supply the number of palette entry that contains the color to use. Palette table entries are numbered beginning at 0. The number of palette table entry is called a palette index because it used to index into the table colors.

PALETTEINDEX macro returns a COLORREF value the RGB macro does. However, the internal contents of the COLORREF value differ. The high order byte contains 1 and low-order word of the COLOREF value contains the index into the logical palette. Therefore the GetRValue, GetGValue and GetBValue macros cannot be used to extract individual value.

```
COLORREF PALETTEINDEX(
        WORD wPaletteIndex);
```

wPaletteIndex: Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return Values

The return value is logical palette index specifier.

3. Palette Relative Color Values

Colors can also be specified as a palette relative RGB color value. Palette relative RGB is specified by supplying the red, green and blue intensities of the color as in explicit RGB. The PALETTERGB macro assembles these intensities into the same format as in explicit color value but encodes the high order byte as 2. Therefore the **GetRValue(),GetGValue()** and **GetBValue()** macros can be used to extract the individual color value and palette index color value.

A palette relative color value is quite handy. This is a cross between an explicit RGB and a palette index color value. When the device on which we are drawing supports logical palette, GDI matches the palette relative RGB color value to nearest color in the logical palette. Specifying a palette relative color value however is not completely equivalent to specifying the

palette index color value to which it is nearest in color. The entries in the logical palette can be changed. A recently added palette entry might more closely match the color specified in a previously used palette relative color value.

PALETTERGB

The PALETTERGB macro accepts three values that represent the relative intensities of Red, Green and Blue and returns a palette-relative Red, Green & Blue specifier consisting of 2 in the high-order byte and an RGB value in the three low-order bytes. An application using a color palette can pass this identifier, instead of an explicit RGB value, to functions that expect a color.

```
COLORREF PALETTERGB(
        BYTE bRed,        // Intensity of Red Color in palette-relative RGB
        BYTE bGreen,      // Intensity of Green Color in palette-relative RGB
        BYTE bBlue        // Intensity of Blue Color in palette-relative RGB
        );
```

Return Values

The return value is a palette-relative RGB specifier. For output devices that supports logical palettes, the system matches a palette-relative RGB value to the nearest color in the logical palette of the device context as though the application had specified an index to that palette entry. If an output device does not support a system palette, the system uses the palette relative RGB as though it were a conventional RGB value returned by the RGB macro.

Drawing Tools

Tool Type	Description
Brush	The brush tool defines the pattern and colors that windows uses to fill the interior of the drawn figures.
Pen	The pen tool defines the width, color and style of lines that windows uses to outline drawn figures.
Font	The font tool identifies the font that windows will use when drawing.
Palette	The palette tool specifies the colors, that window can use to draw on a

	given device.
Region	A region defines an area on a device.
Bitmap	Blocks of pixels data that can be output directly to a device, such as video display.

All of these GDI objects have a number of similarities. Tools are selected by calling **SelectObject**() function and passing the handle to the new object. Windows defines macro API equivalent functions [**SelectBitmap()**, **SelectBrush**(), **SelectFont**(), **SelectPen**()] to select Bitmap, Brush, Font and Pen respectively into a device context. These macros are defined in the ***windowsx.h*** header file. All these GDI objects are deleted by calling the **DeleteObject**() function and passing it the handle of the object.

Windows defines macro API functions (**DeleteBrush**, **DeleteFont**, **DeletePalette**, **DeletePen** & **DeleteRgn**) to delete previously created brush, palette, pen and region respectively.

Bitmaps

Bitmaps are blocks of pixel data that can be output directly to a device, such as video display. They can be thought of as a way to store the pixel data directly from the screen into the memory buffer. Painting bitmaps onto the screen is much faster than using GDI functions like **Rectangle()** and **LineTo()**.

A particular bitmap cannpt be selected into more than one memory device context at one time. Device contexts other than memory device contexts have no default selected bitmap because they can't have a bitmap selected into them. Resources allocated bitmap is released by calling **DeleteObject()** function.

BITMAP Structure

```
typedef struct tagBITMAP{
        int bmType;
        int bmWidth;
        int hmHeight;
        int bmWidthBytes;
```

```
    BYTE bmPlanes;
    BYTE bmBitsPixel;
    LPVOID bmBits;
} BITMAP;
```

Resource Script

```
Pen     BITMAP      "pen.bmp"
```

Sample Code

```
HBITMAP hBitmap;
HDC hdc;
HDC hMemDC;

hBitmap=LoadBitmap(hInstance, "Pen" );
hdc=GetDC(hwnd);
hMemDC=CreateCompatibleDC(hdc);

SelectObject(hMemDC, hBitmap);

BitBlt(hdc, 10, 10, 60, 60, hMemDC, 0, 0, SRCCOPY);

DeleteDC(hMemDC);
ReleaseDC(hdc);
DeleteObject(hBitmap);
```

The above example loads a file names 'pen.bmp' and gives the bitmap resource name 'Pen'. The application uses LoadBitmap() function to bring the bitmap into memory. The application loads the bitmap into a memory device context, and then paints the memory device context onto the screen with the BitBlt() function.

Fonts

A call to the GetTextMetrics() function returns metrics based on the font currently selected in to the device context. All sizes in the TEXTMETRIC structure is given in logical units that depends

upon the current mapping mode of the device context. You must select a desired font into a device context before you retrieve its metrics. It's also necessary to establish the desired mapping mode, so that the returned metrics are in proper units.

TEXTMETRIC Structure

```
typedef struct tagTEXTMETRIC {
        LONG tmHeight;
        LONG tmAscent;
        LONG tmDescent;
        LONG tmInternalLeading;
        LONG tmExternalLeading;
        LONG tmAveCharWidth;
        LONG tmMaxCharWidth;
        LONG tmWeight;
        LONG tmOverhang;
        LONG tmDigitizedAspectX;
        LONG tmDigitizedAspectY;
        char tmFirstChar;
        char tmLastChar;
        char tmDefaultChar;
        char tmBreakChar;
        BYTE tmItalic;
        BYTE tmUnderlined;
        BYTE tmStruckOut;
        BYTE tmPitchAndFamily;
        BYTE tmCharSet;
} TEXTMETRIC;
```

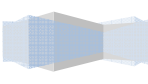
Members

tmHeight

Specifies the height (ascent descent) of characters.

tmAscent

Specifies the ascent (units above the base line) of characters.



tmDescent

Specifies the descent (units below the base line) of characters.

tmInternalLeading

Specifies the amount of leading (space) inside the bounds set by the tmHeight member. Accent marks and other diacritical characters may occur in this area. The designer may set this member to zero.

tmExternalLeading

Specifies the amount of extra leading (space) that the application adds between rows. Because this area is outside the font, it contains no marks and is not altered by text output calls in either OPAQUE or TRANSPARENT mode. The designer may set this member to zero.

tmAveCharWidth

Specifies the average width of characters in the font (generally defined as the width of the letter x). This value does not include the overhang required for bold or italic characters.

tmMaxCharWidth

Specifies the width of the widest character in the font.

tmWeight

Specifies the weight of the font.

tmOverhang

Specifies the extra width per string that may be added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, graphics device interface (GDI) or a device may have to add width to a string on both a per-character and per-string basis. For example, GDI makes a string bold by expanding the spacing of each character and over striking by an offset value; it italicizes a font by shearing the string. In either case, there is an overhang past the basic string. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the amount the top of the font is sheared past the bottom of the font.

The tmOverhang member enables the application to determine how much of the character width returned by a ***GetTextExtentPoint32*** function call on a single character is the actual character width and how much is the per-string extra width. The actual width is the extent minus the overhang.

tmDigitizedAspectX

Specifies the horizontal aspect of the device for which the font was designed.

tmDigitizedAspectY

Specifies the vertical aspect of the device for which the font was designed. The ratio of the tmDigitizedAspectX and tmDigitizedAspectY members is the aspect ratio of the device for which the font was designed.

tmFirstChar

Specifies the value of the first character defined in the font.

tmLastChar

Specifies the value of the last character defined in the font.

tmDefaultChar

Specifies the value of the character to be substituted for characters not in the font.

tmBreakChar

Specifies the value of the character that will be used to define word breaks for text justification.

tmItalic

Specifies an italic font if it is nonzero.

tmUnderlined

Specifies an underlined font if it is nonzero.

tmStruckOut

Specifies a strikeout font if it is nonzero.

tmPitchAndFamily

Specifies information about the pitch, the technology, and the family of a physical font.

The four low-order bits of this member specify information about the pitch and the technology of the font. A constant is defined for each of the four bits.

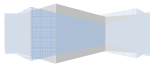
Value	Description
TMPF_FIXED_PITCH	If this bit is set the font is a variable pitch font. If this bit is clear the font is a fixed pitch font. Note that those meanings are the opposite of what the constant name implies.
TMPF_VECTOR	If this bit is set, the font is a vector font.
TMPF_TRUETYPE	If this bit is set, the font is a TrueType font.
TMPF_DEVICE	If this bit is set, the font is a device font.

The four high-order bits of tmPitchAndFamily designate the font's font family. An application can use the value 0xF0 and the bitwise AND operator to mask out the four low-order bits of tmPitchAndFamily, thus obtaining a value that can be directly compared with font family names to find an identical match. For information about font families, see the description of the LOGFONT structure.

tmCharSet

Specify the character set of the font. The character set is one of the following values:

ANSI_CHARSET	DEFAULT_CHARSET
SYMBOL_CHARSET	SHIFTJIS_CHARSET
HANGUL_CHARSET	GB2312_CHARSET
CHINESEBIG5_CHARSET	OEM_CHARSET
JOHAB_CHARSET	HEBREW_CHARSET
ARABIC_CHARSET	GREEK_CHARSET
TURKISH_CHARSET	VIETNAMESE_CHARSET
THAI_CHARSET	EASTEUROPE_CHARSET
RUSSIAN_CHARSET	MAC_CHARSET
BALTIC_CHARSET	



GetTextMetrics() function

```
BOOL GetTextMetrics(
 HDC hdc,
 LPTEXTMETRIC lptm
);
```

Parameters

hdc - Handle to the device context (DC).

lptm - Long pointer to the TEXTMETRIC structure that receives the metrics.

Return Value

Nonzero indicates success. Zero indicates failure. To get extended error information, call GetLastError.

Pens

Pens are used for drawing in Device context. Pen can be either selected from the stock or can be created but before using, it must be selected. And we should also delete the pen after its use.

Creating Pens

The appearance of line is determined by the pen you use when drawing the line. Windows draws a line using the pen that is currently selected into the device context when you call the LineTo, PolyLine or similar drawing functions. The default pen is windows stock object named BLACK_PEN. This pen draws a black line that is 1px wide regardless the current mapping mode. Windows has two more stock pens: WHITE_PEN & NULL_PEN. A white pen draws a white line that is 1 pixel wide regardless of the current mapping mode. A NULL_PEN has no ink so it does not leave any mark when any drawing is done with it.

GetStockPen macro API is to get the handle to one of the stock pens. After getting the handle the pen must be selected with SelectPen macro API.

e.g.

```
HPEN hpen;
hpen=GetStockPen(WHITE_PEN);
SelectPen(hdc, hpen);
```

Following steps are involved while creating, using and deleting own custom pen:

- Create a logical pen using the CreatePen(), CreatePenIndirect() or ExtCreatePen() function.
- Select the logical pen into a device context by calling any of the drawing functions.
- Draw the lines using this pen by calling any of the drawing functions.
- Select either the original pen or stock pen into the device context by calling the SelectPen() function or by using RestoreDC() specifying a context that was saved before the SelectPen() Operation.
- Delete the logical pen by calling the DeleteObject() macro API. This is done only after completion of all drawing operations.

CreatePen()

This function creates a logical pen that has the specified style, width, and color. The pen can subsequently be selected into a device context and used to draw lines and curves.

```
HPEN CreatePen(
 int fnPenStyle,
 int nWidth,
 COLORREF crColor
);
```

Parameters

fnPenStyle - Specifies the pen style. It can be any one of the following values.

Value	Description
PS_SOLID	Pen is solid.
PS_DASH	Pen is dashed. This style is valid only when the pen width is one or less in device units.
PS_DOT	Pen is dotted. This style is valid only when the pen width is one or less in device units.
PS_DASHDOT	Pen has alternating dashes and dots. This style is valid only when the pen width is one or less in device units.

PS_DASDOTDOT	Pen has alternating dashes and double dot. This style is valid only when the pen width is one or less in device units.
PS_NULL	Pen is invisible.
PS_INSIDEFRAME	Pen is solid. When this pen is used in any graphics device interface GDI drawing function that takes a bounding rectangle, the dimension of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to geometric pens.

Fig: Pen Styles

nWidth - Specifies the width of the pen, in logical units. If nWidth is zero, the pen is a single pixel wide, regardless of the current transformation. CreatePen returns a pen with the specified width bit with the PS_SOLID style if you specify a width greater than one for the PS_DASH style.

crColor - Specifies a color reference for the pen color. Colors can be specified by any of the three techniques but it would be easy with explicit RGB.

Return Value

Handle to a logical pen indicates success. NULL indicates failure. To get extended error information, call GetLastError.

Note: After an application creates a logical pen, it can select that pen into a device context by calling **SelectObject() or SelectPen()** function. After a pen is selected into the device context, it can be used to draw lines, curves and other graphical items.

If the value specified by nWidth parameter is zero, a line drawn with the created pen will always be a single pixel wide regardless of the current transformation.
If thevalue specified by nWidth is greater than 1px then the fnPenStyle parameter must be PS_NULL, PS_SOILD or PS_INSIDEFRAME.

If the value specified by nWidth is greater than 1 and fnPenStyle is PS_INSIDEFRAME, the line associated with the pen is drawn inside the frame of all primitives except polygons and polylines.

If the value specified by nWidth parameter is greater than 1, fnPenStyle is PS_INSIDEFRAME, and the color specified by the crColor parameter does not match one of the entries in the logical palette, the system draws lines by using a dithered color. Dithered colors are not available with solid pens.

When you no longer need the pen, call DeleteObject() to delete it.

CreatePenIndirect

This function creates a logical cosmetic pen that has the style, width, and color specified in the LOGPEN structure.

LOGPEN

```
typedef struct tagLOGPEN {
  UINT lopnStyle;
  POINT lopnWidth;
  COLORREF lopnColor;
} LOGPEN;
```

Members

lopnStyle - Specifies the pen style, which is one of the values from 'Fig: Pen Styles' [Page 61].

lopnWidth - Specifies the POINT structure that contains the pen width, in logical units. If the pointer member is NULL, the pen is one pixel wide on raster devices. The y member in the POINT structure for lopnWidth is not used.

lopnColor - Specifies the pen color.

CreatePenIndirect

```
HPEN CreatePenIndirect(
const LOGPEN* lplgpn
);
```

Parameters

lplgpn - Long pointer to the LOGPEN structure that specifies the pen's style, width, and color.

Return Value

A handle that identifies a logical cosmetic pen indicates success. NULL indicates failure. To get extended error information, call GetLastError().

Remarks

After an application creates a logical pen, it can select that pen into a device context by calling the SelectObject() function. After a pen is selected into a device context, it can be used to draw lines and curves. When you no longer need the pen, call the DeleteObject() function to delete it.

E.g.

```
        // this code segment creates a RED DASHED Pen.
LOGPEN lp;
        HPEN hPen;
        lp.lopnStyle=PS_DASH;
        lp.lopnWidth=1;
        lp.lopnColor=RGB(255, 0, 0);
        hPen = CreatePenIndirect(&lp);
```

ExtCreatePen

The ExtCreatePen function creates a logical cosmetic or geometric pen that has the specified style, width, and brush attributes.

```
HPEN ExtCreatePen(
        DWORD dwPenStyle,
        DWORD dwWidth,
        const LOGBRUSH *lplb,
        DWORD dwStyleCount,
        const DWORD *lpStyle
);
```

Parameters

dwPenStyle - A combination of type, style, end cap, and join attributes. The values from each category are combined by using the bitwise OR operator (|).

The pen type can be one of the following values.

Value	Meaning
PS_GEOMETRIC	The pen is geometric.
PS_COSMETIC	The pen is cosmetic.

The pen style can be one of the following values.

Value	Meaning
PS_ALTERNATE	The pen sets every other pixel. (This style is applicable only for cosmetic pens.)
PS_SOLID	The pen is solid.
PS_DASH	The pen is dashed.
PS_DOT	The pen is dotted.
PS_DASHDOT	The pen has alternating dashes and dots.
PS_DASHDOTDOT	The pen has alternating dashes and double dots.
PS_NULL	The pen is invisible.
PS_USERSTYLE	The pen uses a styling array supplied by the user.
PS_INSIDEFRAME	The pen is solid. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to geometric pens.

The end cap is only specified for geometric pens. The end cap can be one of the following values.

Value	Meaning
PS_ENDCAP_ROUND	End caps are round.
PS_ENDCAP_SQUARE	End caps are square.
PS_ENDCAP_FLAT	End caps are flat.

The join is only specified for geometric pens. The join can be one of the following values.

Value	Meaning
PS_JOIN_BEVEL	Joins are beveled.
PS_JOIN_MITER	Joins are mitered when they are within the current limit set by the SetMiterLimit() function. If it exceeds this limit, the join is beveled.
PS_JOIN_ROUND	Joins are round.

dwWidth - The width of the pen. If the dwPenStyle parameter is PS_GEOMETRIC, the width is given in logical units. If dwPenStyle is PS_COSMETIC, the width must be set to 1.

lplb - A pointer to a LOGBRUSH structure. If dwPenStyle is PS_COSMETIC, the lbColor member specifies the color of the pen and the lpStyle member must be set to BS_SOLID. If dwPenStyle is PS_GEOMETRIC, all members must be used to specify the brush attributes of the pen.

dwStyleCount - The length, in DWORD units, of the lpStyle array. This value must be zero if dwPenStyle is not PS_USERSTYLE. The style count is limited to 16.

lpStyle - A pointer to an array. The first value specifies the length of the first dash in a user-defined style; the second value specifies the length of the first space, and so on. This pointer must be NULL if dwPenStyle is not PS_USERSTYLE.

If the lpStyle array is exceeded during line drawing, the pointer is reset to the beginning of the array. When this happens and dwStyleCount is an even number, the pattern of dashes and spaces repeats. However, if dwStyleCount is odd, the pattern reverses when the pointer is reset -- the first element of lpStyle now refers to spaces, the second refers to dashes, and so forth.

Return Value

If the function succeeds, the return value is a handle that identifies a logical pen. If the function fails, the return value is zero.

Remarks

- A geometric pen can have any width and can have any of the attributes of a brush, such as dithers and patterns. A cosmetic pen can only be a single pixel wide and must be a solid color, but cosmetic pens are generally faster than geometric pens.
- The width of a geometric pen is always specified in world units. The width of a cosmetic pen is always 1.
- End caps and joins are only specified for geometric pens.
- After an application creates a logical pen, it can select that pen into a device context by calling the SelectObject function. After a pen is selected into a device context, it can be used to draw lines and curves.
- If dwPenStyle is PS_COSMETIC and PS_USERSTYLE, the entries in the lpStyle array specify lengths of dashes and spaces in style units. A style unit is defined by the device where the pen is used to draw a line.
- If dwPenStyle is PS_GEOMETRIC and PS_USERSTYLE, the entries in the lpStyle array specify lengths of dashes and spaces in logical units.

- If dwPenStyle is PS_ALTERNATE, the style unit is ignored and every other pixel is set.
- If the lbStyle member of the LOGBRUSH structure pointed to by lplb is BS_PATTERN, the bitmap pointed to by the lbHatch member of that structure cannot be a DIB section. A DIB section is a bitmap created by CreateDIBSection. If that bitmap is a DIB section, the ExtCreatePen function fails.
- When an application no longer requires a specified pen, it should call the DeleteObject function to delete the pen.

// sample code that demonstrates the appearance of lines drawn using various pen styles and attributes.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
      {
          PAINTSTRUCT ps;
          LOGBRUSH lb;
          RECT rc;
          HDC hdc;
```

```
int i;
HGDIOBJ hPen = NULL;
HGDIOBJ hPenOld;
DWORD dwPenStyle[] = {
                        PS_DASH,
                        PS_DASHDOT,
                        PS_DOT,
                        PS_INSIDEFRAME,
                        PS_NULL,
                        PS_SOLID
                      };
UINT uHatch[] = {
                   HS_BDIAGONAL,
                   HS_CROSS,
                   HS_DIAGCROSS,
                   HS_FDIAGONAL,
                   HS_HORIZONTAL,
                   HS_VERTICAL
                  };

switch (uMsg)
{
    case WM_PAINT:
    {
        GetClientRect(hWnd, &rc);
        rc.left += 10;
        rc.top += 10;
        rc.bottom -= 10;

        // Initialize the pen's brush.
        lb.lbStyle = BS_SOLID;
        lb.lbColor = RGB(255,0,0);
        lb.lbHatch = 0;

        hdc = BeginPaint(hWnd, &ps);
```

```
        for (i = 0; i < 6; i++)
        {
            hPen = ExtCreatePen(PS_COSMETIC | dwPenStyle[i],
                                1, &lb, 0, NULL);
            hPenOld = SelectObject(hdc, hPen);
            MoveToEx(hdc, rc.left + (i * 20), rc.top, NULL);
            LineTo(hdc, rc.left + (i * 20), rc.bottom);
            SelectObject(hdc, hPenOld);
            DeleteObject(hPen);
        }
        rc.left += 150;
        for (i = 0; i < 6; i++)
        {
            lb.lbStyle = BS_HATCHED;
            lb.lbColor = RGB(0,0,255);
            lb.lbHatch = uHatch[i];
            hPen = ExtCreatePen(PS_GEOMETRIC,
                                5, &lb, 0, NULL);
            hPenOld = SelectObject(hdc, hPen);
            MoveToEx(hdc, rc.left + (i * 20), rc.top, NULL);
            LineTo(hdc, rc.left + (i * 20), rc.bottom);
            SelectObject(hdc, hPenOld);
            DeleteObject(hPen);
        }
        EndPaint(hWnd, &ps);
    }
    break;

    case WM_DESTROY:
        DeleteObject(hPen);
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
```

```
        }

        return FALSE;
    }
```

SetDCPenColor

SetDCPenColor function sets the current device context (DC) pen color to the specified color value. If the device cannot represent the specified color value, the color is set to the nearest physical color.

```
COLORREF SetDCPenColor(
    HDC hdc,
    COLORREF crColor
    );
```

Parameters

***hdc* -** A handle to the DC.

crColor - The new pen color.

Return Value

If the function succeeds, the return value specifies the previous DC pen color as a COLORREF value. If the function fails, the return value is CLR_INVALID.

Remarks

The function returns the previous DC_PEN color, even if the stock pen DC_PEN is not selected in the DC; however, this will not be used in drawing operations until the stock DC_PEN is selected in the DC.

The GetStockObject function with an argument of DC_BRUSH or DC_PEN can be used interchangeably with the SetDCPenColor and SetDCBrushColor functions.

```
// Sample code - setting the Pen or Brush Color
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
```

```
    HDC hdc;

    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:

        {
            hdc = BeginPaint(hWnd, &ps);
        //    Initializing original object
            HGDIOBJ original = NULL;

        //    Saving the original object
            original = SelectObject(hdc,GetStockObject(DC_PEN));

        //    Rectangle function is defined as...
        //    BOOL Rectangle(hdc, xLeft, yTop, yRight, yBottom);

        //    Drawing a rectangle with just a black pen
        //     The black pen object is selected and sent to the current device
context
        //  The default brush is WHITE_BRUSH
            SelectObject(hdc, GetStockObject(BLACK_PEN));
```

```
            Rectangle(hdc,0,0,200,200);

        //    Select DC_PEN so you can change the color of the pen with
        //    COLORREF SetDCPenColor(HDC hdc, COLORREF color)
            SelectObject(hdc, GetStockObject(DC_PEN));

        //    Select DC_BRUSH so you can change the brush color from the
        //    default WHITE_BRUSH to any other color
            SelectObject(hdc, GetStockObject(DC_BRUSH));

        //    Set the DC Brush to Red
        //    The RGB macro is declared in "Windowsx.h"
            SetDCBrushColor(hdc, RGB(255,0,0));

        //    Set the Pen to Blue
            SetDCPenColor(hdc, RGB(0,0,255));

        //    Drawing a rectangle with the current Device Context
            Rectangle(hdc,100,300,200,400);

        //    Changing the color of the brush to Green
            SetDCBrushColor(hdc, RGB(0,255,0));
            Rectangle(hdc,300,150,500,300);

        //    Restoring the original object
            SelectObject(hdc,original);

        // It is not necessary to call DeleteObject to delete stock objects.
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Regions

Regions are more complex than pens. The clipping region attribute of a device context in number of ways.

SelectObject:

HRGN hrgn;

SelectObject(hdc, hrgn);

This can be done with another function:

SelectClipRgn:

SelectClipRgn(hdc, hrgn);

Both the SelectObject() and SelectClipRgn() return a value that specifies the region's type. These functions do not return the previously selected region. One of the side effects of the BeginPaint() API call is to select the invalidated region as being clipping region.

Rectangular area from clipping region can be excluded by specifying the upper-left and lower-right corners of the rectangle when calling the ExcludeClipRgn() function.

ExcludeClipRgn(hdc, xUL, yUL, xLR, yLR);

Similarly, the update region of a window from the clipping region of device context can be excluded by specifying the window when calling ExcludeUpdateRgn() function.

ExcludeUpdateRgn(hdc, hwnd);

Creating and using stock and custom brushes

Figures created with pen can be filled using different brushes. It can be filled whatever color or pattern. The using procedure of brush is same as that of pen. We select a brush in to the device context by calling the SelectObject() or SelectBrush() macro API.

```
HGDIOBJ SelectObject(
        HDC hdc,
        HGDIOBJ hgdiobj
        );
```

E. g. HBRUSH hBrush;

HBRUSH hPrevBrush;

hPrevBrush=SelectBrush(hdc, hBrush);

Stock Brushes

The easiest way to use the brush is selecting it from the stock. The default brush in a device context is stock object BLACK_BRUSH.

The GetStockObject function retrieves a handle to one of the stock pens, brushes, fonts, or palettes.

```
HGDIOBJ GetStockObject(
        int fnObject
);
```

Parameters

fnObject - The type of stock object. This parameter can be one of the following values.

Value	Meaning
BLACK_BRUSH	Black brush.
DKGRAY_BRUSH	Dark gray brush.
GRAY_BRUSH	Gray brush.
HOLLOW_BRUSH	Hollow brush (equivalent to NULL_BRUSH).
LTGRAY_BRUSH	Light gray brush.
NULL_BRUSH	Null brush (equivalent to HOLLOW_BRUSH).
WHITE_BRUSH	White brush.

Return Value

- If the function succeeds, the return value is a handle to the requested logical object. If the function fails, the return value is NULL.

- Use the DKGRAY_BRUSH, GRAY_BRUSH, and LTGRAY_BRUSH stock objects only in windows with the CS_HREDRAW and CS_VREDRAW styles. Using a gray stock brush in any other style of window can lead to misalignment of brush patterns after a window is moved or sized. The origins of stock brushes cannot be adjusted.
- The HOLLOW_BRUSH and NULL_BRUSH stock objects are equivalent. It is not necessary (but it is not harmful) to delete stock objects by calling DeleteObject().

  E.g.

```
HBRUSH hBrush;
hBrush=GetStockBrush(LTGRAY_BRUSH);
```

Custom Brushes

Creating custom brush is lot like creating custom pen. A custom brush is a logical brush, just as a custom pen is a logical pen. So appearance is not fully determined until the brush is selected into a device context.

CreateSolidBrush

The CreateSolidBrush function creates a logical brush that has the specified solid color.

```
HBRUSH CreateSolidBrush(
        COLORREF crColor
);
```

Parameters

crColor - The color of the brush. To create a COLORREF color value, use the RGB macro.

Return Value

If the function succeeds, the return value identifies a logical brush. If the function fails, the return value is NULL.

Remarks

- When you no longer need the HBRUSH object, call the DeleteObject function to delete it.
- A solid brush is a bitmap that the system uses to paint the interiors of filled shapes.
- After an application creates a brush by calling CreateSolidBrush, it can select that brush into any device context by calling the SelectObject function.

Example

Although you can specify any color for a pen when creating it, the system uses only colors that are available on the device. This means the system uses the closest matching color when it realizes the pen for drawing. When creating brushes, the system generates a dithered color if you specify a color that the device does not support. In either case, you can use the RGB macro to specify a color when creating a pen or brush.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    RECT clientRect;
    RECT textRect;
    HRGN bgRgn;
    HBRUSH hBrush;
    HPEN hPen;
    switch (message)
    {
       case WM_PAINT:
        {
        hdc = BeginPaint(hWnd, &ps);

        // Fill the client area with a brush
        GetClientRect(hWnd, &clientRect);
        bgRgn = CreateRectRgnIndirect(&clientRect);
        hBrush = CreateSolidBrush(RGB(200,200,200));
        FillRgn(hdc, bgRgn, hBrush);

        hPen = CreatePen(PS_DOT,1,RGB(0,255,0));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(0,0,0));
        Rectangle(hdc, 10,10,200,200);
```

```
        // Text caption
        SetBkColor(hdc, RGB(255,255,255));
        SetRect(&textRect, 10, 210, 200,200);
        DrawText(hdc,TEXT("PS_DOT"),-1,&textRect, DT_CENTER | DT_NOCLIP);

        hPen = CreatePen(PS_DASHDOTDOT,1,RGB(0,255,255));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(255,0,0));
        SelectObject(hdc,CreateSolidBrush(RGB(0,0,0)));
        Rectangle(hdc, 210,10,400,200);

        // Text caption
        SetBkColor(hdc, RGB(255,200,200));
        SetRect(&textRect, 210, 210, 400,200);
        DrawText(hdc,TEXT("PS_DASHDOTDOT"),-1,&textRect,DT_CENTER|
DT_NOCLIP);

        hPen = CreatePen(PS_DASHDOT,1,RGB(255,0,0));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(255,255,0));
        SelectObject(hdc,CreateSolidBrush(RGB(100,200,255)));
        Rectangle(hdc, 410,10,600,200);

        // Text caption
        SetBkColor(hdc, RGB(200,255,200));
        SetRect(&textRect, 410, 210, 600,200);
        DrawText(hdc,TEXT("PS_DASHDOT"),-1,&textRect,DT_CENTER|DT_NOCLIP);

        // When fnPenStyle is PS_SOLID, nWidth may be more than 1.
        // Also, if you set the width of any pen to be greater than 1,
        // then it will draw a solid line, even if you try to select another style.

        hPen = CreatePen(PS_SOLID,5,RGB(255,0,0));
        SelectObject(hdc, hPen);
        // Setting the background color doesn't matter
        // when the style is PS_SOLID
```

```
SetBkColor(hdc, RGB(255,255,255));
SelectObject(hdc,CreateSolidBrush(RGB(200,100,50)));
Rectangle(hdc, 10,300,200,500);

// Text caption
SetBkColor(hdc, RGB(200,200,255));
SetRect(&textRect, 10, 510, 200,500);
DrawText(hdc,TEXT("PS_SOLID"),-1,&textRect,DT_CENTER | DT_NOCLIP);

hPen = CreatePen(PS_DASH,1,RGB(0,255,0));
SelectObject(hdc, hPen);
SetBkColor(hdc, RGB(0,0,0));
SelectObject(hdc,CreateSolidBrush(RGB(200,200,255)));
Rectangle(hdc, 210,300,400,500);

// Text caption
SetBkColor(hdc, RGB(255,255,200));
SetRect(&textRect, 210, 510, 400,200);
DrawText(hdc,TEXT("PS_DASH"),-1,&textRect, DT_CENTER | DT_NOCLIP);

hPen = CreatePen(PS_NULL,1,RGB(0,255,0));
SelectObject(hdc, hPen);
// Setting the background color doesn't matter
// when the style is PS_NULL
SetBkColor(hdc, RGB(0,0,0));
SelectObject(hdc,CreateSolidBrush(RGB(255,255,255)));
Rectangle(hdc, 410,300,600,500);
// Text caption
SetBkColor(hdc, RGB(200,255,255));
SetRect(&textRect, 410, 510, 600,500);
DrawText(hdc,TEXT("PS_NULL"),-1,&textRect, DT_CENTER | DT_NOCLIP);
// Clean up
DeleteObject(hBrush);
DeleteObject(hPen);
GetStockObject(WHITE_BRUSH);
```

```
        GetStockObject(DC_PEN);
        EndPaint(hWnd, &ps);
        break;
        }
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

CreateHatchBrush

The CreateHatchBrush function creates a logical brush that has the specified hatch pattern and color.

```
HBRUSH CreateHatchBrush(
        int fnStyle,
        COLORREF clrref
);
```

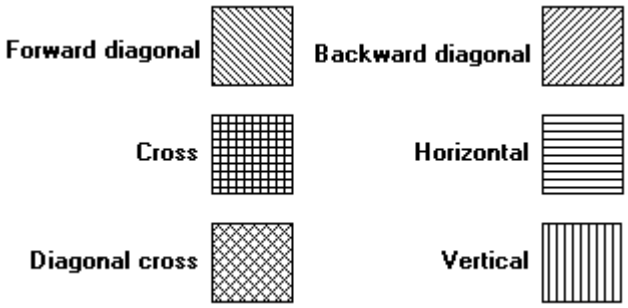
Parameters

fnStyle - The hatch style of the brush. This parameter can be one of the following values.

Value	Meaning
HS_BDIAGONAL	45-degree upward left-to-right hatch
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree downward left-to-right hatch
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

There are six predefined logical hatch brushes maintained by GDI. The following rectangles

were painted by using the six predefined hatch brushes.



clrref - The foreground color of the brush that is used for the hatches. To create a COLORREF color value, use the RGB macro.

Return Value

If the function succeeds, the return value identifies a logical brush. If the function fails, the return value is NULL.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes. After an application creates a brush by calling CreateHatchBrush, it can select that brush into any device context by calling the SelectObject function. It can also call SetBkMode to affect the rendering of the brush.

When you no longer need the brush, call the DeleteObject function to delete it.

DrawIcon

Draws an icon or cursor into the specified device context. To specify additional drawing options, use the DrawIconEx function.

Syntax

```
BOOL WINAPI DrawIcon(
        HDC hDC,
        int X,
        int Y,
        HICON hIcon
);
```

Parameters

hDC - A handle to the device context into which the icon or cursor will be drawn.

X - The logical x-coordinate of the upper-left corner of the icon.

Y - The logical y-coordinate of the upper-left corner of the icon.

hIcon - A handle to the icon to be drawn.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

DrawIcon places the icon's upper-left corner at the location specified by the X and Y parameters. The location is subject to the current mapping mode of the device context. DrawIcon draws the icon or cursor using the width and height specified by the system metric values for icons. ***(For more information : GetSystemMetrics.)***

DrawIconEx

Draws an icon or cursor into the specified device context, performing the specified raster operations, and stretching or compressing the icon or cursor as specified.

Syntax

```
BOOL WINAPI DrawIconEx(
        HDC hdc,
        int xLeft,
        int yTop,
        HICON hIcon,
        int cxWidth,
        int cyWidth,
        UINT istepIfAniCur,
        HBRUSH hbrFlickerFreeDraw,
        UINT diFlags
);
```

Bitmap Brush

Windows paints an area by repainting the bitmap both horizontally and vertically until the entire area is covered. However, it isn't restricted to aligning the upper-left corner of the bitmap with upper-left corner of the device context. The starting alignment of the bitmap with respect to the area to be painted is determined by two factors.

- The brush origin attribute of a device context. The brush origin specifies the pixel within the brush bitmap that windows aligns with the origin of the device context and can be any one of the pixels in the bitmap.
- The second factor is the relative position in device coordinates of the area to be painted with respect to the origin of the device context.

The brush origin can range from 0 to the width of the bitmap minus 1 for the horizontal coordinate and from 0 to the height of bitmap minus 1 for the vertical coordinate. It is also set by calling the SetBrushOrgEx function.

The SetBrushOrgEx function sets the brush origin that GDI assigns to the next brush an application selects into the specified device context.

Syntax

```
BOOL SetBrushOrgEx(
        HDC hdc,
        int nXOrg,
        int nYOrg,
        LPPOINT lppt
        );
```

Parameters

hdc - A handle to the device context.

nXOrg - The x-coordinate, in device units, of the new brush origin. If this value is greater than the brush width, its value is reduced using the modulus operator (nXOrgmod brush width).

nYOrg - The y-coordinate, in device units, of the new brush origin. If this value is greater than the brush height, its value is reduced using the modulus operator (nYOrgmod brush height).

lppt - A pointer to a POINT structure that receives the previous brush origin. This parameter can be NULL if the previous brush origin is not required.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- A brush is a bitmap that the system uses to paint the interiors of filled shapes.
- The brush origin is a pair of coordinates specifying the location of one pixel in the bitmap. The default brush origin coordinates are (0,0). For horizontal coordinates, the value 0 corresponds to the leftmost column of pixels; the width corresponds to the rightmost column. For vertical coordinates, the value 0 corresponds to the uppermost row of pixels; the height corresponds to the lowermost row.
- The system automatically tracks the origin of all window-managed device contexts and adjusts their brushes as necessary to maintain an alignment of patterns on the surface. The brush origin that is set with this call is relative to the upper-left corner of the client area.

  E.g.

  ```
  SetBrushOrgEx(hdc, 4, 0, NULL);
  ```

  It shifts the diagonal line that previously intersected the upper-left corner 4 pixels to the right so that the line no longer intersects the corner.

Drawing Modes

Windows uses the drawing mode attribute to determine how to copy the 'ink' from a pen onto the drawing surface of the device context. When you draw with a pen, the pixel values of the ink of the pen actually are combined with the pixel values of the display surface in a bitwise Boolean operation. The drawing mode specifies the particular Boolean operation to use.

The SetROP2 function sets the current foreground mix mode. GDI uses the foreground mix mode to combine pens and interiors of filled objects with the colors already on the screen. The foreground mix mode defines how colors from the brush or pen and the colors in the existing image are to be combined.

```
int SetROP2(
        HDC hdc,
        int fnDrawMode
       );
```

Parameters

hdc - A handle to the device context.

fnDrawMode - The mix mode. This parameter can be one of the following values.

Mix mode	Meaning
R2_BLACK	Pixel is always 0.
R2_COPYPEN	Pixel is the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the screen and the inverse of the pen.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the screen.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the screen.
R2_MERGENOTPEN	Pixel is a combination of the screen color and the inverse of the pen color.
R2_MERGEPEN	Pixel is a combination of the pen color and the screen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the screen color.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the screen color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN

	color.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.
R2_WHITE	Pixel is always 1.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the screen, but not in both.

Return Value

If the function succeeds, the return value specifies the previous mix mode. If the function fails, the return value is zero.

Remarks

Mix modes define how GDI combines source and destination colors when drawing with the current pen. The mix modes are binary raster operation codes, representing all possible Boolean functions of two variables, using the binary operations AND, OR, and XOR (exclusive OR), and the unary operation NOT. The mix mode is for raster devices only; it is not available for vector devices.

Retrieve the current drawing mode

The GetROP2 function retrieves the foreground mix mode of the specified device context. The mix mode specifies how the pen or interior color and the color already on the screen are combined to yield a new color.

Syntax

```
int GetROP2(
    HDC hdc
    );
```

Parameters

hdc - Handle to the device context.

Return Value

If the function succeeds, the return value specifies the foreground mix mode. If the function fails, the return value is zero.

The ROP stands for Raster Operation, the term used to describe a bitwise Boolean operations of the pixels of the raster display device. The 2 in ROP2 derives from the two operands: the pixels written by the pen and the pixels on the display surface.

The Mapping Mode

The mapping mode attribute of a device context affects nearly all drawing operations performed on the device context. Most (not all) GDI functions, interprets coordinate and sizes in terms of logical coordinates and logical units. Windows translates these logical coordinates into device coordinates and logical units into device units. This translation is controlled by current mapping mode, the window and viewport origins and the window and viewport extends.

To set a mapping mode, the SetMapMode function is called and to retrieve the current mapping mode for a DC the GetMapMode function is called.

The **SetMapMode** function sets the mapping mode of the specified device context. The mapping mode defines the unit of measure used to transform page-space units into device-space units, and also defines the orientation of the device's x and y axes.

```
int SetMapMode(
        HDC hdc,
        int fnMapMode
        );
```

Parameters

hdc - A handle to the device context.

fnMapMode – The new mapping mode. This parameter can be one of the following values.

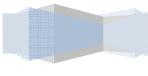
The mapping modes are described in the following table.

Mapping mode	Description
MM_ANISOTROPIC	Each unit in page space is mapped to an application-specified unit in device space. The axis may or may not be equally scaled (for example, a circle drawn in world space may appear to be an ellipse when depicted on a given device). The orientation of the

	axis is also specified by the application. x is not equal to y.
MM_HIENGLISH	Each unit in page space is mapped to **0.001 inch** in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_HIMETRIC	Each unit in page space is mapped to **0.01 millimeter** in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_ISOTROPIC	Each unit in page space is mapped to an application-defined unit in device space. The axes are always equally scaled. The orientation of the axes may be specified by the application. x is not equal to y.
MM_LOENGLISH	Each unit in page space is mapped to **0.01 inch** in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_LOMETRIC	Each unit in page space is mapped to **0.1 millimeter** in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_TEXT	Each unit in page space is mapped to **one pixel**; that is, no scaling is performed at all. When no translation is in effect (this is the default), page space in the MM_TEXT mapping mode is equivalent to physical device space. The value of x increases from left to right. The value of y increases from top to bottom.
MM_TWIPS	Each unit in page space is mapped to one twentieth of a printer's point **(1/1440 inch).** The value of x increases from left to right. The value of y increases from bottom to top.

Return Value

If the function succeeds, the return value identifies the previous mapping mode. If the function fails, the return value is zero.



Remarks

- The MM_TEXT mode allows applications to work in device pixels, whose size varies from device to device.
- The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS modes are useful for applications drawing in physically meaningful units (such as inches or millimeters).
- The MM_ISOTROPIC mode ensures a 1:1 aspect ratio.
- The MM_ANISOTROPIC mode allows the x-coordinates and y-coordinates to be adjusted independently.

There are 16 ROP2 codes. This number is derived from the 16 possible outcomes of combining a pen and a monochrome display surface. The pen can be either black or white. Likewise, the display surface pixel can be either black or white. There are four combinations of the pen and the display surface: a white pen on a white surface, a white pen in black surface, a black pen on a white surface and a black pen in black surface. The default drawing mode for a device context is the R2_COPYPEN mode.

Drawing Mode	**Selected pen (P)**	**1**	**2**	**0**	**0**	**Decimal Result**	**Boolean Operations**
	Destination bitmap (D)	**1**	**0**	**1**	**0**		
R2_BLACK		**0**	**0**	**0**	**0**	**0**	**0**
R2_NOTMERGEPEN		**0**	**0**	**0**	**1**	**1**	**~(P | D)**
R2_MASKNOTPEN		**0**	**0**	**1**	**0**	**2**	**~P & D**
R2_NOTCOPYPEN		**0**	**0**	**1**	**1**	**3**	**~P**
R2_MASKPENNOT		**0**	**1**	**0**	**0**	**4**	**P & ~D**
R2_NOT		**0**	**1**	**0**	**1**	**5**	**~D**
R2_XORPEN		**0**	**1**	**1**	**0**	**6**	**P^D**
R2_NOTMASKPEN		**0**	**1**	**1**	**1**	**7**	**~ (P & D)**
R2_MASKPEN		**1**	**0**	**0**	**0**	**8**	**P & D**
R2_NOTXORPEN		**1**	**0**	**0**	**1**	**9**	**~ (P ^ D)**
R2_NOPR2_NOP		**1**	**0**	**1**	**0**	**10**	**D**
R2_MERGENOTPEN		**1**	**0**	**1**	**1**	**11**	**~P | D**
R2_COPYPEN		**1**	**1**	**0**	**0**	**12**	**P (Default)**

R2_MERGEPENNOT	**1**	**1**	**0**	**1**	**13**	**P | ~D**
R2_MERGEPEN	**1**	**1**	**1**	**0**	**14**	**P | D**
R2_WHITE	**1**	**1**	**1**	**1**	**15**	**1**

A similar procedure handles color pens writing on a color surface. Rather than use ones and zeros in monochrome system, GDI user RGB values to represent the colors of the pen and the destination. An RGB color value is long integer containing Red, Green, and Blue color fields, each 1 byte wide, representing intensity for 0 to 255. The bitwise Boolean operation is performed on the two RGB color values.

E.g.

When a red pen is writing on a white destination surface using R2_NOTMASKEDPEN operation, this operation sets the destination to the result of negating the logical AND of the pen color and original destination color. The pen is red (0xFF0000), the destination is white (0xFFFFFF) and the result is 0x0000FF.

Except MM_ANISOTROPHIC and MM_ISOTROPIC mapping modes, all other mapping modes have been constrained mapping modes.

The MM_ANISOTROPHIC mode is an unconstrained mapping mode, and the MM_ISOTROPHIC mode is partially constrained mapping mode. Both the mapping modes allow specifying the scaling factor for each axis. When MM_ISOTROPIC mapping mode is used, windows adjusts one of the scaling factors so that one logical unit maps to an identical distance on both axes. It is partially constraints because, although you can specify the scaling factors for both axes, windows will change one of them so that logical unit horizontally covers the same physical distance as logical unit vertically.

The MM_ANISOTROPIC mapping mode allows specifying separate scaling factors for each axis. When MM_ANISOTROPHIC mapping mode is used one logical unit along the horizontal axis doesn't necessarily travel the same physical distance as one logical unit along the vertical axis.

SetWindowExtEx

The SetWindowExtEx function sets the horizontal and vertical extents of the window for a device context by using the specified values.

```
BOOL SetWindowExtEx(
        HDC hdc,
        int nXExtent,
        int nYExtent,
        LPSIZE lpSize
);
```

Parameters

hdc - A handle to the device context.

nXExtent - The window's horizontal extent in logical units.

nYExtent - The window's vertical extent in logical units.

lpSize - A pointer to a SIZE structure that receives the previous window extents, in logical units. If lpSize is NULL, this parameter is not used.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

SIZE Structure

The SIZE structure specifies the width and height of a rectangle.

```
typedef struct tagSIZE {
 LONG cx;
 LONG cy;
} SIZE, *PSIZE;
```

Members

cx - Specifies the rectangle's width. The units depend on which function uses this.

cy - Specifies the rectangle's height. The units depend on which function uses this.

Remarks

The rectangle dimensions stored in this structure may correspond to viewport extents, window extents, text extents, bitmap dimensions, or the aspect-ratio filter for some extended functions.

SetViewportExtEx

The SetViewportExtEx function sets the horizontal and vertical extents of the viewport for a device context by using the specified values.

```
BOOL SetViewportExtEx(
        HDC hdc,
        int nXExtent,
        int nYExtent,
        LPSIZE lpSize
);
```

Parameters

hdc - A handle to the device context.

nXExtent - The horizontal extent, in device units, of the viewport.

nYExtent - The vertical extent, in device units, of the viewport.

lpSize - A pointer to a SIZE structure that receives the previous viewport extents, in device units. If lpSize is NULL, this parameter is not used.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks (For BOTH)

The window refers to the logical coordinate system of the page space. The extent is the maximum value of an axis. This function sets the maximum values for the horizontal and vertical axes of the window (in logical coordinates).

When the following mapping modes are set, calls to the SetWindowExtEx and SetViewportExtEx functions are ignored:

- MM_HIENGLISH

- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

SetWindowOrgEx

The SetWindowOrgEx function specifies which window point maps to the viewport origin (0,0).

```
BOOL SetWindowOrgEx(
        HDC hdc,
        int X,
        int Y,
        LPPOINT lpPoint
);
```

Parameters

hdc - A handle to the device context.

X - The x-coordinate, in logical units, of the new window origin.

Y - The y-coordinate, in logical units, of the new window origin.

lpPoint - A pointer to a POINT structure that receives the previous origin of the window, in logical units. If lpPoint is NULL, this parameter is not used.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

SetViewportOrgEx

The SetViewportOrgEx function specifies which device point maps to the window origin (0,0).

Syntax

```
BOOL SetViewportOrgEx(
        HDC hdc,
        int X,
        int Y,
        LPPOINT lpPoint
        );
```

Parameters

hdc - A handle to the device context.

X - The x-coordinate, in device units, of the new viewport origin.

Y - The y-coordinate, in device units, of the new viewport origin.

lpPoint - A pointer to a POINT structure that receives the previous viewport origin, in device coordinates. If lpPoint is NULL, this parameter is not used.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

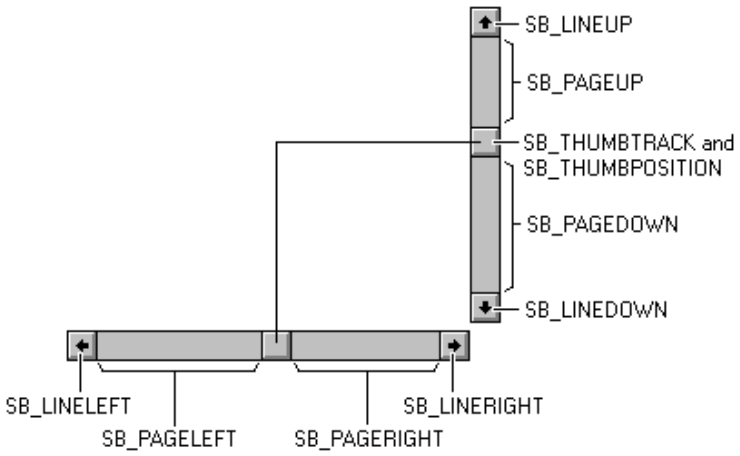
Remarks (for BOTH)

This function (along with SetViewportExtEx and SetWindowExtEx) helps define the mapping from the logical coordinate space (also known as a window) to the device coordinate space (the viewport). SetViewportOrgEx specifies which device point maps to the logical point (0,0). It has the effect of shifting the axes so that the logical point (0,0) no longer refers to the upper-left corner.

```
//map the logical point (0,0) to the device point (xViewOrg, yViewOrg)
SetViewportOrgEx ( hdc, xViewOrg, yViewOrg, NULL)
```

Introduction to Scroll Bar (Parts of a Scroll Bar)

A scroll bar is an object that allows the user to navigate either left and right or up and down, either on a document or on a section of the window. A scroll bar appears as a long bar with a (small) button at each end. Between these buttons, there is a moveable bar called a thumb. To scroll, the user can click one of the buttons or grab the thumb and drag it:



Types of Scroll Bars

There are two types of scroll bars: vertical or horizontal. A vertical scroll bar allows the user to navigate up and down on a document or a section of a window. A horizontal scroll bar allows the user to navigate left and right on a document or a section of a window. As far as Microsoft Windows is concerned, there are two categories of scroll bars: automatic and control-based.

Automatic Scroll Bars

Some controls need a scroll bar to efficiently implement their functionality. The primary example is the edit control, which is used to display text. On that control, when the text is too long, the user needs to be able to scroll down and up to access the document fully. In the same way, if the text is too wide, the user needs to be able to scroll left and right to view the whole document.

When creating a text-based document or window, you can easily ask that one or both scroll bars be added. Of course, an edit control must be able to handle multiple lines of text. This is taken care of by adding the ES_MULTILINE flag to its styles. Then:

- To add a vertical scroll bar to the window, add the WS_VSCROLL flag to the Style argument of the CreateWindow() or the CreateWindowEx() function.
- To add a horizontal scroll bar to the window, add the WS_HSCROLL flag to the Style argument of the CreateWindow() or the CreateWindowEx() function.
- To make the vertical scroll bar appear when necessary, that is, when the document is too long, add the ES_AUTOVSCROLL style
- To make the horizontal scroll bar appear as soon as at least one line of the document is too wide, add the ES_AUTOVSCROLL style

Of course, you can use only one, two, three or all four styles.

Control-Based Scroll Bars

Microsoft Windows provides another type of scroll bar. Treated as its own control, a scroll bar is created like any other window and can be positioned anywhere on its host.

To create a scroll bar as a Windows control, call the CreateWindow() or the CreateWindowEx() functions and specify the class name as SCROLLBAR.

Scroll Bar Creation

The easiest way to add a scroll bar to a project is through the resource script of the project. The syntax to follow is:

SCROLLBAR id, x, y, width, height [[, style [[, ExtendedStyle]]]]

This declaration starts with the SCROLLBAR keyword as the name of the class that creates a scroll bar.

- The id is the identification of the control
- The x parameter is the Left parameter of the control
- The y parameter is the Top parameter of the control
- The width parameter is the distance from the left to the right border of the control
- The height parameter is the distance from the top to the bottom border of the control

Here is an example:

```
SCROLLBAR IDC_SCROLLBAR1,10,45,215,11
```

Alternatively, to create a scroll bar, you can use either the CreateWindow() or the CreateWindowEx() functions, specifying the class name as SCROLLBAR.

Sub-classing Window Class

When an application creates a window, the operating 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 Windows 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.

Sub-classing is a technique that allows an application to capture and process messages sent or posted to a particular window before the window has a chance to process them. By sub-classing a window, an application can increase, modify, or monitor the behavior of the window.

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.

All GUI applications on Windows are driven by getting messages from the operating system, and translating them into events for the user to handle. So when the mouse moves, a window is given a message telling it that the mouse has moved, and here are its current coordinates. Or when the user presses a key, a different message is generated and a window is alerted. This message dispatching is how all events are fired -- the system lets the framework know something has happened, and the framework passes the information along to you in the form of an event.

It is traditionally called a WndProc (pronounced 'wind-proc') and it's short for window procedure. You let the system know what WndProc to call by specifying one in a window's class definition (called a WNDCLASS) structure, which is then registered with the system. Every window created with that class definition will have its WndProc called by the system for all messages. The OS knows when to call the WndProc because there's a loop in the application known as the message pump which handles all the message passing.

```
while (!done) {
        MSG msg;
        if (GetMessage( &msg, NULL, 0, 0 )) {
                if (msg.message == WM_QUIT)       done = true;

                TranslateMessage( &msg );
                DispatchMessage( &msg );
        }
}
```

Chapter 04: Graphical Output

There are three types of graphical figures that can be produced by windows program. They are from simplest to the most complex: Points (Pixels), Lines & Bounded areas. A series of lines is used to create polygons. A windows program is most efficient when use most appropriate drawing command.

Getting & Setting the Color of a Pixel

The GetPixel function retrieves the red, green, blue (RGB) color value of the pixel at the specified coordinates.

```
COLORREF GetPixel(
        HDC hdc,        //Handle of DC
        int xPos,       //x Coordinate, in logical units, of the pixel to be examined.
        Int yPos        //y Coordinate, in logical units, of the pixel to be examined.
        );
```

Return Value

The return value is the RGB value of the pixel. If the pixel is outside of the current clipping region, the return value is CLR_INVALID.

Remarks

- The pixel must be within the boundaries of the current clipping region. Not all devices support GetPixel. An application should call GetDeviceCaps to determine whether a specified device supports this function.
- A bitmap must be selected within the device context, otherwise, CLR_INVALID is returned on all pixels.

The SetPixel function sets the pixel at the specified coordinates to the specified color.

```
COLORREF SetPixel(
        HDC hdc,            //Handle of DC
        int xPos,           //x Coordinate, in logical units, of the point to be set.
        Int yPos,           //y coordinate, in logical units, of the point to be set.
        COLORREF crColor    //pixel color (RGB)
        );
```

Return Value

If the function **succeeds**, the return value is the RGB value that the function sets the pixel to. This value may differ from the color specified by crColor; that occurs when an exact match for the specified color cannot be found. ***On Failure this r****eturns -1. To get more information you can call* ***GetLastError****.*

Remarks

The function fails if the pixel coordinates lie outside of the current clipping region. Not all devices support the SetPixel function.

Drawing Lines

- **LineTo**

The **LineTo** function draws a line from the current position up to, but not including, the specified point.

```
BOOL LineTo(
        HDC hdc,
        Int xEnd,
        Int yEnd
        );
```

Returns

On Success: Returns NonZero & On Failure: Returns Zero.

Remarks

The line is drawn by using the current pen and, if the pen is a geometric pen, the current brush. If LineTo succeeds, the current position is set to the specified ending point.

- **MoveToEx**

The MoveToEx function updates the current position to the specified point and optionally returns the previous position.

```
BOOL MoveToEx(
        HDC hdc,
        int X,              // New x Position
        int Y,              // New y Position
        LPPOINT lpPoint     // Pointer to the POINT Structure to store current pen position. If
                            NULL Previous Position not returned.
        );
```

POINT Structure

```
struct POINT
{
        LONG x;             // x-coordinate of the point
        LONG y;             // y-coordinate of the point
};
```

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The MoveToEx function affects all drawing functions.

E.g.

```
void Marker(LONG x, LONG y, HWND hwnd)
{
	HDC hdc;
	hdc = GetDC(hwnd);
	MoveToEx(hdc, (int) x - 10, (int) y, (LPPOINT) NULL);
	LineTo(hdc, (int) x + 10, (int) y);
	MoveToEx(hdc, (int) x, (int) y - 10, (LPPOINT) NULL);
	LineTo(hdc, (int) x, (int) y + 10);
 ReleaseDC(hwnd, hdc);
}
```

- **GetCurrentPositionEx**

The GetCurrentPositionEx function retrieves the current position in logical coordinates.

```
BOOL GetCurrentPositionEx(
	HDC hdc,
	LPPOINT lpPoint		// Address of the structure receiving current position
	);
```

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

- **Polyline**

The Polyline function draws a series of line segments by connecting the points in the specified array.

```
BOOL Polyline(
	HDC hdc,
	const POINT *lppt,
	int cPoints
	);
```

Parameters

hdc - A handle to a device context.

lppt - A pointer to an array of POINT structures, in logical units.

cPoints - The number of points in the array. This number must be greater than or equal to two.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The lines are drawn from the first point through subsequent points by using the current pen. Unlike the LineTo or PolylineTo functions, the Polyline function neither uses nor updates the current position.

- **PolylineTo**

The PolylineTo function draws one or more straight lines.

```
BOOL PolylineTo(
        HDC hdc,
        const POINT *lppt,
        DWORD cCount
        );
```

Parameters

hdc - A handle to the device context.

lppt - A pointer to an array of POINT structures that contains the vertices of the line, in logical units.

cCount - The number of points in the array.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- Unlike the Polyline function, the PolylineTo function uses and updates the current position.
- A line is drawn from the current position to the first point specified by the lppt parameter by using the current pen. For each additional line, the function draws from the ending point of the previous line to the next point specified by lppt.
- PolylineTo moves the current position to the ending point of the last line.
- If the line segments drawn by this function form a closed figure, the figure is not filled.

PolyPolyline

The PolyPolyline function draws multiple series of connected line segments.

```
BOOL PolyPolyline(
        HDC hdc,
        const POINT *lppt,
        const DWORD *lpdwPolyPoints,
        DWORD cCount
);
```

Parameters

hdc - A handle to the device context.

lppt - A pointer to an array of POINT structures that contains the vertices of the polylines, in logical units. The polylines are specified consecutively.

lpdwPolyPoints - A pointer to an array of variables specifying the number of points in the lppt array for the corresponding polyline. Each entry must be greater than or equal to two.

cCount - The total number of entries in the lpdwPolyPoints array.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The line segments are drawn by using the current pen. The figures formed by the segments are not filled. The current position is neither used nor updated by this function.

- **Arc**

The Arc function draws an elliptical arc.

```
BOOL Arc(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect,
```

```
        int nXStartArc,
        int nYStartArc,
        int nXEndArc,
        int nYEndArc
);
```

Parameters

hdc - A handle to the device context where drawing takes place.

nLeftRect - The x-coordinate, in logical units, of the upper-left corner of the bounding rectangle.

nTopRect - The y-coordinate, in logical units, of the upper-left corner of the bounding rectangle.

nRightRect - The x-coordinate, in logical units, of the lower-right corner of the bounding rectangle.

nBottomRect - The y-coordinate, in logical units, of the lower-right corner of the bounding rectangle.

nXStartArc - The x-coordinate, in logical units, of the ending point of the radial line defining the starting point of the arc.

nYStartArc - The y-coordinate, in logical units, of the ending point of the radial line defining the starting point of the arc.

nXEndArc - The x-coordinate, in logical units, of the ending point of the radial line defining the ending point of the arc.

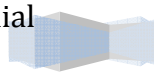
nYEndArc - The y-coordinate, in logical units, of the ending point of the radial line defining the ending point of the arc.

Return Value

If the arc is drawn, the return value is nonzero. If the arc is not drawn, the return value is zero.

Remarks

The points (nLeftRect, nTopRect) and (nRightRect, nBottomRect) specify the bounding rectangle. An ellipse formed by the specified bounding rectangle defines the curve of the arc. The arc extends in the current drawing direction from the point where it intersects the radial



from the center of the bounding rectangle to the (nXStartArc, nYStartArc) point. The arc ends where it intersects the radial from the center of the bounding rectangle to the (nXEndArc, nYEndArc) point. If the starting point and ending point are the same, a complete ellipse is drawn.

The arc is drawn using the current pen; it is not filled. The current position is neither used nor updated by Arc.

- **ArcTo**

The ArcTo function draws an elliptical arc.

```
BOOL ArcTo(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect,
        int nXRadial1,
        int nYRadial1,
        int nXRadial2,
        int nYRadial2
);
```

Parameters

hdc - A handle to the device context where drawing takes place.

nLeftRect - The x-coordinate, in logical units, of the upper-left corner of the bounding rectangle.

nTopRect - The y-coordinate, in logical units, of the upper-left corner of the bounding rectangle.

nRightRect - The x-coordinate, in logical units, of the lower-right corner of the bounding rectangle.

nBottomRect - The y-coordinate, in logical units, of the lower-right corner of the bounding rectangle.

nXRadial1 - The x-coordinate, in logical units, of the endpoint of the radial defining the starting point of the arc.
nYRadial1 - The y-coordinate, in logical units, of the endpoint of the radial defining the starting point of the arc.
nXRadial2 - The x-coordinate, in logical units, of the endpoint of the radial defining the ending point of the arc.
nYRadial2 - The y-coordinate, in logical units, of the endpoint of the radial defining the ending point of the arc.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

ArcTo is similar to the Arc function, except that the current position is updated. The points (nLeftRect, nTopRect) and (nRightRect, nBottomRect) specify the bounding rectangle. An ellipse formed by the specified bounding rectangle defines the curve of the arc. The arc extends counterclockwise from the point where it intersects the radial line from the center of the bounding rectangle to the (nXRadial1, nYRadial1) point. The arc ends where it intersects the radial line from the center of the bounding rectangle to the (nXRadial2, nYRadial2) point. If the starting point and ending point are the same, a complete ellipse is drawn.

A line is drawn from the current position to the starting point of the arc. If no error occurs, the current position is set to the ending point of the arc.

The arc is drawn using the current pen; it is not filled.

- **PolyBezier**

```
BOOL PolyBezier(
        HDC hdc,
        const POINT *lppt,
        DWORD cPoints
);
```

Parameters

hdc - A handle to a device context.

lppt - A pointer to an array of POINT structures that contain the endpoints and control points of the curve(s), in logical units.

cPoints - The number of points in the lppt array. This value must be one more than three times the number of curves to be drawn, because each Bèzier curve requires two control points and an endpoint, and the initial curve requires an additional starting point.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The PolyBezier function draws cubic Bèzier curves by using the endpoints and control points specified by the lppt parameter. The first curve is drawn from the first point to the fourth point by using the second and third points as control points. Each subsequent curve in the sequence needs exactly three more points: the ending point of the previous curve is used as the starting point, the next two points in the sequence are control points, and the third is the ending point.

The current position is neither used nor updated by the PolyBezier function. The figure is not filled. This function draws lines by using the current pen.

- **PolyBezierTo**

The PolyBezierTo function draws one or more Bèzier curves.

```
BOOL PolyBezierTo(
        HDC hdc,
        const POINT *lppt,
        DWORD cCount
);
```

Parameters

hdc - A handle to a device context.

lppt - A pointer to an array of POINT structures that contains the endpoints and control points, in logical units.

cCount - The number of points in the lppt array. This value must be three times the number of curves to be drawn, because each Bèzier curve requires two control points and an ending point.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

This function draws cubic Bèzier curves by using the control points specified by the lppt parameter. The first curve is drawn from the current position to the third point by using the first two points as control points. For each subsequent curve, the function needs exactly three more points, and uses the ending point of the previous curve as the starting point for the next.

PolyBezierTo moves the current position to the ending point of the last Bèzier curve. The figure is not filled. This function draws lines by using the current pen.

Drawing Filled Areas

- **Rectangle**

The Rectangle function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush.

```
BOOL Rectangle(
    HDC hdc,
    Int xUpperLeft, // The x-coordinate, in logical coordinates, of the upper-left corner of the rectangle.
    int uUpperLeft, // The y-coordinate, in logical coordinates, of the upper-left corner of the rectangle.
    int xLowerRight, // The x-coordinate, in logical coordinates, of the lower-right corner of the rectangle.
    int xLowerLeft //The y-coordinate, in logical coordinates, of the lower-right corner of the rectangle.
    );
```

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The current position is neither used nor updated by Rectangle. The rectangle that is drawn excludes the bottom and right edges.
- If a PS_NULL pen is used, the dimensions of the rectangle are 1 pixel less in height and 1 pixel less in width.

- **Polygon**

The Polygon function draws a polygon consisting of two or more vertices connected by straight lines. The polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode.

```
BOOL Polygon(
        HDC hdc,
        const POINT *lpPoints,
        int nCount
        );
```

Parameters

hdc - A handle to the device context.

lpPoints - A pointer to an array of POINT structures that specify the vertices of the polygon, in logical coordinates.

nCount - The number of vertices in the array. This value must be greater than or equal to 2.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The polygon is closed automatically by drawing a line from the last vertex to the first.
- The current position is neither used nor updated by the Polygon function.
- Any extra points are ignored. To draw a line with more points, divide your data into groups, each of which have less than the maximum number of points, and call the function for each group of points. Remember to connect the line segments.

- **PolyPolygon**

The PolyPolygon function draws a series of closed polygons. Each polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode. The polygons drawn by this function can overlap.

```
BOOL PolyPolygon(
        HDC hdc,
        const POINT *lpPoints,
        const INT *lpPolyCounts,
        int nCount
        );
```

Parameters

hdc - A handle to the device context.

lpPoints - A pointer to an array of POINT structures that define the vertices of the polygons, in logical coordinates. The polygons are specified consecutively. Each polygon is closed automatically by drawing a line from the last vertex to the first. Each vertex should be specified once.

lpPolyCounts - A pointer to an array of integers, each of which specifies the number of points in the corresponding polygon. Each integer must be greater than or equal to 2.

nCount - The total number of polygons.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The current position is neither used nor updated by this function.
- Any extra points are ignored. To draw the polygons with more points, divide your data into groups, each of which have less than the maximum number of points, and call the function for each group of points. Note, it is best to have a polygon in only one of the groups.

- **Ellipse**

The Ellipse function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle. The ellipse is outlined by using the current pen and is filled by using the current brush.

```
BOOL Ellipse(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect
        );
```

Parameters

hdc - A handle to the device context.

nLeftRect - The x-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nTopRect - The y-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nRightRect - The x-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

nBottomRect - The y-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The current position is neither used nor updated by Ellipse.

- **Chord**

The Chord function draws a chord (a region bounded by the intersection of an ellipse and a line segment, called a secant). The chord is outlined by using the current pen and filled by using the current brush.

```
BOOL Chord(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
```

```
    int nBottomRect,
    int nXRadial1,
    int nYRadial1,
    int nXRadial2,
    int nYRadial2
);
```

Parameters

hdc - A handle to the device context in which the chord appears.

nLeftRect - The x-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nTopRect - The y-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nRightRect - The x-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

nBottomRect - The y-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

nXRadial1 - The x-coordinate, in logical coordinates, of the endpoint of the radial defining the beginning of the chord.

nYRadial1 - The y-coordinate, in logical coordinates, of the endpoint of the radial defining the beginning of the chord.

nXRadial2 - The x-coordinate, in logical coordinates, of the endpoint of the radial defining the end of the chord.

nYRadial2 - The y-coordinate, in logical coordinates, of the endpoint of the radial defining the end of the chord.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The curve of the chord is defined by an ellipse that fits the specified bounding rectangle. The curve begins at the point where the ellipse intersects the first radial and extends counterclockwise to the point where the ellipse intersects

the second radial. The chord is closed by drawing a line from the intersection of the first radial and the curve to the intersection of the second radial and the curve.

- If the starting point and ending point of the curve are the same, a complete ellipse is drawn.
- The current position is neither used nor updated by Chord.

- **Pie**

The Pie function draws a pie-shaped wedge bounded by the intersection of an ellipse and two radials. The pie is outlined by using the current pen and filled by using the current brush.

```
BOOL Pie(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect,
        int nXRadial1,
        int nYRadial1,
        int nXRadial2,
        int nYRadial2
);
```

Parameters

hdc - A handle to the device context.

nLeftRect - The x-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nTopRect - The y-coordinate, in logical coordinates, of the upper-left corner of the bounding rectangle.

nRightRect - The x-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

nBottomRect - The y-coordinate, in logical coordinates, of the lower-right corner of the bounding rectangle.

nXRadial1 - The x-coordinate, in logical coordinates, of the endpoint of the first radial.

nYRadial1 - The y-coordinate, in logical coordinates, of the endpoint of the first radial.

nXRadial2 - The x-coordinate, in logical coordinates, of the endpoint of the second radial.

nYRadial2 - The y-coordinate, in logical coordinates, of the endpoint of the second radial.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The curve of the pie is defined by an ellipse that fits the specified bounding rectangle. The curve begins at the point where the ellipse intersects the first radial and extends counterclockwise to the point where the ellipse intersects the second radial.
- The current position is neither used nor updated by the Pie function.

- **RoundRect**

The RoundRect function draws a rectangle with rounded corners. The rectangle is outlined by using the current pen and filled by using the current brush.

```
BOOL RoundRect(
        HDC hdc,
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect,
        int nWidth,
        int nHeight
        );
```

Parameters

hdc - A handle to the device context.

nLeftRect - The x-coordinate, in logical coordinates, of the upper-left corner of the rectangle.

nTopRect - The y-coordinate, in logical coordinates, of the upper-left corner of the rectangle.

nRightRect - The x-coordinate, in logical coordinates, of the lower-right corner of the rectangle.

nBottomRect - The y-coordinate, in logical coordinates, of the lower-right corner of the rectangle.

nWidth - The width, in logical coordinates, of the ellipse used to draw the rounded corners.

nHeight - The height, in logical coordinates, of the ellipse used to draw the rounded corners.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The current position is neither used nor updated by this function.

- **FillRect**

The FillRect function fills a rectangle by using the specified brush. This function includes the left and top borders, but excludes the right and bottom borders of the rectangle.

```
int FillRect(
        HDC hDC,
        const RECT *lprc,
        HBRUSH hbr
);
```

Parameters

hDC - A handle to the device context.

lprc - A pointer to a RECT structure that contains the logical coordinates of the rectangle to be filled.

hbr - A handle to the brush used to fill the rectangle.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The brush identified by the hbr parameter may be either a handle to a logical brush or a color value. If specifying a handle to a logical brush, call one of the following functions to obtain the handle: CreateHatchBrush, CreatePatternBrush, or CreateSolidBrush. Additionally, you may retrieve a handle to one of the stock brushes by using the GetStockObject function. If specifying a color value for the hbr parameter, it must be one of the standard system colors (the value 1 must be added to the chosen color).

```
FillRect(hdc, &rect, (HBRUSH) (COLOR_WINDOW+1));
```

[Note: For a list of all the standard system colors, see GetSysColor.]

When filling the specified rectangle, FillRect does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but not including, the right column and bottom row, regardless of the current mapping mode.

- **FrameRect**

The FrameRect function draws a border around the specified rectangle by using the specified brush. The width and height of the border are always one logical unit.

```
int FrameRect(
    HDC hDC,
    const RECT *lprc,
    HBRUSH hbr
);
```

Parameters

hDC - A handle to the device context in which the border is drawn.

lprc - A pointer to a RECT structure that contains the logical coordinates of the upper-left and lower-right corners of the rectangle.

hbr - A handle to the brush used to draw the border.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- The brush identified by the hbr parameter must have been created by using the CreateHatchBrush, CreatePatternBrush, or CreateSolidBrush function, or retrieved by using the GetStockObject function.
- If the bottom member of the RECT structure is less than or equal to the top member, or if the right member is less than or equal to the left member, the function does not draw the rectangle.

- **InvertRect**

The InvertRect function inverts a rectangle in a window by performing a logical NOT operation on the color values for each pixel in the rectangle's interior.

```
BOOL InvertRect(
    HDC hDC,
    const RECT *lprc
);
```

Parameters

hDC - A handle to the device context.

lprc - A pointer to a RECT structure that contains the logical coordinates of the rectangle to be inverted.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

- On monochrome screens, InvertRect makes white pixels black and black pixels white. On color screens, the inversion depends on how colors are generated for the screen. Calling InvertRect twice for the same rectangle restores the display to its previous colors.

- **CreatePolygonRgn**

The CreatePolygonRgn function creates a polygonal region.

```
HRGN CreatePolygonRgn(
        const POINT *lppt,
        int cPoints,
        int fnPolyFillMode
);
```

Parameters

lppt - A pointer to an array of POINT structures that define the vertices of the polygon in logical units. The polygon is presumed closed. Each vertex can be specified only once.

cPoints - The number of points in the array.

fnPolyFillMode - The fill mode used to determine which pixels are in the region. This parameter can be one of the following values.

Value	Meaning
ALTERNATE	Selects alternate mode (fills area between odd-numbered and even-numbered polygon sides on each scan line).
WINDING	Selects winding mode (fills any region with a nonzero winding value).

Return Value

If the function succeeds, the return value is the handle to the region. If the function fails, the return value is NULL.

Remarks

When you no longer need the HRGN object, call the DeleteObject function to delete it.

- **CreateRectRgn**

The CreateRectRgn function creates a rectangular region.

```
HRGN CreateRectRgn(
        int nLeftRect,
        int nTopRect,
        int nRightRect,
         int nBottomRect
);
```

Parameters

nLeftRect - Specifies the x-coordinate of the upper-left corner of the region in logical units.

nTopRect - Specifies the y-coordinate of the upper-left corner of the region in logical units.

nRightRect - Specifies the x-coordinate of the lower-right corner of the region in logical units.

nBottomRect - Specifies the y-coordinate of the lower-right corner of the region in logical units.

Return Value

If the function succeeds, the return value is the handle to the region. If the function fails, the return value is NULL.

Remarks

When you no longer need the HRGN object, call the DeleteObject function to delete it.

- **CreateEllipticRgn**

The CreateEllipticRgn function creates an elliptical region.

```
HRGN CreateEllipticRgn(
        int nLeftRect,
        int nTopRect,
        int nRightRect,
        int nBottomRect
);
```

Parameters

nLeftRect - Specifies the x-coordinate in logical units, of the upper-left corner of the bounding rectangle of the ellipse.

nTopRect - Specifies the y-coordinate in logical units, of the upper-left corner of the bounding rectangle of the ellipse.

nRightRect - Specifies the x-coordinate in logical units, of the lower-right corner of the bounding rectangle of the ellipse.

nBottomRect - Specifies the y-coordinate in logical units, of the lower-right corner of the bounding rectangle of the ellipse.

Return Value

If the function succeeds, the return value is the handle to the region. If the function fails, the return value is NULL.

Remarks

- When you no longer need the HRGN object, call the DeleteObject function to delete it.
- A bounding rectangle defines the size, shape, and orientation of the region: The long sides of the rectangle define the length of the ellipse's major axis; the short sides define the length of the ellipse's minor axis; and the center of the rectangle defines the intersection of the major and minor axes.

- **FillRgn**

The FillRgn function fills a region by using the specified brush.

```
BOOL FillRgn(
        HDC hdc,
        HRGN hrgn,
        HBRUSH hbr
        );
```

Parameters

hdc - Handle to the device context.

hrgn - Handle to the region to be filled. The region's coordinates are presumed to be in logical units.

hbr - Handle to the brush to be used to fill the region.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

- **PaintRgn**

The PaintRgn function paints the specified region by using the brush currently selected into the device context.

```
BOOL PaintRgn(
        HDC hdc,
        HRGN hrgn
        );
```

Parameters

hdc - Handle to the device context.

hrgn - Handle to the region to be filled. The region's coordinates are presumed to be logical coordinates.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Chapter 05: Keyboard, Mouse, Timer

Windows sends input to a program's windows function as messages gets stored in the hardware input queue.

We will study three types of input mechanisms here, Keyboard, Mouse and Timer.

Keyboard Input

The system provides device-independent keyboard support for applications by installing a keyboard device driver appropriate for the current keyboard. The system provides language-independent keyboard support by using the language-specific keyboard layout currently selected by the user or the application. The keyboard device driver receives scan codes from the keyboard, which are sent to the keyboard layout where they are translated into messages and posted to the appropriate windows in your application.

Assigned to each key on a keyboard is a unique value called a scan code, a device-dependent identifier for the key on the keyboard. A keyboard generates two scan codes when the user types a key—one when the user presses the key and another when the user releases the key.

The keyboard device driver interprets a scan code and translates (maps) it to a virtual-key code, a device-independent value defined by the system that identifies the purpose of a key. After translating a scan code, the keyboard layout creates a message that includes the scan code, the virtual-key code, and other information about the keystroke, and then places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Eventually, the thread's message loop removes the message and passes it to the appropriate window procedure for processing. The following figure illustrates the keyboard input model.

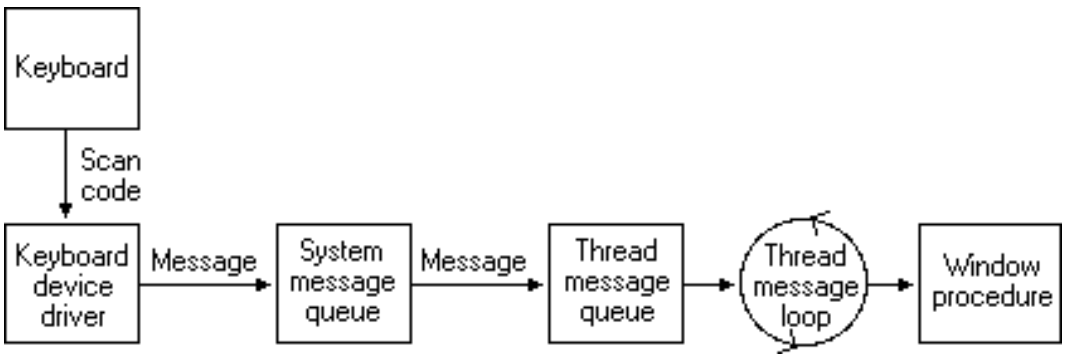


Figure: Keyboard input processing model

Windows keyboard input messages

Application Keyboard Messages (Non-System)	
WM_KEYDOWN	Message reporting a key down transition. Posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is not pressed.
WM_KEYUP	Posted to the window with the keyboard focus when a nonsystem key is released. A nonsystem key is a key that is pressed when the ALT key is not pressed, or a keyboard key that is pressed when a window has the keyboard focus.
WM_DEADCHAR	Posted to the window with the keyboard focus when a WM_KEYUP message is translated by the TranslateMessage function. WM_DEADCHAR specifies a character code generated by a dead key. A dead key is a key that generates a character, such as the umlaut (double-dot), which is combined with another character to form a composite character. For example, the umlaut-O character (Ö) is generated by typing the dead key for the umlaut character, and then typing the O key.
WM_CHAR	Posted to the window with the keyboard focus when a WM_KEYDOWN message is translated by the TranslateMessage function. The WM_CHAR message contains the character code of the key that was pressed.
System Keyboard Messages	
WM_SYSKEYDOWN	Posted to the window with the keyboard focus when the user presses the F10 key (which activates the menu bar) or holds down the ALT key and then presses another key. It also occurs when no window currently has the keyboard focus; in this case, the WM_SYSKEYDOWN message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the lParam parameter.
WM_SYSKEYUP	Posted to the window with the keyboard focus when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the keyboard focus; in this case, the WM_SYSKEYUP message is

	sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the lParam parameter. A window receives this message through its WindowProc function.
WM_SYSDEADCHAR	Sent to the window with the keyboard focus when a WM_SYSKEYDOWN message is translated by the TranslateMessage function. WM_SYSDEADCHAR specifies the character code of a system dead key — that is, a dead key that is pressed while holding down the ALT key.
WM_SYSCHAR	Posted to the window with the keyboard focus when a WM_SYSKEYDOWN message is translated by the TranslateMessage function. It specifies the character code of a system character key — that is, a character key that is pressed while the ALT key is down.
Keyboard focus messages	
WM_SETFOCUS	Sent to a window after it has gained the keyboard focus.
WM_KILLFOCUS	Sent to a window immediately before it loses the keyboard focus.

Keyboard Input originates when the user presses or releases a key on the keyboard. The keyboard device driver via windows converts the event into a message that places into the system queue. When application with the input-focus attempts to retrieve a message from its application queue and the queue is empty, windows looks for a message for that application in the system queue. It then transfers keyboard messages from the system queue to the application's thread queue. Keyboard messages aren't placed directly into the thread queue because an earlier message in the queue might cause the input focus to move to a different thread.

Many times keyboard message won't need to be processed in order to react to keyboard input. Windows translates some keyboard input to other types of messages. For example, the user can use the keyboard to select an item from a menu in the application. Windows then processes the keyboard messages and generates a message indicating that a menu selection was made. This process enables to respond to the menu selection message regardless of whether the selection was made via the keyboard or the mouse.

For keyboard two messages matter most: the WM_CHAR message tells about visible character (numbers, letters, punctuations and so on.) and WM_KEYDOWN tell about non-visible keys (function keys and arrow keys).

Keyboard Focus and Activation

The system posts keyboard messages to the message queue of the foreground thread that created the window with the keyboard focus. The keyboard is shared resource used by all the applications running under windows. A single application often shares the keyboard among several windows. A commong interface uses multiple edit controls to accept input data. The tab key moves the input focus from one edit control to the next, so the user can type into the next edit box. The active window itself or its child always has input focus. The title bar of the active window is highlighted (if only that have title bar). The keyboard focus is a temporary property of a window. The system shares the keyboard among all windows on the display by shifting the keyboard focus, at the user's direction, from one window to another. The window that has the keyboard focus receives (from the message queue of the thread that created it) all keyboard messages until the focus changes to a different window.

The keyboard focus identifies the specified window to which keyboard messages are sent. As the user types on the keyboard each keystrokes adds events to the hardware event queue. The focused window is lucky to which all keyboard messages are sent.

The concept of keyboard focus is related to that of the active window. The active window is the top-level window the user is currently working with. The window with the keyboard focus is either the active window, or a child window of the active window. To help the user identify the active window, the system places it at the top of the Z order and highlights its title bar (if it has one) and border.

A thread can call the GetFocus function to determine which of its windows (if any) currently has the keyboard focus. A thread can give the keyboard focus to one of its windows by calling the SetFocus function. When the keyboard focus changes from one window to another, the system sends a WM_KILLFOCUS message to the window that has lost the focus, and then sends a WM_SETFOCUS message to the window that has gained the focus. The wParam parameter of the message specifies the handle of the window receiving the input focus. It can also be NULL. The wParam parameter of the window function specifies the handle of the window losing the input focus. It can also be NULL.

The user can activate a top-level window by clicking it, selecting it using the ALT+TAB or ALT+ESC key combination, or selecting it from the Task List. A thread can activate a top-level window by using the SetActiveWindow function. It can determine whether a top-level window it created is active by using the GetActiveWindow function.

GetFocus and **SetFocus** functions are used to determine and change the window that owns the input focus. The **GetFocus** function returns the handle of the window that currently has the input focus. The **SetFocus** assigns the input focus to the specified window and returns the handle of the window losing the input focus, if any. Windows directs all subsequent keyboard input to the specified window. Calling **SetFocus** function genetates the WM_KILLFOCUS and WM_SETFOCUS messages.

When one window is deactivated and another activated, the system sends the WM_ACTIVATE message. The low-order word of the wParam parameter is zero if the window is being deactivated and nonzero if it is being activated. When the default window procedure receives the WM_ACTIVATE message, it sets the keyboard focus to the active window.

Note: The WM_KILLFOCUS & WM_SETFOCUS messages are not requests to change the focus; they are notifications that SetFocus has actually changed the focus.

Key Press & Release Message

Windows generates one or two messages when key is pressed: WM_KEYDOWN or WM_SYSKEYDOWN. And two messages when key is released: WM_KEYUP or WM_SYSKEYUP.

When you hold down a key until the typematic action begins, multiple key-down messages are generated, eventually followed by a single key up message when that key is released. Each keyboard message is time-stamped when it's created. You can retrieve the time a message was generated directly from MSG structure of by calling **GetMessageTime** function.

Certain key press generates the keyboard messages WM_SYSKEYDOWN and WM_SYSKEYUP. Typing a keyalong with the alt key is the most frequent source of these messages. System keyboard messages generally are passed along to the **DefWindowProc** function. Theyoften invoke from application's menu (alt plus a letter key) and the application's system menu (Alt plus a function key) and change the active window and therefore the location of input focus (Alt plus tab or Alt plus Esc).

System keyboard messages are indented for windows. Windows eventually will process such messages and generate other more specific messages indicating the result of such processing. For example: windows process the Alt plus F4 keystroke and send the message to the window to close the window.

Non-system keyboard messages are intended for your application to do with as it pleases. A key that normally generates a non system keyboard message sends a system keyboard message instead. For example: when a minimized (iconic) window is the active window (title under the icon is highligted) typically no window has the input focus. Any key pressed or released during this time generates a WM_SYSKEYDOWN and WM_SYSKEYUP messages. Windows does this for following reason.

When the application window is iconic, it isn't expecting to receive keyboard input from user. You usually don't want keyboard input triggering actions from the application when its iconic.

WM_KEYDOWN

Parameters

wParam - The virtual-key code of the nonsystem key. *[Virtual-Key Codes.]*

lParam - The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown following.

Bits	Meaning
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is always 0 for a WM_KEYDOWN message.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is zero if the key is up.
31	The transition state. The value is always 0 for a WM_KEYDOWN message.

Return Value

An application should return zero if it processes this message.

WM_KEYUP

Parameters

wParam - The virtual-key code of the nonsystem key. *[Virtual-Key Codes.]*

lParam - The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

Bits	Meaning
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always 1 for a WM_KEYUP message.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is always 0 for a WM_KEYUP message.
30	The previous key state. The value is always 1 for a WM_KEYUP message.
31	The transition state. The value is always 1 for a WM_KEYUP message.

Return Value

An application should return zero if it processes this message.

WM_SYSKEYDOWN

Parameters

wParam - The virtual-key code of the key being pressed. *[Virtual-Key Codes.]*

lParam - The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

Bits	Meaning
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.

24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is down while the key is pressed; it is 0 if the WM_SYSKEYDOWN message is posted to the active window because no window has the keyboard focus.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	The transition state. The value is always 0 for a WM_SYSKEYDOWN message.

Return Value

An application should return zero if it processes this message.

WM_SYSKEYUP

Parameters

wParam - The virtual-key code of the key being released. [*Virtual-Key Codes.]*

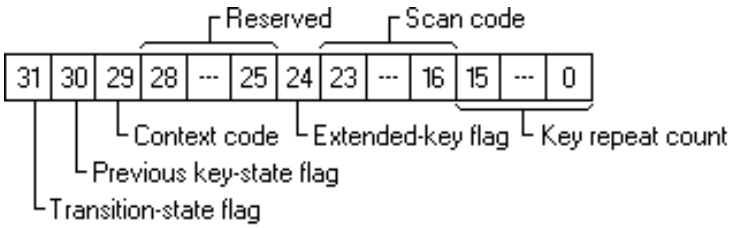
lParam - The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

Bits	Meaning
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always one for a WM_SYSKEYUP message.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is zero.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is down while the key is released; it is zero if the WM_SYSKEYDOWN message is posted to the active window because no window has the keyboard focus.
30	The previous key state. The value is always 1 for a WM_SYSKEYUP message.
31	The transition state. The value is always 1 for a WM_SYSKEYUP message.

Return Value

An application should return zero if it processes this message.

The following illustration shows the locations of flags and values in the lParam parameter.



Repeat Count

You can check the repeat count to determine whether a keystroke message represents more than one keystroke. The system increments the count when the keyboard generates WM_KEYDOWN or WM_SYSKEYDOWN messages faster than an application can process them. This often occurs when the user holds down a key long enough to start the keyboard's automatic repeat feature. Instead of filling the system message queue with the resulting key-down messages, the system combines the messages into a single key down message and increments the repeat count. Releasing a key cannot start the automatic repeat feature, so the repeat count for WM_KEYUP and WM_SYSKEYUP messages is always set to 1.

Scan Code

The scan code is the value that the keyboard hardware generates when the user presses a key. It is a device-dependent value that identifies the key pressed, as opposed to the character represented by the key. An application typically ignores scan codes. Instead, it uses the device-independent virtual-key codes to interpret keystroke messages.

Extended-Key Flag

The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the enhanced keyboard. The extended keys consist of the ALT and CTRL keys on the right-hand side of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; the NUM LOCK key; the BREAK (CTRL+PAUSE) key; the PRINT SCRN key; and the divide (/) and ENTER keys in the numeric keypad. The extended-key flag is set if the key is an extended key.

Context Code

The context code indicates whether the ALT key was down when the keystroke message was generated. The code is 1 if the ALT key was down and 0 if it was up.

Previous Key-State Flag

The previous key-state flag indicates whether the key that generated the keystroke message was previously up or down. It is 1 if the key was previously down and 0 if the key was previously up. You can use this flag to identify keystroke messages generated by the keyboard's automatic repeat feature. This flag is set to 1 for WM_KEYDOWN and WM_SYSKEYDOWN keystroke messages generated by the automatic repeat feature. It is always set to 0 for WM_KEYUP and WM_SYSKEYUP messages.

Transition-State Flag

The transition-state flag indicates whether pressing a key or releasing a key generated the keystroke message. This flag is always set to 0 for WM_KEYDOWN and WM_SYSKEYDOWN messages; it is always set to 1 for WM_KEYUP and WM_SYSKEYUP messages.

Virtual Key Codes

Virtual key codes are symbolic constant names, hexadecimal values, and keyboard equivalents for the virtual key codes used by Windows. Windows provides a virtual key code value in the wParam of the WM_KEYDOWN, WM_SYSKEYDOWN, WM_KEYUP, WM_SYSKEYUP messages. A virtual key code is device independent value for a specific key. Use of these codes isolates a window application from hardware dependencies caused by differences in keyboards for different languages. A given character generates same virtual code no matter where that key is located on a particular keyboard or what language is in use.

Windows defines special constants for each key the user can press. The virtual-key codes identify various virtual keys. These constants can then be used to refer to the keystroke when using Delphi and Windows API calls or in an OnKeyUp or OnKeyDown event handler. Virtual keys mainly consist of actual keyboard keys, but also include "virtual" elements such as the three mouse buttons. Delphi defines all constants for Windows virtual key codes in the Windows unit.

The following table shows the symbolic constant names, hexadecimal values, and mouse or keyboard equivalents for the virtual-key codes used by the system. The codes are listed in numeric order.

Constant/value		Description
VK_LBUTTON	0x01	Left mouse button
VK_RBUTTON	0x02	Right mouse button
VK_CANCEL	0x03	Control-break processing
VK_MBUTTON	0x04	Middle mouse button (three-button mouse)

VK_XBUTTON1	0x05	X1 mouse button
VK_XBUTTON2	0x06	X2 mouse button
-	0x07	Undefined
VK_BACK	0x08	BACKSPACE key
VK_TAB	0x09	TAB key
-	0x0A-0B	Reserved
VK_CLEAR	0x0C	CLEAR key
VK_RETURN	0x0D	ENTER key
-	0x0E-0F	Undefined
VK_SHIFT	0x10	SHIFT key
VK_CONTROL	0x11	CTRL key
VK_MENU	0x12	ALT key
VK_PAUSE	0x13	PAUSE key
VK_CAPITAL	0x14	CAPS LOCK key
VK_KANA	0x15	IME Kana mode
VK_HANGUEL	0x15	IME Hanguel mode (maintained for compatibility; use VK_HANGUL)
VK_HANGUL	0x15	IME Hangul mode
-	0x16	Undefined
VK_JUNJA	0x17	IME Junja mode
VK_FINAL	0x18	IME final mode
VK_HANJA	0x19	IME Hanja mode
VK_KANJI	0x19	IME Kanji mode
-	0x1A	Undefined
VK_ESCAPE	0x1B	ESC key
VK_CONVERT	0x1C	IME convert
VK_NONCONVERT	0x1D	IME nonconvert
VK_ACCEPT	0x1E	IME accept
VK_MODECHANGE	0x1F	IME mode change request
VK_SPACE	0x20	SPACEBAR
VK_PRIOR	0x21	PAGE UP key
VK_NEXT	0x22	PAGE DOWN key
VK_END	0x23	END key
VK_HOME	0x24	HOME key
VK_LEFT	0x25	LEFT ARROW key
VK_UP	0x26	UP ARROW key
VK_RIGHT	0x27	RIGHT ARROW key

VK_DOWN	0x28	DOWN ARROW key
VK_SELECT	0x29	SELECT key
VK_PRINT	0x2A	PRINT key
VK_EXECUTE	0x2B	EXECUTE key
VK_SNAPSHOT	0x2C	PRINT SCREEN key
VK_INSERT	0x2D	INS key
VK_DELETE	0x2E	DEL key
VK_HELP	0x2F	HELP key
VK_0	0x30	0 key
VK_1	0x31	1 key
VK_2	0x32	2 key
VK_3	0x33	3 key
VK_4	0x34	4 key
VK_5	0x35	5 key
VK_6	0x36	6 key
VK_7	0x37	7 key
VK_8	0x38	8 key
VK_9	0x39	9 key
-	0x3A-40	Undefined
VK_A	0x41	A key
VK_B	0x42	B key
VK_C	0x43	C key
VK_D	0x44	D key
VK_E	0x45	E key
VK_F	0x46	F key
VK_G	0x47	G key
VK_H	0x48	H key
VK_I	0x49	I key
VK_J	0x4A	J key
VK_K	0x4B	K key
VK_L	0x4C	L key
VK_M	0x4D	M key
VK_N	0x4E	N key
VK_O	0x4F	O key
VK_P	0x50	P key
VK_Q	0x51	Q key
VK_R	0x52	R key

VK_S	0x53	S key
VK_T	0x54	T key
VK_U	0x55	U key
VK_V	0x56	V key
VK_W	0x57	W key
VK_X	0x58	X key
VK_Y	0x59	Y key
VK_Z	0x5A	Z key
VK_LWIN	0x5B	Left Windows key (Natural keyboard)
VK_RWIN	0x5C	Right Windows key (Natural keyboard)
VK_APPS	0x5D	Applications key (Natural keyboard)
-	0x5E	Reserved
VK_SLEEP	0x5F	Computer Sleep key
VK_NUMPAD0	0x60	Numeric keypad 0 key
VK_NUMPAD1	0x61	Numeric keypad 1 key
VK_NUMPAD2	0x62	Numeric keypad 2 key
VK_NUMPAD3	0x63	Numeric keypad 3 key
VK_NUMPAD4	0x64	Numeric keypad 4 key
VK_NUMPAD5	0x65	Numeric keypad 5 key
VK_NUMPAD6	0x66	Numeric keypad 6 key
VK_NUMPAD7	0x67	Numeric keypad 7 key
VK_NUMPAD8	0x68	Numeric keypad 8 key
VK_NUMPAD9	0x69	Numeric keypad 9 key
VK_MULTIPLY	0x6A	Multiply key
VK_ADD	0x6B	Add key
VK_SEPARATOR	0x6C	Separator key
VK_SUBTRACT	0x6D	Subtract key
VK_DECIMAL	0x6E	Decimal key
VK_DIVIDE	0x6F	Divide key
VK_F1	0x70	F1 key
VK_F2	0x71	F2 key
VK_F3	0x72	F3 key
VK_F4	0x73	F4 key
VK_F5	0x74	F5 key
VK_F6	0x75	F6 key
VK_F7	0x76	F7 key
VK_F8	0x77	F8 key

VK_F9	0x78	F9 key
VK_F10	0x79	F10 key
VK_F11	0x7A	F11 key
VK_F12	0x7B	F12 key
VK_F13	0x7C	F13 key
VK_F14	0x7D	F14 key
VK_F15	0x7E	F15 key
VK_F16	0x7F	F16 key
VK_F17	0x80H	F17 key
VK_F18	0x81H	F18 key
VK_F19	0x82H	F19 key
VK_F20	0x83H	F20 key
VK_F21	0x84H	F21 key
VK_F22	0x85H	F22 key
VK_F23	0x86H	F23 key
VK_F24	0x87H	F24 key
-	0x88-8F	Unassigned
VK_NUMLOCK	0x90	NUM LOCK key
VK_SCROLL	0x91	SCROLL LOCK key
	0x92-96	OEM specific
-	0x97-9F	Unassigned
VK_LSHIFT	0xA0	Left SHIFT key
VK_RSHIFT	0xA1	Right SHIFT key
VK_LCONTROL	0xA2	Left CONTROL key
VK_RCONTROL	0xA3	Right CONTROL key
VK_LMENU	0xA4	Left MENU key
VK_RMENU	0xA5	Right MENU key
VK_BROWSER_BACK	0xA6	Browser Back key
VK_BROWSER_FORWARD	0xA7	Browser Forward key
VK_BROWSER_REFRESH	0xA8	Browser Refresh key
VK_BROWSER_STOP	0xA9	Browser Stop key
VK_BROWSER_SEARCH	0xAA	Browser Search key
VK_BROWSER_FAVORITES	0xAB	Browser Favorites key
VK_BROWSER_HOME	0xAC	Browser Start and Home key
VK_VOLUME_MUTE	0xAD	Volume Mute key
VK_VOLUME_DOWN	0xAE	Volume Down key
VK_VOLUME_UP	0xAF	Volume Up key

VK_MEDIA_NEXT_TRACK	0xB0	Next Track key
VK_MEDIA_PREV_TRACK	0xB1	Previous Track key
VK_MEDIA_STOP	0xB2	Stop Media key
VK_MEDIA_PLAY_PAUSE	0xB3	Play/Pause Media key
VK_LAUNCH_MAIL	0xB4	Start Mail key
VK_LAUNCH_MEDIA_SELECT	0xB5	Select Media key
VK_LAUNCH_APP1	0xB6	Start Application 1 key
VK_LAUNCH_APP2	0xB7	Start Application 2 key
-	0xB8-B9	Reserved
VK_OEM_1	0xBA	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ';:' key .
VK_OEM_PLUS	0xBB	For any country/region, the '+' key.
VK_OEM_COMMA	0xBC	For any country/region, the ',' key
VK_OEM_MINUS	0xBD	For any country/region, the '-' key
VK_OEM_PERIOD	0xBE	For any country/region, the '.' key
VK_OEM_2	0xBF	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '/?' key
VK_OEM_3	0xC0	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '`~' key
-	0xC1-D7	Reserved
-	0xD8-DA	Unassigned
VK_OEM_4	0xDB	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '[{' key
VK_OEM_5	0xDC	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '|' key
VK_OEM_6	0xDD	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ']}' key
VK_OEM_7	0xDE	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the 'single-quote/double-quote' key
VK_OEM_8	0xDF	Used for miscellaneous characters; it can vary by keyboard.
-	0xE0	Reserved
	0xE1	OEM specific
VK_OEM_102	0xE2	Either the angle bracket key or the backslash key on the RT 102-key keyboard.
	0xE3-E4	OEM specific
VK_PROCESSKEY	0xE5	IME PROCESS key

	0xE6	OEM specific
VK_PACKET	0xE7	Used to pass Unicode characters as if they were keystrokes. The VK_PACKET key is the low word of a 32-bit Virtual Key value used for non-keyboard input methods. For more information, see Remark in KEYBDINPUT, SendInput, WM_KEYDOWN, and WM_KEYUP
-	0xE8	Unassigned
	0xE9-F5	OEM specific
VK_ATTN	0xF6	Attn key
VK_CRSEL	0xF7	CrSel key
VK_EXSEL	0xF8	ExSel key
VK_EREOF	0xF9	Erase EOF key
VK_PLAY	0xFA	Play key
VK_ZOOM	0xFB	Zoom key
VK_NONAME	0xFC	Reserved
VK_PA1	0xFD	PA1 key
VK_OEM_CLEAR	0xFE	Clear key

Character Message

Keystroke messages provide a lot of information about keystrokes, but they don't provide character codes for character keystrokes. To retrieve character codes, an application must include the TranslateMessage function in its thread message loop. TranslateMessage passes a WM_KEYDOWN or WM_SYSKEYDOWN message to the keyboard layout. The layout examines the message's virtual-key code and, if it corresponds to a character key, provides the character code equivalent (taking into account the state of the SHIFT and CAPS LOCK keys). It then generates a character message that includes the character code and places the message at the top of the message queue. The next iteration of the message loop removes the character message from the queue and dispatches the message to the appropriate window procedure.

Keystroke message aren't very convenient while processing character keys such as letters and numbers. VK_A virtual key doesn't tell whether it's uppercase A or lowercase a or Ctrl plus A. You can look at the state information about the shift and ctrl keys but also need to know country-dependent information about the keyboard to properly decide what character corresponds to a particular keystroke messages because of how extended character sets are used.

The windows API contains function for converting between virtual key code messages and WM_CHAR messages which is also called printable character messages because most are uppercase letters, lowercase letters, numbers or punctuation marks. That is they correspond to characters that can be seen in the display device either monitor or the printer. Windows provides **TranslateMessage** function in message loop which performs exactly as per our requirement in this case.

```
while(GetMessage(&msg, 0,0,0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

TranslateMessage function produces a character message for each virtual-key message that corresponds to a character from the ANSI or Unicode character set.

A window procedure can receive the following character messages: WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, WM_SYSDEADCHAR, and WM_UNICHAR. The TranslateMessage function generates a WM_CHAR or WM_DEADCHAR message when it processes a WM_KEYDOWN message. Similarly, it generates a WM_SYSCHAR or WM_SYSDEADCHAR message when it processes a WM_SYSKEYDOWN message.

The Character message is placed at the head of the queue. That message will be either one: WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, WM_SYSDEADCHAR. Every WM_CHAR message will be preceded by WM_KEYDOWN message and followed by a WM_KEYUP message. It will be the next message retrieved by GetMessage or PeekMessage function call. The original virtual-key message isn't changed in any way. The **TranslateMessage**() function simply uses the information from the virtual key messages to produce a corresponding character message for keys that map to ANSI or Unicode character. The contents of the lParam parameter of a character message are identical to the contents of the lParam parameter of the key-down message that was translated to produce the character message.

In summary, Keyboard input starts as scan codes. A windows keyboard driver converts these codes to hardware-independent form: the virtual key codes. Typically, windows application then sends those virtual key codes to windows API functions that generates WM_CHAR character messages when printable character are typed. For non printable character like function keys and navigation keys we need to relay on the WM_KEYDOWN virtual key message.

The character Sets

Virtually everyone knows that a computer doesn't understand characters, it understands numbers. Thus, each character you see on the screen in a program such as Word is maintained internally as a number. The "mapping" of characters to numbers is known as a character set.

A "character set" is a mapping of characters to their identifying code values. The character set most commonly used in computers today is Unicode, a global standard for character encoding. Internally, Windows applications use the UTF-16 implementation of Unicode. In UTF-16, most characters are identified by two-byte codes.

A character set is a convention that describes how character codes are displayed on a graphics display device. They are displayed on display screens and printed output by calling various GDI text drawing functions.

In the relatively short history of computers, there have been several different character sets used. Some are described below:

Windows Character Set

The Windows character set is the most commonly used character set. It is essentially equivalent to the ANSI character set. The blank character is the first character in the Windows character set. It has a hexadecimal value of 0x20 (decimal 32). The last character in the Windows character set has a hexadecimal value of 0xFF (decimal 255).

Many fonts specify a default character. Whenever a request is made for a character that is not in the font, the system provides this default character. Many fonts using the Windows character set specify the period (.) as the default character. TrueType and OpenType fonts typically use an open box as the default character.

Fonts use a break character called a quad to separate words and justify text. Most fonts using the Windows character set specify that the blank character will serve as the break character.

OEM Character Set (MS-DOS character set)

OEM stands for Original Equipment manufacturer. The OEM character set is the character set build into the underlying machine. The OEM character set actually refers to any hardware specific character sets. For example, The VGA adaptor has characters burned into ROM. This type of hardware specific character is what programmers need to keep in mind when they program for those devices.

The OEM character set is typically used in full-screen MS-DOS sessions for screen display. Characters 32 through 127 are usually the same in the OEM, U.S. ASCII, and Windows character sets. The other characters in the OEM character set (0 through 31 and 128 through 255) correspond to the characters that can be displayed in a full-screen MS-DOS session. These characters are generally different from the Windows characters.

OEM is actually a misleading term for a company that has a special relationship with computer producers. OEMs are manufacturers who resell another company's product under their own name and branding. It refers specifically to the act of a company rebranding a product to its own name and offering its own warranty, support and licensing of the product. The term is really a misnomer because OEMs are not the original manufacturers; they are the customizers.

In windows programming we call this MS_DOS character set rather than OEM Character set.

The ANSI and ASCII Character set

The first character set used in small computers was ASCII, which is an acronym for American Standard Code for Information Interchange. It started as a code of 128 characters, using seven bits to represent all the characters. (A bit is a binary digit; it can have either two values: on or off. Thus, seven bits can have 2^7 or 128 possible unique values.). The codes from 0-31 are control characters and are not displayable. The code from 32-126 represents displayable characters.

ASCII was first developed for machines that used only seven bits of each byte (such as teletypes). Early personal computers, however, used eight bits, and thus could utilize 2^8 or 256 possible values for a character code. This led to what was known as extended ASCII, where the first 128 characters matched those in ASCII, but the second 128 were left up to the computer manufacturer. In early IBM PC models, the extended ASCII character set included some foreign-language symbols and many line-drawing characters, used for basic graphics.

It is interesting to note that ASCII has not given way to extended ASCII in all computer systems. Indeed, many mainframe and mini computer systems still support strictly seven-bit ASCII. Windows versions through Windows 95 utilize what is called an ANSI character set. When Microsoft first created windows, the American National Standard Institute (ANSI) had defined standards for character set, which are similar but not identical to the ASCII character. This is a single-byte character set that can represent up to 256 characters. The original ASCII character set occupies the first 128 characters of the ANSI set used in Windows.

Unicode Character Set

The Windows character set uses 8 bits to represent each character; therefore, the maximum number of characters that can be expressed using 8 bits is 256 (2^8). This is usually sufficient for Western languages, including the diacritical marks used in French, German, Spanish, and other languages. However, Eastern languages employ thousands of separate characters, which cannot be encoded by using a single-byte coding scheme. With the proliferation of computer commerce, double-byte coding schemes were developed so that characters could be represented in 8-bit, 16-bit, 24-bit, or 32-bit sequences. This requires complicated passing algorithms; even so, using different code sets could yield entirely different results on two different computers.

To address the problem of multiple coding schemes, the Unicode standard for data representation was developed. A 16-bit character coding scheme, Unicode can represent 65,536 (2^16) characters, which is enough to include all languages in computer commerce today, as well as punctuation marks, mathematical symbols, and room for expansion. Unicode establishes a unique code for every character to ensure that character translation is always accurate.

Compared to older mechanisms for handling character and string data, Unicode simplifies software localization and improves multilingual text processing. By using Unicode to represent character and string data in your applications, you can enable universal data exchange capabilities for global marketing, using a single binary file for every possible character code. Unicode does the following:

- Allows any combination of characters, drawn from any combination of scripts and languages, to co-exist in a single document.
- Defines semantics for each character.
- Standardizes script behavior.
- Provides a standard algorithm for bidirectional text.
- Defines cross-mappings to other standards.
- Defines multiple encodings of its single character set: UTF-7, UTF-8, UTF-16, and UTF-32. Conversion of data among these encodings is lossless.

Unicode supports numerous scripts used by languages around the world, and also a large number of technical symbols and special characters used in publishing. The supported scripts include, but are not limited to, Latin, Greek, Cyrillic, Hebrew, Arabic, Devanagari, Thai, Han, Hangul, Hiragana, and Katakana. Supported languages include, but are not limited to, German,

French, English, Greek, Russian, Hebrew, Arabic, Hindi, Thai, Chinese, Korean, and Japanese. Unicode currently can represent the vast majority of characters in modern computer use around the world, and continues to be updated to make it even more complete.

Contd. ...

Mouse Input

The mouse is an optional pointing device for graphical operating system like windows. The small bitmap image that represents the pointer is called cursor. As the user moves the mouse input device, the cursor moves accordingly. A user can perform many different functions with the mouse such as selecting items from menu, clicking on the button and indicating areas of interests in a window. If any application wants the keyboard to move cursor your application has to program explicitely.

Windows supports a mouse with one, two or three buttons. Windows refers to the buttons as the left button, the middle button and right button. Those who prefer to use the mouse with their left hands can tell windows via control panel to swap the buttons. Then the left mouse button sends right-mouse-button messages and the right mouse button sends left-mouse-button messages. This is because the buttons are described in terms of their physical placement on the mouse rather than logical usage.

If we want to check to see whether a mouse is present use GetSystemMetrics function. The GetSystemMetrics function call returns a nonzero value when a mouse is installed.

GetSystemMetrics

Retrieves the specified system metric or system configuration setting. Note that all dimensions retrieved by GetSystemMetrics are in pixels.

```
int WINAPI GetSystemMetrics(
        int nIndex
);
```

Parameters

nIndex - The system metric or configuration setting to be retrieved. This parameter can be one of the following values (**Only few are given here and description is not included.**).

- SM_ARRANGE
- SM_CLEANBOOT
- SM_CMONITORS
- SM_CMOUSEBUTTONS

- SM_CXBORDER
- SM_CXCURSOR
- SM_MOUSEBUTTONS
- SM_MOUSEPRESENT

E.g.

```
int MousePresent = GetSystemMetrics(SM_MOUSEPRESENT);
// to find the number of mouse buttons
int mousebuttons= GetSystemMetrics(SM_CMOUSEBUTTONS);
```

Mouse Messages

Like keyboard input, mouse input arrives in the form of messages. These are called mouse messages. There are 22 different mouse messages but only three are commonly used: WM_LBUTTONDOWN, WM_LBUTTONUP and WM_MOUSEMOVE. The mouse messages divided into three groups. The 10 messages in the first group are the message they signals in the windows client area. The next 10 messages in the next group signal the mouse actions in non-client area of a window.

The remaining two messages are the WM_NCHITTEST and WM_MOUSEACTIVATE messages. Each time the user moves the mouse or press and release a mouse button. Windows sends a series of messages to the window function for the window under the cursor.

Client Area mouse messages	
WM_LBUTTONDBLCLK	Left mouse button has been double clicked in the client area.
WM_LBUTTONDOWN	Left mouse button has been clicked in the client area.
WM_LBUTTONUP	Left mouse button has been double clicked & released in the client area.
WM_ MBUTTONDBLCLK	Middle mouse button has been double clicked in the client area.
WM_ MBUTTONDOWN	Middle mouse button has been clicked in the client area.

WM_ MBUTTONUP	Middle mouse button has been double clicked & released in the client area.
WM_ RBUTTONDBLCLK	Right mouse button has been double clicked in the client area.
WM_ RBUTTONDOWN	Right mouse button has been clicked in the client area.
WM_ RBUTTONUP	Right mouse button has been double clicked & released in the client area.
WM_MOUSEMOVE	Mouse has been moved in client area.
Non - Client Area mouse messages	
WM_NCLBUTTONDBLCLK	Left mouse button has been double clicked in the non-client area.
WM_NCLBUTTONDOWN	Left mouse button has been clicked in the non-client area.
WM_NCLBUTTONUP	Left mouse button has been double clicked & released in the non-client area.
WM_ NCMBUTTONDBLCLK	Middle mouse button has been double clicked in the non-client area.
WM_ NCMBUTTONDOWN	Middle mouse button has been clicked in the non-client area.
WM_ NCMBUTTONUP	Middle mouse button has been double clicked & released in the non-client area.
WM_ NCRBUTTONDBLCLK	Right mouse button has been double clicked in the non-client area.
WM_ NCRBUTTONDOWN	Right mouse button has been clicked in the non-client area.

WM_ NCRBUTTONUP	Right mouse button has been double clicked & released in the non-client area.
WM_NCMOUSEMOVE	Mouse has been moved in non-client area.
Mouse Cursor Message	
WM_NCHITTEST	Mouse has been moved, location not yet as unknown. This is first time generated.
WM_SETCURSOR	Mouse button has been pressed over an inactive window.

While moving the mouse and pressing or releasing a mouse button generates a series of messages. The first message that arrives is a WM_NCHITTEST message. As the mouse cursor flies over different windows, it broadcasts a constant stream of messages. The results of these messages allow windows to set the mouse cursor to the correct shape. These messages are sent to whichever window or even the focus window.

The first message, WM_NCHITTEST is a query. With this message, windows is asking the window to identify which part of the window the cursor is above. Windows is not looking for coordinates. Instead it is asking for a hit test code that indicates weather the cursor is over.

E.g. the client area or Caption bar.

The wParam parameter to a window function isn't used for a WM_NCHITTEST message. The lParam parameter contains the horizontal and vertical screen coordinates of the hot spot of the curson. The horizontal coordinate is in the low order word and the vertical coordinate in the high order word.

Mouse activation message

At certain times an application's activation is tied to mouse input. In particular, when a user clicks on an inactive top-level window, a WM_MOUSEACTIVATE message is sent to that window. You may reply to that message yourself or you can let windows default handler reply.

The WM_MOUSEACTIVATE is a query and the system wants an answer. Note that WM_ACTIVATE – a message that might seem related is notification not a query. The systemdoesn't care what you do with notification, it's your job to respond or ignore it. With a query message on the other hand, the return value is very important because windows pays attention to your reply.

WM_MOUSEACTIVATE

The WM_MOUSEACTIVATE message is sent when the cursor is in an inactive window and the user presses a mouse button.

A window receives this message through its WindowProc function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // Handle to window
    UINT uMsg,          // WM_MOUSEACTIVATE
    WPARAM wParam,      // Handle to parent
    LPARAM lParam       // Hit-Test value and message
    );
```

wParam – Handle to the top level parent window of the window being activated.

lParam

The low order word specifies the hit-test value returned by the DefWindowProc function as a result of processing the WM_NCHITTEST message.

The high order word specifies the identifier of the mouse message generated when the user pressed a mousebutton. The mouse message is either discarded or posted to the window, depending on the return value.

Return Values

The return value specifies whether the window should be activated and whether the identifier of the mouse should be discarded. It must be one of the following values:

Value	Meaning
MA_ACTIVATE	Activates the window, and does not discard the mouse message.

MA_ACTIVATEANDEAT	Activates the window and discards the mouse message.
MA_NOACTIVATE	Does not activate the window and doesnot discard the mouse message.
MA_NOACTIVATEANDEAT	Does not activate the window, but discards the mouse message.

Capturing the Mouse

Sometimes you have to ask the mouse to give full attention to a single window. This could be done by capturing the mouse. Capturing the mouse forces all mouse messages to be sent to a single application window. Example includes drag and drop, dragging to select multiple items (such as multiple lines in a word processor) and dragging to draw. Capturing the mouse is necessary to force the mouse to stay in touch until it finishes what it has started.

Mouse capture can be seen within the most text editing windows. These windows scroll when you click in window filled with text and drag beyond the window border. These windows are only able to continue receiving mouse input – in the form of WM_MOUSEMOVE messages – because mouse has been captured.

All mouse input can be captured using the SetCapture function and wait for the user to release the mouse button before releasing the captured input. When use of mouse is finished, ReleaseCapture must be called to release it an enable the multitasking of the mouse.

Mouse Capture Operations

- **SetCapture**

Sets the mouse capture to the specified window belonging to the current thread.SetCapture captures mouse input either when the mouse is over the capturing window, or when the mouse button was pressed while the mouse was over the capturing window and the button is still down. Only one window at a time can capture the mouse.

If the mouse cursor is over a window created by another thread, the system will direct mouse input to the specified window only if a mouse button is down.

```
HWND WINAPI SetCapture(
        HWND hWnd
);
```

Parameters

hWnd - A handle to the window in the current thread that is to capture the mouse.

Return Value

The return value is a handle to the window that had previously captured the mouse. If there is no such window, the return value is NULL.

Remarks

Only the foreground window can capture the mouse. When a background window attempts to do so, the window receives messages only for mouse events that occur when the cursor hot spot is within the visible portion of the window. Also, even if the foreground window has captured the mouse, the user can still click another window, bringing it to the foreground.

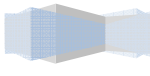
When the window no longer requires all mouse input, the thread that created the window should call the ReleaseCapture function to release the mouse. This function cannot be used to capture mouse input meant for another process.

When the mouse is captured, menu hotkeys and other keyboard accelerators do not work.

- **ReleaseCapture**

Releases the mouse capture from a window in the current thread and restores normal mouse input processing. A window that has captured the mouse receives all mouse input, regardless of the position of the cursor, except when a mouse button is clicked while the cursor hot spot is in the window of another thread.


```
BOOL WINAPI ReleaseCapture(void);
```



Parameters

This function has no parameters.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information GetLastError can be called.

Remarks

An application calls this function after calling the SetCapture function.

GetCapture

Retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.

HWND WINAPI GetCapture(void);

Parameters

This function has no parameters.

Return Value

The return value is a handle to the capture window associated with the current thread. If no window in the thread has captured the mouse, the return value is NULL.

Remarks

A NULL return value means the current thread has not captured the mouse. However, it is possible that another thread or process has captured the mouse.

- **ClipCursor**

Confines the cursor to a rectangular area on the screen. If a subsequent cursor position (set by the SetCursorPos function or the mouse) lies outside the rectangle, the system automatically adjusts the position to keep the cursor inside the rectangular area.

```
BOOL WINAPI ClipCursor(
        const RECT *lpRect
);
```

Parameters

lpRect - A pointer to the structure that contains the screen coordinates of the upper-left and lower-right corners of the confining rectangle. If this parameter is NULL, the cursor is free to move anywhere on the screen.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The cursor is a shared resource. If an application confines the cursor, it must release the cursor by using ClipCursor before relinquishing control to another application.

GetLastError Function

Retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

```
DWORD WINAPI GetLastError(void);
```

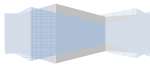
This function has no parameters.

Return Value

The return value is the calling thread's last-error code. The Return Value section of the documentation for each function that sets the last-error code notes the conditions under which the function sets the last-error code. Most functions that set the thread's last-error code set it when they fail. However, some functions also set the last-error code when they succeed. If the function is not documented to set the last-error code, the value returned by this function is simply the most recent last-error code to have been set; some functions set the last-error code to 0 on success and others do not.

For a complete list of error codes provided by the operating system, see System Error Codes.

Contd. ...



Timer Input

Timer is also an input device which can ask windows to periodically notify the application when specified time has elapsed. This notification comes in the form of a WM_TIMER message.

Application request the use of timer by calling the SetTimer function specifying how often windows should send WM_TIMER message (in milliseconds). When you call the SetTimer function, you actually request windows to create a system timer event. You are not actually using the hardware timer. Windows itself uses the hardware timer to simulate multiple logical timers.

- **SetTimer**

Creates a timer with the specified time-out value.

```
UINT_PTR WINAPI SetTimer(
        HWND hWnd,
        UINT_PTR nIDEvent,
        UINT uElapse,
        TIMERPROC lpTimerFunc
);
```

Parameters

hWnd - A handle to the window to be associated with the timer. This window must be owned by the calling thread. If a NULL value for hWnd is passed in along with an nIDEvent of an existing timer, that timer will be replaced in the same way that an existing non-NULL hWnd timer will be.

nIDEvent - A nonzero timer identifier. If the hWnd parameter is NULL, and the nIDEvent does not match an existing timer then it is ignored and a new timer ID is generated. If the hWnd parameter is not NULL and the window specified by hWnd already has a timer with the value nIDEvent, then the existing timer is replaced by the new timer. When SetTimer replaces a timer, the timer is reset. Therefore, a message will be sent after the current time-out value elapses, but the previously set time-out value is ignored. If the call is not intended to replace an existing timer, nIDEvent should be 0 if the hWnd is NULL.

uElapse - The time-out value, in milliseconds.

lpTimerFunc - A pointer to the function to be notified when the time-out value elapses. For more information about the function, see TimerProc. If lpTimerFunc is NULL, the system posts a WM_TIMER message to the application queue. The hwnd member of the message's MSG structure contains the value of the hWnd parameter.

Return Value

If the function succeeds and the hWnd parameter is NULL, the return value is an integer identifying the new timer. An application can pass this value to the KillTimer function to destroy the timer. If the function succeeds and the hWnd parameter is not NULL, then the return value is a nonzero integer. An application can pass the value of the nIDEvent parameter to the KillTimer function to destroy the timer.

If the function fails to create a timer, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application can process WM_TIMER messages by including a WM_TIMER case statement in the window procedure or by specifying a TimerProc callback function when creating the timer. When you specify a TimerProc callback function, the default window procedure calls the callback function when it processes WM_TIMER. Therefore, you need to dispatch messages in the calling thread, even when you use TimerProc instead of processing WM_TIMER.

The wParam parameter of the WM_TIMER message contains the value of the nIDEvent parameter.

The timer identifier, nIDEvent, is specific to the associated window. Another window can have its own timer which has the same identifier as a timer owned by another window. The timers are distinct.

SetTimer can reuse timer IDs in the case where hWnd is NULL.

- **KillTimer**

Destroys the specified timer.

```
BOOL WINAPI KillTimer(
        HWND hWnd,
        UINT_PTR uIDEvent
);
```

Parameters

hWnd - A handle to the window associated with the specified timer. This value must be the same as the hWnd value passed to the SetTimer function that created the timer.

uIDEvent - The timer to be destroyed. If the window handle passed to SetTimer is valid, this parameter must be the same as the nIDEvent value passed to SetTimer. If the application calls SetTimer with hWnd set to NULL, this parameter must be the timer identifier returned by SetTimer.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The KillTimer function does not remove WM_TIMER messages already posted to the message queue.

E.g.

```
// Set two timers.
SetTimer(hwnd,             // handle to main window
    IDT_TIMER1,            // timer identifier
    10000,                 // 10-second interval
    (TIMERPROC) NULL);     // no timer callback

SetTimer(hwnd,             // handle to main window
    IDT_TIMER2,            // timer identifier
    300000,                // five-minute interval
    (TIMERPROC) NULL);     // no timer callback
```

To process the WM_TIMER messages generated by these timers, add a WM_TIMER case statement to the window procedure for the hwnd parameter.

```
case WM_TIMER:
     switch (wParam)
    {
        case IDT_TIMER1:
            // process the 10-second timer
              return 0;
```

```
        case IDT_TIMER2:
            // process the five-minute timer
            return 0;
    }
```

Applications should use the KillTimer function to destroy timers that are no longer necessary. The following example destroys the timers identified by the constants IDT_TIMER1 & IDT_TIMER2.

```
// Destroy the timers.

KillTimer(hwnd, IDT_TIMER1);
KillTimer(hwnd, IDT_TIMER2);
```

Windows handles a WM_TIMER message like it does a WM_PAINT message. Both are considered low priority message and are not delivered when other applications have higher-priority messages (meaning any message other than WM_PAINT and WM_TIMER) in their Queues. Instead, the other applications are given control and allowed to process their messages. They can result in your application's not receiving its WM_TIMER message when due.

In fact, multiple timer periods might expire before your application gets to retrieve its WM_TIMER message from its queue. However, you'll receive only one WM_TIMER message and it will represent all expired periods. Windows doesn't place a second WM_TIMER Message in the queue just as it doesn't place multiple WM_PAINT messages in the queue.

Like the WM_PAINT message, the WM_TIMER message is not actually ever in the message queue. It is actually a flag in the thread's queue header. Therefore, there is only one flag to represent the timer. This flag is set every time the timer expires. If it is set and there is nothing else in the queue the GetMessage or PeekMessage function will return WM_TIMER message.

The buttom line is that you must not rely on counting WM_TIMER message to determine elapsed time. A WM_TIMER message might arrive as early as 54 milliseconds before the exlected time or at indeterminate amount of time letter. Depending upon the behavior of application and other running application, the WM_TIMER message could arrive milliseconds, seconds or minutes after it was supposed to arrive. When you need to know the precise amount of time that has elapsed, such as when updating a clock, use the arrival of WM_TIMER message as a trigger to fetch the actual time and use it accordingly.

Chapter 06: Using Controls

Overview

A control is a child window an application uses in conjuction with another window to perform simple input and output tasks. Controls are most often used within dialog boxes but they can also be used in other windows controls within dialog boxes provide the user with the means to type text, choose options and direct a dialog boxto complete its action. Controls in other windows provides a variety of services, such as letting the user coosing commands, view status and view and edit text.

Windows sends message for a window function specified when registering the class. The window functions for the predefined classes reside within windows. The windows based controls are child windows that are created from classes which a windows has defined for you. So when you create a control you are creating a child window the same as any other child window in the application.

The window function determines what a window looks like and how it responds to input. You use the predefined window class when you need a window that behaves in a standard way. For Example: the BUTTON window class can draw its window in a number of button styles and report when the user clicks a mouse button while over the other window. You receive two major benefits when you use control. First, you don't need to write, debug and maintain code for a control. Second, by using standard controls where applicable, you make it much easier for a user to learn how to run and use your application.

You should whenever possible use standard controls. With this you not only get a consistent set of behavior but development cost also drops. A control handles the messy details of managing program input and notifies its parent when anything of interest orrurs. For example: when you click a push button the push button control redraws the button in the down state when WM_LBUTTONDOWN message arrives. When the WM_LBUTTONUP message arrives the control redraws the button in the up state and sends a message to its parent window signaling that button was pressed.

There are two ways to create a control:

1. You create a child window (control) and specify a predefined class name as the window class.
2. You create control as the part of a dialog box.

Standard button controls

1. **BUTTON**
2. **STATIC**
3. **EDIT**
4. **LISTBOX**
5. **COMBOBOX**
6. **SCROLLBAR**
7. **MDICLIENT**

These are window based controls that have been in windows since win16 programming. There is an additional set of controls, the common control. In addition to the base controls and common controls you can obtain from variety of sources any number of custom controls which operate very much like the base controls or common controls.

Creating a control in a window

You create a control in the client area of tis parent window by calling the **CreateWindow** function and specifying the predefined class name as the window class.

We have already seen the parameters of CreateWindow function in **Chapter 2.**

E.g.

```
hwndButton=CreateWindow(
                "BUTTON",           // Button Class
                "OK",               // Button Caption
                WS_VISIBLE |        // Make button visible
                WS_CHILD |          // Controls are child
                BS_DEFPUSBUTTON,    // Default push button
                10,                 // Starting x-coordinate
                10,                 // Starting y - coordinate
                100,                // Button Width
                100,                // Button Height
                hwnd,               // Parent Window Handle
                (HMENU)IDOK,        // Control ID
                hInstance,          // Instance handle
                NULL);              // No Additional Data
```

The statement creates a child window that is 100 pixel wide and 100 pixels high. Windows places the window at the point (10, 10) in the client area of the **hwnd** window. The window belongs to the the **BUTTON** class, one of the predefined window classes. The window class name parameter is not case sensitive. You can use the names button, Button & BUTTON interchangeably. The third parameter to the CreateWindow function call (the 32bit window style paraemter) has two parts: the window styles (these are the symbolic identifiers beginning with WS_) in high-order word and the control styles in the low-order word window styles apply to all types of controls. Hense you normally use the WS_VISIBLE window style when creating any type of control so that normally the control becomes visible when the parent window is visible. Controls are always WS_CHILD window. You combine window styles and controls together using the bitwise OR operations.

The BS_DEFPUSHBUTTON is a control style, specifically a button style that applies only to window based on the BUTTON class. The control styles you can select depend upon the control class. You specify a control class to choose the general type of control e.g. button, scrollbar, edit etc. and then specify the control styles to indicate the exact type of control.

Child window don't have menu. The hMenu parameter of a CreateWindow function call that creates a child window is actually a child window identifier, not a menu handle. In the previous example the IDOK is associated with the created push button control. You must cast the child window identifier to the HMENU data types when compiling with strict type checking.

A control interacts with its parent window by sending a notification message. There are several types of notification message. A control includes the child window identifier when it sends a message to its parent window. A parent window containing multiple controls can distinguish between the controls by their IDs.

Control Notification

A control notifies its parent when something occurs of possible interest to the parent. Most of the base controls do this by sending a WM_COMMAND message to their parent. The low order word of the wParam parameter of the message contains the ID of the control sending the message and the type of event that generated the notification. The control ID value allows determining which control sent the WM_COMMAND message. The high order word of the wParam parameter of WM_COMMAND message contains the notification code. The notification

code contains additional informations from the control explaining what the message means. Notification code values have different meanings depending on the type of control.

Using control

Using control in a window requires the following steps:

1. Select a control class and a control style based on the appearance and function of the desired control.
2. Create the control at the appropriate location in the client area of the window.
3. Send control any messages needed to set its initial value.
4. Make the control visible when it doesn't have WS_VISIBLE style.
5. Process WM_COMMAD message it the parent window's function to react to the control's change pf state.

Remember that a control is simply a particular type of child window. Everything that applies to the child window applies to a control. You cannot create a control as a top level window, a control must have a parent window and top level window doesn't have a parent. Actions affecting the parent window also affect the control. When you want to move the control within the client area of its parent, you call the **MoveWindow** function.

- **MoveWindow**

Changes the position and dimensions of the specified window. For a top-level window, the position and dimensions are relative to the upper-left corner of the screen. For a child window, they are relative to the upper-left corner of the parent window's client area.

```
BOOL WINAPI MoveWindow(
        HWND hWnd,
        int X,
        int Y,
        int nWidth,
        int nHeight,
        BOOL bRepaint
        );
```

Parameters

hWnd - A handle to the window.

X - The new position of the left side of the window.

Y - The new position of the top of the window.

nWidth - The new width of the window.

nHeight - The new height of the window.

bRepaint - Indicates whether the window is to be repainted. If this parameter is TRUE, the window receives a message. If the parameter is FALSE, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of moving a child window.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

- **EnableWindow**

Enables or disables mouse and keyboard input to the specified window or control. When input is disabled, the window does not receive input such as mouse clicks and key presses. When input is enabled, the window receives all input.

```
BOOL WINAPI EnableWindow(
    HWND hWnd,
    BOOL bEnable
);
```

Parameters

hWnd - A handle to the window to be enabled or disabled.

bEnable - Indicates whether to enable or disable the window. If this parameter is TRUE, the window is enabled. If the parameter is FALSE, the window is disabled.

Return Value

If the window was previously disabled, the return value is nonzero. If the window was not previously disabled, the return value is zero.

- **ShowWindow**

Sets the specified window's show state.

```
BOOL WINAPI ShowWindow(
    HWND hWnd,
    int nCmdShow
);
```

Parameters

hWnd - A handle to the window.

nCmdShow - Controls how the window is to be shown. This parameter is ignored the first time an application calls ShowWindow, if the program that launched the application provides a STARTUPINFO structure. Otherwise, the first time ShowWindow is called, the value should be the value obtained by the WinMain function in its nCmdShow parameter. In subsequent calls, this parameter can be one of the following values.

Value		Meaning
SW_FORCEMINIMIZE	11	Minimizes a window, even if the thread that owns the window is not responding. This flag should only be used when minimizing windows from a different thread.
SW_HIDE	0	Hides the window and activates another window.
SW_MAXIMIZE	3	Maximizes the specified window.
SW_MINIMIZE	6	Minimizes the specified window and activates the next top-level window in the Z order.
SW_RESTORE	9	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	5	Activates the window and displays it in its current size and position.
SW_SHOWDEFAULT	10	Sets the show state based on the SW_ value specified in the STARTUPINFO structure passed to the CreateProcess function by the program that started the application.
SW_SHOWMAXIMIZED	3	Activates the window and displays it as a maximized window.
SW_SHOWMINIMIZED	2	Activates the window and displays it as a minimized window.
SW_SHOWMINNOACTIVE	7	Displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED, except the window is not activated.
SW_SHOWNA	8	Displays the window in its current size and position. This value is similar to SW_SHOW, except that the window is not activated.

SW_SHOWNOACTIVATE	4	Displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL, except that the window is not activated.
SW_SHOWNORMAL	1	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 window was previously visible, the return value is nonzero. If the window was previously hidden, the return value is zero.

- **DestroyWindow**

Destroys the specified window. The function sends WM_DESTROY and WM_NCDESTROY messages to the window to deactivate it and remove the keyboard focus from it. The function also destroys the window's menu, flushes the thread message queue, destroys timers, removes clipboard ownership, and breaks the clipboard viewer chain (if the window is at the top of the viewer chain).

If the specified window is a parent or owner window, DestroyWindow automatically destroys the associated child or owned windows when it destroys the parent or owner window. The function first destroys child or owned windows, and then it destroys the parent or owner window. DestroyWindow also destroys modeless dialog boxes created by the CreateDialog function.

```
BOOL WINAPI DestroyWindow(
        HWND hWnd
);
```

Parameters

hWnd - A handle to the window to be destroyed.

Return Value

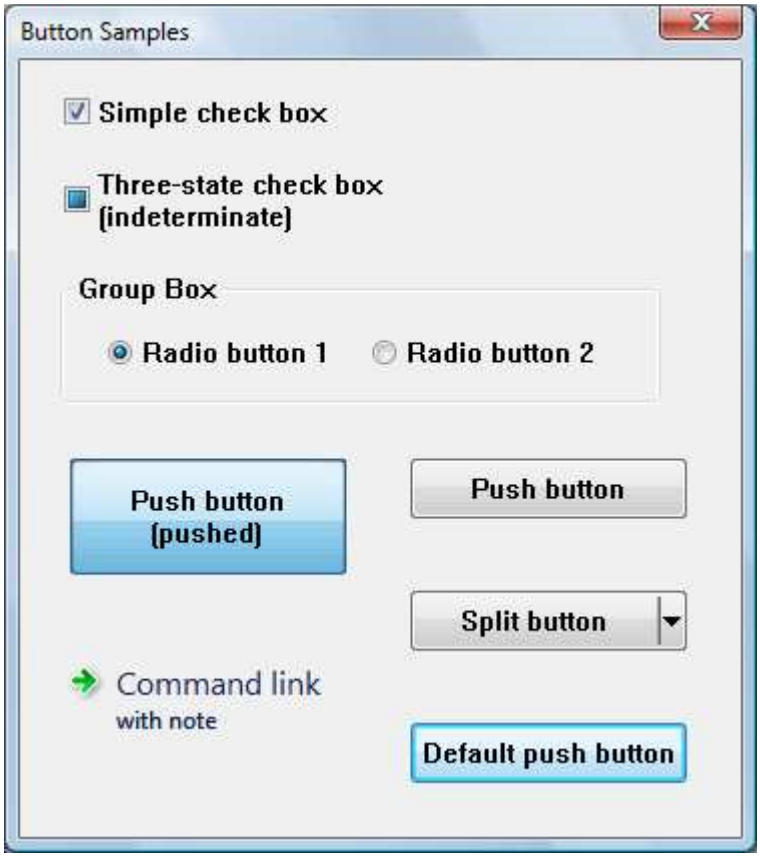
If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A thread cannot use DestroyWindow to destroy a window created by a different thread. If the window being destroyed is a child window that does not have the WS_EX_NOPARENTNOTIFY style, a WM_PARENTNOTIFY message is sent to the parent.

BUTTON CLASS

Dialog boxes and controls support communication between an application and the user. A button is a control the user can click to provide input to an application.



Control Associated with BUTTON Class

- **PushButtons**
- **DefPushButtons**
- **CheckBox**
- **RadioButton**
- **GroupBox**

There are several types of buttons and one or more button styles to distinguish among buttons of the same type. The user clicks a button using the mouse or keyboard. Clicking a button typically changes its visual appearance and state. The system, the button and the application cooperate in changing the button's appearance and state. A button can send messages to its

parent window and a parent window can send messages to a button. Some buttons are painted by the system, some by the application. Buttons can be used alone or in groups and can appear with or without application defined text (a label). They belong to the BUTTON class.

Class Styles

Once BUTTON class is chosen, particular button style of button must be selected. To create a specific button style. You create a window of class BUTTON and specify the WS_CHILD window style plus the appropriate button style. You should logically OR the WS_CHILD with the other styles.

Constant		Description
BS_3STATE	*	Creates a button that is the same as a check box, except that the box can be grayed as well as checked or cleared. Use the grayed state to show that the state of the check box is not determined.
BS_AUTO3STATE	*	Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, indeterminate, and cleared.
BS_AUTOCHECKBOX	*	Creates a button that is the same as a check box, except that the check state automatically toggles between checked and cleared each time the user selects the check box.
BS_AUTORADIOBUTTON	*	Creates a button that is the same as a radio button, except that when the user selects it, the system automatically sets the button's check state to checked and automatically sets the check state for all other buttons in the same group to cleared.
BS_CHECKBOX	*	Creates a small, empty check box with text. By default, the text is displayed to the right of the check box. To display the text to the left of the check box, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style).
BS_DEFPUSHBUTTON	*	Creates a push button that behaves like a BS_PUSHBUTTON style button, but has a distinct appearance. If the button is in a dialog box, the user can select the button by pressing the ENTER key, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely (default) option.

BS_GROUPBOX	*	Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper left corner.
BS_LEFTTEXT	*	Places text on the left side of the radio button or check box when combined with a radio button or check box style. Same as the BS_RIGHTBUTTON style.
BS_OWNERDRAW	*	Creates an owner-drawn button. The owner window receives a WM_DRAWITEM message when a visual aspect of the button has changed. Do not combine the BS_OWNERDRAW style with any other button styles.
BS_PUSHBUTTON	*	Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button.
BS_RADIOBUTTON	*	Creates a small circle with text. By default, the text is displayed to the right of the circle. To display the text to the left of the circle, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style). Use radio buttons for groups of related, but mutually exclusive choices.
BS_USERBUTTON		Obsolete, but provided for compatibility with 16-bit versions of Windows. Applications should use BS_OWNERDRAW instead.
BS_BITMAP		Specifies that the button displays a bitmap. See the Remarks section for its interaction with BS_ICON.
BS_BOTTOM		Places text at the bottom of the button rectangle.
BS_CENTER		Centers text horizontally in the button rectangle.
BS_ICON		Specifies that the button displays an icon. See the Remarks section for its interaction with BS_BITMAP.
BS_FLAT		Specifies that the button is two-dimensional; it does not use the default shading to create a 3-D image.
BS_LEFT		Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button.
BS_MULTILINE		Wraps the button text to multiple lines if the text string is too long to fit on a single line in the button rectangle.

BS_NOTIFY	Enables a button to send BN_KILLFOCUS and BN_SETFOCUS notification codes to its parent window. Note that buttons send the BN_CLICKED notification code regardless of whether it has this style. To get BN_DBLCLK notification codes, the button must have the BS_RADIOBUTTON or BS_OWNERDRAW style.
BS_PUSHLIKE	Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.
BS_RIGHT	Right-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is right justified on the right side of the check box or radio button.
BS_RIGHTBUTTON	Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the BS_LEFTTEXT style.
BS_TEXT	Specifies that the button displays text.
BS_VCENTER	Places text in the middle (vertically) of the button rectangle.

We can dynamically change some of the control style by calling **SetWindowLong** function.

SetWindowLong

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

```
LONG WINAPI SetWindowLong(
        HWND hWnd,
        int nIndex,
        LONG dwNewLong
);
```

Parameters

hWnd - A handle to the window and, indirectly, the class to which the window belongs.

nIndex - The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of an integer. To set any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE	Sets a new extended window style.
GWL_HINSTANCE	Sets a new application instance handle.
GWL_ID	Sets a new identifier of the child window. The window cannot be a top-level window.
GWL_STYLE	Sets a new window style.
GWL_USERDATA	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
GWL_WNDPROC	Sets a new address for the window procedure.

You cannot change this attribute if the window does not belong to the same process as the calling thread.

dwNewLong - The replacement value.

Return Value

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the function fails, the return value is zero.

Following are the button control styles which can be changed dynamically:

- **BS_BITMAP**
- **BS_BOTTOM**
- **BS_CENTER**
- **BS_ICON**
- **BS_LEFT**
- **BS_TEXT**
- **BS_NOTIFY**
- **BS_MULTILINE**
- **BS_RIGHT**
- **BS_TOP**

- **BS_VCENTER**
- **WS_DISABLED**
- **WS_VISIBLE**

A button is modified by sending messages be calling SendMessage() or PostMessage() function. Button query messages is shown below, not all messages apply to all button styles; the applicability of message to a particular button depends upon its style.

Message	Description
BM_CLICK	Simulates the user clicking a button. This message causes the button to receive the WM_LBUTTONDOWN and WM_LBUTTONUP messages, and the button's parent window to receive a BN_CLICKED notification code.
BM_GETCHECK	Gets the check state of a radio button or check box. You can send this message explicitly or use the Button_GetCheck macro.
BM_GETIMAGE	Retrieves a handle to the image (icon or bitmap) associated with the button.
BM_GETSTATE	Retrieves the state of a button or check box. You can send this message explicitly or use the Button_GetState macro.
BM_SETCHECK	Sets the check state of a radio button or check box. You can send this message explicitly or by using the Button_SetCheck macro.
BM_SETIMAGE	Associates a new image (icon or bitmap) with the button.
BM_SETSTATE	Sets the highlight state of a button. The highlight state indicates whether the button is highlighted as if the user had pushed it. You can send this message explicitly or use the Button_SetState macro.
BM_SETSTYLE	Sets the style of a button. You can send this message explicitly or use the Button_SetStyle macro.
BM_SETSTYLE	Sets the style of a button. You can send this message explicitly or use the Button_SetStyle macro.
WM_ENABLE	Sent when an application changes the enabled state of a window. It is sent to the window whose enabled state is changing. This message is sent before the EnableWindow function returns, but after the enabled state (WS_DISABLED style bit) of the window has changed.A window receives this message through its WindowProc function.

WM_GETTEXT	Copies the text that corresponds to a window into a buffer provided by the caller.
WM_SETTEXT	Sets the text of a window.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

The label of button control can be changed by calling the **SetWindowText** function.

SetWindowText

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application.

```
BOOL WINAPI SetWindowText(
        HWND hWnd,
        LPCTSTR lpString
);
```

Parameters

hWnd - A handle to the window or control whose text is to be changed.

lpString - The new title or control text.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero

GetWindowText

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application.

```
int WINAPI GetWindowText(
        HWND hWnd,
        LPTSTR lpString,
        int nMaxCount
);
```

Parameters

hWnd - A handle to the window or control containing the text.

lpString - The buffer that will receive the text. If the string is as long or longer than the buffer, the string is truncated and terminated with a null character.

nMaxCount - The maximum number of characters to copy to the buffer, including the null character. If the text exceeds this limit, it is truncated.

Return Value

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character. If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero.

This function cannot retrieve the text of an edit control in another application.

Button Notification messages

Notification Code	Meaning
BN_CLICKED	Sent when the user clicks a button. The parent window of the button receives this notification code through the WM_COMMAND message.
BN_DISABLE	Sent when a button is disabled.
BN_DBLCLK	Sent when the user double-clicks a button. This notification code is sent automatically for BS_USERBUTTON, BS_RADIOBUTTON, and BS_OWNERDRAW buttons. Other button types send BN_DBLCLK only if they have the BS_NOTIFY style. The parent window of the button receives this notification code through the WM_COMMAND message.
BN_HILITE	Sent when the user selects a button. This notification code is provided only for compatibility with 16-bit versions of Windows earlier than version 3.0
BN_PAINT	Sent when a button should be painted. **Note**: This notification code is provided only for compatibility with 16-bit versions of Windows earlier than version 3.0.
BN_UNHILITE	Sent when the highlight should be removed from a button. **Note:** This notification code is provided only for compatibility with 16-bit versions of Windows earlier than version 3.0.

BN_SETFOCUS	Sent when a button receives the keyboard focus. The button must have the BS_NOTIFY style to send this notification code. The parent window of the button receives this notification code through the WM_COMMAND message.
BN_KILLFOCUS	Sent when a button loses the keyboard focus. The button must have the BS_NOTIFY style to send this notification code. The parent window of the button receives this notification code through the WM_COMMAND message.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

Push Buttons

Push button is used to trigger an immediate action. Each time the user clicks the push button, the button send a WM_COMMAND message with a notification code of BN_CLICKED to its parent window. A push button does not have multiple states as do other kinds of button. A push button can only be pushed; it can not be checked or grayed

A Default Push Button works the same as a regular push button when used in the client area of a window.

Push Button Styles

Styles	Meaning
Vertical Style (Any One)	
BS_BOTTOM	The text is bottom aligned in the control rectangle.
BS_TOP	The text is top aligned in the control rectangle.
BS_VCENTER	The text is vertically center aligned in the control rectangle.
Horizontal Style (Any one)	
BS_CENTER	The text is horizontally center aligned in the control rectangle.
BS_LEFT	The text is left justified in the control rectangle.
BS_RIGHT	The text is right justified in the control rectangle.

Additional Styles	
BS_MULTILINE	Text will be wrapped onto multiple lines if it is too wide.

Note: Before creation of the control, define the ID for each control you use. It can be done by #define statement either on the top of the soruce file or in seperade *.h files. We need to do this because that is used as one of the parameter in CreateWindow() function.Windows returns the ID of control in the LOWORD of wParam parameter of windows callback function.

Creating Button

The following example shows how to use the CreateWindow function for creating a default push button:

```
#define IDC_OK 101
HWND hwndDefButton;
hwndDefButton=CreateWindow(
                    "BUTTON",          // Button Class
                    "OK",              // Button Caption
                    WS_VISIBLE |       // Make button visible
                    WS_CHILD |         // Controls are child
                    BS_DEFPUSBUTTON,   // Default push button
                    10,                // Starting x-coordinate
                    10,                // Starting y - coordinate
                    100,               // Button Width
                    100,               // Button Height
                    hwnd,              // Parent Window Handle
                    (HMENU)IDC_OK,     // Control ID
                    hInstance,         // Instance handle
                    NULL);             // No Additional Data
```

The following code simulates the appearance of click on OK push button created earlier:

```
SendMessage(pushbutton, BM_SETSTATE, TRUE, 0);
```

The BM_SETSTATE message controls only the appearance of the exterior of a button. It has no effect on the check state of a radio button or check box, because this message controls only the appearance of the button being clicked. It doesn't actually cause the button to behave as if it had be clicked. To simulate the actual clicking of the pushbutton we need to use BM_CLICK message or its corresponding Button_Click function.

SendMessage

Sends the specified message to a window or windows. The SendMessage function calls the window procedure for the specified window and does not return until the window procedure has processed the message.

```
LRESULT WINAPI SendMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

Parameters

hWnd - A handle to the window whose window procedure will receive the message. If this parameter is HWND_BROADCAST ((HWND)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Msg - The message to be sent.

wParam - Additional message-specific information.

lParam - Additional message-specific information.

Return Value

The return value specifies the result of the message processing; it depends on the message sent.

Check Boxes

A check is a special case of the BUTTON class. Check box can be used to allow a user to select options from the applications. A check box displays as a square box with a label to one side. The default is to show the label to the right of the textbox. When BS_LEFTTEXT button style is used along with the check box button style, the button displays the label to the left of the square box.

*To crate the Checkbox use **CreateWindow** function with either of the following 'Checkbox Styles':*

- **BS_CHECKBOX**
- **BS_AUTOCECKBOX**
- **BS_3STATE**
- **BS_AUTO3STATE**

The BS_CHECKBOX style creates a checkbox that support two states either checked or unchecked. The BS_AUTOCHECK style creates a check box that automatically toggles between those 2 states. The BS_3STATE style creates a check box that supports 3 states; checked, indeterminate and unchecked and the BS_AUTO3STATE creates a checkbox that automatically toggles among three states.

A check box control sends its parent window WM_COMMAND message with a notification code of BN_CLICKED when the user clicks the control or presses the space bar when the control has input focus. The BS_CHECKBOX and BS_3STATE style do not change the appearance as the user clicks the checkbox. Only the auto style automatically changes the appearance.

If BS_NOTIFY style is specified, additional notifications BST_UNCHECKED, BST_CHECKED or BST_INDETERMINATE can be retrived.

When you want the appearance of the BS_CECKBOX and BS_3STATE style controls to reflect their current state, you must send a BM_SETCHECK message to the control. The BM_SETCHECK message tells the control to display the checkbox as unchecked, checked or indeterminate.

```
int prev= (int)SendMessage(hRbutton, WM_SETCHECK, checkstate, 0);
                                    OR
int prev= Button_SetCheck(hRbutton, checkstate);
```

```
#define IDC_DIRECTION       102
HWND hwndCheckBox;
```

```
hwndCheckBox=CreateWindow(
                        "BUTTON",
                        "Left",
                        WS_VISIBLE |
                        WS_CHILD |
                        BS_3STATE,
                        10,
                        10,
                        100,
                        100,
                        hwnd,
                        (HMENU)IDC_DIRECTION,
                        hInstance,
                        NULL);

SendMessage(hwndCheckBox, BM_SETCHECK, BST_INDETERMINATE, 0);
                                    OR
Button_SetCheck(hwndCheckBox, BST_INDETERMINATE);
```

Radio Button

Radio buttons are another special case of the BUTTON class. Radio button controls are quite similar to checkbox controls except that radio button is circular, whereas a check box is square. A radio button is checked by displaying a solid dot in the center of the circle. You typically create radio buttons to allow the user to select one item from the collection of mutually exclusive choices.

```
#define     IDC_BACK        103
#define     IDC_FORWARD     104

HWND hwndBack;
HWND hwndForward;

hwndBack=CreateWindow("BUTTON", "Back", WS_CHILD | WS_VISIBLE | WS_TABSTOP |
WS_GROUP | WS_AUTORADIOBUTTON, 100, 200, 70, 15, hwnd, (HMENU)IDC_BACK, hInstance,
NULL);
```

hwndForward=CreateWindow("BUTTON", "Forward", WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_GROUP | WS_AUTORADIOBUTTON, 100, 220, 70, 15, hwnd, (HMENU)IDC_FORWARD, hInstance, NULL);

Like the check box a radio button also sends WM_COMMAND button to its parent with a notification code of BN_CLICKED. For a BS_RADIOBUTTON style you must send a BM_SETCHECK message to request it to display the check mark which is a dot.

SendMessage(hwndBack, BM_SETCHECK, BST_CHECKED,0);

Because each radio buttons in a group represents a mutually exclusive option you should also send BM_SETCHECK messages to all other radio buttons in the group to ensure that their check marks are off.

SendMessage(hwndForward, BM_SETCHECK, BST_UNCHECKED, 0)

OR

Button_SetCheck(hwndForward, BST_UNCHECKED)

Radio button with BS_AUTORADIOBUTTON style for radio buttons is used in a dialog box. When all the radio buttons in a group within a dialog box have BS_AUTORADIOBUTTON windows automaticallu uncheck previously checked radio button. Windows determines what a group is by searching through the controls looking for a radio button with WS_GROUP flag set.

Windows will turn off all radio buttons in the group except the one that was clicked, which it turns on. By convention a group box often follows the set of radio buttons and is of a size to enclose the radio buttons.

EDIT Class

An edit control is a child window tha accepts keyboard input. The user can enter and edit one or more lines of text. You can allow the text to word wrap automatically to the textline or use an edit control that automatically scrolls the text – horizontally or vertically. You select the desired features of an edit control by combining edit class styles with the bit-wise OR operator. There are two kinds of edit controls in win32, the ordinary edit control and the Richtext edit control.

An edit control is a rectangular control window typically used in a dialog box to permit the user to enter and edit text by typing on the keyboard. Edit controls support both the Unicode as well ASCII & ANSI charater sets.

Edit Control Styles

To create an edit control using the CreateWindow or CreateWindowEx function, specify the EDIT class, appropriate window style constants, and a combination of the following edit control styles. After the control has been created, these styles cannot be modified, except as noted.

Constant	Description
ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.
ES_AUTOVSCROLL	Automatically scrolls text up one page when the user presses the ENTER key on the last line.
ES_CENTER	Windows 98/Me, Windows 2000/XP: Centers text in a single-line or multiline edit control. *Windows 95, Windows NT 4.0 and earlier: Centers text in a multiline edit control.*
ES_LEFT	Aligns text with the left margin.
ES_LOWERCASE	Converts all characters to lowercase as they are typed into the edit control. To change this style after the control has been created, use **SetWindowLong**.
ES_MULTILINE	Designates a multiline edit control. The default is single-line edit control. When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the ES_WANTRETURN style. When the multiline edit control is not in a dialog box and the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify ES_AUTOVSCROLL, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed. If you specify the ES_AUTOHSCROLL style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify ES_AUTOHSCROLL, the control automatically wraps words to the beginning of the next line when necessary. A new line is

also started if the user presses the ENTER key. The window size determines the position of the Wordwrap. If the window size changes, the Wordwrapping position changes and the text is redisplayed.

Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

ES_NOHIDESEL	Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus.
ES_NUMBER	Allows only digits to be entered into the edit control. Note that, even with this set, it is still possible to paste non-digits into the edit control. To change this style after the control has been created, use **SetWindowLong**.
ES_OEMCONVERT	Converts text entered in the edit control. The text is converted from the Windows character set to the OEM character set and then back to the Windows character set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the edit control to OEM characters. This style is most useful for edit controls that contain file names that will be used on file systems that do not support Unicode. To change this style after the control has been created, use **SetWindowLong**.
ES_PASSWORD	Displays an asterisk (*) for each character typed into the edit control. This style is valid only for single-line edit controls. An application can use **SetPasswordChar** member function to change the password character. Windows XP: If the edit control is from user32.dll, the default password character is an asterisk. However, if the edit control is from comctl32.dll version 6, the default character is a black circle. To change the characters that is displayed, or set or clear this style, use the EM_SETPASSWORDCHAR message.

ES_READONLY	Prevents the user from typing or editing text in the edit control. To change this style after the control has been created, use the EM_SETREADONLY message. To change this style after the control has been created, use **Edit_SetReadOnly()**.
ES_RIGHT	Windows 98/Me, Windows 2000/XP: Right-aligns text in a single-line or multiline edit control. Windows 95, Windows NT 4.0 and earlier: Right aligns text in a multiline edit control.
ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control. To change this style after the control has been created, use **SetWindowLong**.
ES_WANTRETURN	Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control. To change this style after the control has been created, use **SetWindowLong**.

SetWindowLong

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

```
LONG WINAPI SetWindowLong(
        HWND hWnd,
        int nIndex,
        LONG dwNewLong
);
```

Parameters

hWnd- A handle to the window and, indirectly, the class to which the window belongs.

nIndex - The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of an integer.

dwNewLong - The replacement value.

Return Value

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the function fails, the return value is zero.

ES_PASSWORD style is used for edit control when you don't want the control to display the characters when the user is typing. An edit control wint ES_PASSWORD style displays (*) by default in place of types texts. I you application require to show other character then (*) you can send ES_SETPASSWORDCHAR message with desired character to show. You can also call Edit_SetPasswordChar macro to set the password character of an edit control.

Edit_SetPasswordChar(hwndEdit, IDC_EDIT, "#");

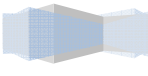
Edit Control Messages

Messages	Description
EM_CANUNDO	Determines whether there are any actions in an edit control's undo queue. You can send this message to either an edit control or a rich edit control.
EM_CHARFROMPOS	Gets information about the character closest to a specified point in the client area of an edit control. You can send this message to either an edit control or a rich edit control.
EM_EMPTYUNDOBUFFER	Resets the undo flag of an edit control. The undo flag is set whenever an operation within the edit control can be undone. You can send this message to either an edit control or a rich edit control.
EM_FMTLINES	Sets a flag that determines whether a multiline edit control includes soft line-break characters. A soft line break consists of two carriage returns and a line feed and is inserted at the end of a line that is broken because of wordwrapping.
EM_GETCUEBANNER	Gets the text that is displayed as the textual cue, or tip, in an edit control.
EM_GETFIRSTVISIBLELINE	Gets the zero-based index of the uppermost visible line in a multiline edit control. You can send this message to either an edit control or a rich edit control.
EM_GETHANDLE	Gets a handle of the memory currently allocated for a multiline edit control's text.
EM_GETHILITE	This message is not implemented.
EM_GETIMESTATUS	Gets a set of status flags that indicate how the edit control interacts with the Input Method Editor (IME).

EM_GETLIMITTEXT	Gets the current text limit for an edit control. You can send this message to either an edit control or a rich edit control.
EM_GETLINE	Copies a line of text from an edit control and places it in a specified buffer. You can send this message to either an edit control or a rich edit control.
EM_GETLINECOUNT	Gets the number of lines in a multiline edit control. You can send this message to either an edit control or a rich edit control.
EM_GETMARGINS	Gets the widths of the left and right margins for an edit control.
EM_GETMODIFY	Gets the state of an edit control's modification flag. The flag indicates whether the contents of the edit control have been modified. You can send this message to either an edit control or a rich edit control.
EM_GETPASSWORDCHAR	Gets the password character that an edit control displays when the user enters text. You can send this message to either an edit control or a rich edit control.
EM_GETRECT	Gets the formatting rectangle of an edit control. The formatting rectangle is the limiting rectangle into which the control draws the text. The limiting rectangle is independent of the size of the edit-control window. You can send this message to either an edit control or a rich edit control.
EM_GETSEL	Gets the starting and ending character positions (in TCHARs) of the current selection in an edit control. You can send this message to either an edit control or a rich edit control.
EM_GETTHUMB	Gets the position of the scroll box (thumb) in the vertical scroll bar of a multiline edit control. You can send this message to either an edit control or a rich edit control.
EM_GETWORDBREAKPROC	Gets the address of the current Wordwrap function. You can send this message to either an edit control or a rich edit control.
EM_HIDEBALLOONTIP	Hides any balloon tip associated with an edit control.
EM_LIMITTEXT	Sets the text limit of an edit control. The text limit is the maximum amount of text, in TCHARs, that the user can type into the edit control. You can send this message to either an edit control or a rich edit control.
EM_LINEFROMCHAR	Gets the index of the line that contains the specified character

	index in a multiline edit control. A character index is the zero-based index of the character from the beginning of the edit control. You can send this message to either an edit control or a rich edit control.
EM_LINEINDEX	Gets the character index of the first character of a specified line in a multiline edit control. A character index is the zero-based index of the character from the beginning of the edit control. You can send this message to either an edit control or a rich edit control.
EM_LINELENGTH	Retrieves the length, in characters, of a line in an edit control. You can send this message to either an edit control or a rich edit control.
EM_LINESCROLL	Scrolls the text in a multiline edit control.
EM_NOSETFOCUS	Intended for internal use; not recommended for use in applications. Prevents a single-line edit control from receiving keyboard focus. You can send this message explicitly or by using the Edit_NoSetFocus macro.
EM_POSFROMCHAR	Retrieves the client area coordinates of a specified character in an edit control. You can send this message to either an edit control or a rich edit control.
EM_REPLACESEL	Replaces the selected text in an edit control or a rich edit control with the specified text.
EM_SCROLL	Scrolls the text vertically in a multiline edit control. This message is equivalent to sending a WM_VSCROLL message to the edit control. You can send this message to either an edit control or a rich edit control.
EM_SCROLLCARET	Scrolls the caret into view in an edit control. You can send this message to either an edit control or a rich edit control.
EM_SETCUEBANNER	Sets the textual cue, or tip, that is displayed by the edit control to prompt the user for information.
EM_SETHANDLE	Sets the handle of the memory that will be used by a multiline edit control.
EM_SETHILITE	This message is not implemented.

EM_SETIMESTATUS	Sets the status flags that determine how an edit control interacts with the Input Method Editor (IME).
EM_SETLIMITTEXT	Sets the text limit of an edit control. The text limit is the maximum amount of text, in TCHARs, that the user can type into the edit control. You can send this message to either an edit control or a rich edit control.
EM_SETMARGINS	Sets the widths of the left and right margins for an edit control. The message redraws the control to reflect the new margins. You can send this message to either an edit control or a rich edit control.
EM_SETMODIFY	Sets or clears the modification flag for an edit control. The modification flag indicates whether the text within the edit control has been modified. You can send this message to either an edit control or a rich edit control.
EM_SETPASSWORDCHAR	Sets or removes the password character for an edit control. When a password character is set, that character is displayed in place of the characters typed by the user. You can send this message to either an edit control or a rich edit control.
EM_SETREADONLY	Sets or removes the read-only style (ES_READONLY) of an edit control. You can send this message to either an edit control or a rich edit control.
EM_SETRECT	Sets the formatting rectangle of a multiline edit control. The formatting rectangle is the limiting rectangle into which the control draws the text. The limiting rectangle is independent of the size of the edit control window. This message is processed only by multiline edit controls. You can send this message to either an edit control or a rich edit control.
EM_SETRECTNP	Sets the formatting rectangle of a multiline edit control. The EM_SETRECTNP message is identical to the EM_SETRECT message, except that EM_SETRECTNP does not redraw the edit control window. The formatting rectangle is the limiting rectangle into which the control draws the text. The limiting rectangle is independent of



	the size of the edit control window. This message is processed only by multiline edit controls. You can send this message to either an edit control or a rich edit control.
EM_SETSEL	Selects a range of characters in an edit control. You can send this message to either an edit control or a rich edit control.
EM_SETTABSTOPS	The EM_SETTABSTOPS message sets the tab stops in a multiline edit control. When text is copied to the control, any tab character in the text causes space to be generated up to the next tab stop. This message is processed only by multiline edit controls. You can send this message to either an edit control or a rich edit control.
EM_SETWORDBREAKPROC	Replaces an edit control's default Wordwrap function with an application-defined Wordwrap function. You can send this message to either an edit control or a rich edit control.
EM_SHOWBALLOONTIP	The EM_SHOWBALLOONTIP message displays a balloon tip associated with an edit control.
EM_TAKEFOCUS	Intended for internal use; not recommended for use in applications. Forces a single-line edit control to receive keyboard focus. You can send this message explicitly or by using the Edit_TakeFocus macro.
EM_UNDO	This message undoes the last edit control operation in the control's undo queue. You can send this message to either an edit control or a rich edit control.
WM_UNDO	An application sends a WM_UNDO message to an edit control to undo the last operation. When this message is sent to an edit control, the previously deleted text is restored or the previously added text is deleted.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

Edit Control Notification Messages

Notification Code	Meaning
EN_CHANGE	Sent when the user has taken an action that may have altered text in an edit control. Unlike the EN_UPDATE notification code, this

	notification code is sent after the system updates the screen. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_ERRSPACE	Sent when an edit control cannot allocate enough memory to meet a specific request. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_HSCROLL	Sent when the user clicks an edit control's horizontal scroll bar. The parent window of the edit control receives this notification code through a WM_COMMAND message. The parent window is notified before the screen is updated.
EN_KILLFOCUS	Sent when an edit control loses the keyboard focus. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_MAXTEXT	Sent when the current text insertion has exceeded the specified number of characters for the edit control. The text insertion has been truncated. This notification code is also sent when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted would exceed the width of the edit control. This notification code is also sent when an edit control does not have the ES_AUTOVSCROLL style and the total number of lines resulting from a text insertion would exceed the height of the edit control. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_SETFOCUS	Sent when an edit control receives the keyboard focus. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_UPDATE	Sent when an edit control is about to redraw itself. This notification code is sent after the control has formatted the text, but before it displays the text. This makes it possible to resize the edit control window, if necessary. The parent window of the edit control receives this notification code through a WM_COMMAND message.
EN_VSCROLL	Sent when the user clicks an edit control's vertical scroll bar or when the user scrolls the mouse wheel over the edit control. The

	parent window of the edit control receives this notification code through a WM_COMMAND message. The parent window is notified before the screen is updated.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

Working with selection in edit control

The text in an edit control may be selected and selection is indeicated by highlighting color over t he selection character. The edit control handles the details of responding to the mouse button down, dragging and mouse button release, as well as the painting of the selection highlight. In addition it supports the standard windows keyboard shortcuts for selection as shown below:

Short cut key	Effect
CTRL + C	Copies the selection to the clipboard in CF_TEXT format.
CTRL + V	Pastes the clipboard contents, replacing the selection if there is any. If no data in clipboard then, no effect.
CTRL + X	Cuts the selection to the clipboard in CF_TEXT format.
←	Moves the caret left.
→	Moves the caret right.
↓	In Single line: NO EFFECT. In Multi line: Moves up one line.
↑	In Single line: NO EFFECT. In Multi line: Moves down one line.
CTRL + ←	Moves the caret left one word.
CTRL + →	Moves the caret right one word.
SHIFT + ←	Moves the selection end at the caret to the left by one position.
SHIFT + →	Moves the selection end at the caret to the right by one position.
SHIFT + ↑	Moves the caret up one line carrying the selection end with it.
SHIFT + ↓	Moves the caret down one line carrying the selection end with it.
CTRL + SHIFT + ←	Moves the selection end at the caret to the left by on word.

CTRL + SHIFT + ⟶	Moves the selection end at the caret to the right by on word.
DELETE	If there is selection, it removes the contents of the selection. Otherwise deletes text to the right of the caret position.
CTRL + DELETE	If there is selection, it removes the contents of the selection. Otherwise deletes text from the caret position to the end of the line.
BACKSPACE	If there is selection, it removes the contents of the selection. Otherwise deletes text to the left of the caret position.

List box

A list box is a control window that contains a list of items from which the user can choose. A list box can display its contents in a single column or in multiple columns. When a single column list box contains more items than can be displayed in the control's client area, you can add a vertical scroll bar to scroll the other items into view. A list box normally allows the users to select only one item. However list box that allows multiple selections can also be created.

If the list box is not large enough to display all the list box items at once, the listbox provides a scroll bar. The user scrolls through the list box items and applies or removes selection status as necessary. Selecting a list box item changes its visual appearance usually by changing the text and background colors to those specified by the relevant operating system metrics. When the user selects or deselects an item, the system sends a motification message to the parent window of the list box.

E.g.

```
#define IDC_LISTBOX        105
HWND hwndListBox;

hwndListBox = CreateWindow("LISTBOX", NULL, WS_CHILD | WS_VISIBLE | LBS_STANDARD, 100, 200, 300, 60, hwnd, IDC_LISTBOX, hInstance, NULL);
```

This **creates** a single column list box with a border around it. The control sends notification messages to its parent window when the user selects an item in the list box.

This list box keeps its contents stored alphabetically. The list box also has a vertical scroll bar. The feature is implemented by specifying the LBS_STANDARD list box class style.

List Box Class Styles

To create a list box by using the CreateWindow or CreateWindowEx function, use the LISTBOX class, appropriate window style constants, and the following style constants to define the list box.

Constant	Description
LBS_COMBOBOX	Notifies a list box that it is part of a combo box. This allows coordination between the two controls so that they present a unified UI. The combo box itself must set this style. If the style is set by anything but the combo box, the list box will regard itself incorrectly as a child of a combo box and a failure will result.
LBS_DISABLENOSCROLL	Shows a disabled horizontal or vertical scroll bar when the list box does not contain enough items to scroll. If you do not specify this style, the scroll bar is hidden when the list box does not contain enough items. This style must be used with the WS_VSCROLL or WS_HSCROLL style.
LBS_EXTENDEDSEL	Allows multiple items to be selected by using the SHIFT key and the mouse or special key combinations.
LBS_HASSTRINGS	Specifies that a list box contains items consisting of strings. The list box maintains the memory and addresses for the strings so that the application can use the LB_GETTEXT message to retrieve the text for a particular item. By default, all list boxes except owner-drawn list boxes have this style. You can create an owner-drawn list box either with or without this style.
LBS_MULTICOLUMN	Specifies a multi-column list box that is scrolled horizontally. The list box automatically calculates the width of the columns, or an application can set the width by using the LB_SETCOLUMNWIDTH message. If a list box

has the LBS_OWNERDRAWFIXED style, an application can set the width when the list box sends the WM_MEASUREITEM message. A list box with the LBS_MULTICOLUMN style cannot scroll vertically—it ignores any WM_VSCROLL messages it receives.

The LBS_MULTICOLUMN and LBS_OWNERDRAWVARIABLE styles cannot be combined. If both are specified, LBS_OWNERDRAWVARIABLE is ignored.

LBS_MULTIPLESEL Turns string selection on or off each time the user clicks or double-clicks a string in the list box. The user can select any number of strings.

LBS_NOINTEGRALHEIGHT Specifies that the size of the list box is exactly the size specified by the application when it created the list box. Normally, the system sizes a list box so that the list box does not display partial items. For list boxes with the LBS_OWNERDRAWVARIABLE style, the LBS_NOINTEGRALHEIGHT style is always enforced.

LBS_NOREDRAW Specifies that the list box's appearance is not updated when changes are made. To change the redraw state of the control, use the WM_SETREDRAW message.

LBS_NOSEL Specifies that the list box contains items that can be viewed but not selected.

LBS_NOTIFY Causes the list box to send a notification code to the parent window whenever the user clicks a list box item (LBN_SELCHANGE), double-clicks an item (LBN_DBLCLK), or cancels the selection (LBN_SELCANCEL).

LBS_OWNERDRAWFIXED Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are the same height. The owner window receives a WM_MEASUREITEM message when the list box is created and a WM_DRAWITEM message when a visual aspect of the list box has changed.

LBS_OWNERDRAWVARIABLE Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a WM_MEASUREITEM message for each item in the box when the list box is created and a WM_DRAWITEM message when a visual aspect of the list box has changed. This style causes the LBS_NOINTEGRALHEIGHT style to be enabled. This style is ignored if the LBS_MULTICOLUMN style is specified.

LBS_SORT Sorts strings in the list box alphabetically.

LBS_STANDARD Sorts strings in the list box alphabetically. The parent window receives a notification code whenever the user clicks a list box item, double-clicks an item, or or cancels the selection. The list box has a vertical scroll bar, and it has borders on all sides. This style combines the LBS_NOTIFY, LBS_SORT, WS_VSCROLL, and WS_BORDER styles.

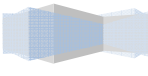
LBS_USETABSTOPS Enables a list box to recognize and expand tab characters when drawing its strings. You can use the LB_SETTABSTOPS message to specify tab stop positions. The default tab positions are 32 dialog template units apart. Dialog template units are the device-independent units used in dialog box templates. To convert measurements from dialog template units to screen units (pixels), use the MapDialogRect function.

LBS_WANTKEYBOARDINPUT Specifies that the owner of the list box receives WM_VKEYTOITEM messages whenever the user presses a key and the list box has the input focus. This enables an application to perform special processing on the keyboard input.

Messages to a List Box Control

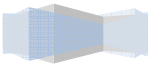
Message	Description
LB_ADDFILE	Adds the specified filename to a list box that contains a directory listing.
LB_ADDSTRING	Adds a string to a list box. If the list box does not have

	the LBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list and the list is sorted.
LB_DELETESTRING	Deletes a string in a list box.
LB_DIR	Adds names to the list displayed by a list box. The message adds the names of directories and files that match a specified string and set of file attributes. LB_DIR can also add mapped drive letters to the list box.
LB_FINDSTRING	Finds the first string in a list box that begins with the specified string.
LB_FINDSTRINGEXACT	Finds the first list box string that exactly matches the specified string, except that the search is not case sensitive.
LB_GETANCHORINDEX	Gets the index of the anchor item—that is, the item from which a multiple selection starts. A multiple selection spans all items from the anchor item to the caret item.
LB_GETCARETINDEX	Retrieves the index of the item that has the focus in a multiple-selection list box. The item may or may not be selected.
LB_GETCOUNT	Gets the number of items in a list box.
LB_GETCURSEL	Gets the index of the currently selected item, if any, in a single-selection list box.
LB_GETHORIZONTALEXTENT	Gets the width, in pixels, that a list box can be scrolled horizontally (the scrollable width) if the list box has a horizontal scroll bar.
LB_GETITEMDATA	Gets the application-defined value associated with the specified list box item.
LB_GETITEMHEIGHT	Gets the height of items in a list box.
LB_GETITEMRECT	Gets the dimensions of the rectangle that bounds a



	list box item as it is currently displayed in the list box.
LB_GETLISTBOXINFO	Gets the number of items per column in a specified list box.
LB_GETLOCALE	Gets the current locale of the list box. You can use the locale to determine the correct sorting order of displayed text (for list boxes with the LBS_SORT style) and of text added by the LB_ADDSTRING message.
LB_GETSEL	Gets the selection state of an item.
LB_GETSELCOUNT	Gets the total number of selected items in a multiple-selection list box.
LB_GETSELITEMS	Fills a buffer with an array of integers that specify the item numbers of selected items in a multiple-selection list box.
LB_GETTEXT	Gets a string from a list box.
LB_GETTEXTLEN	Gets the length of a string in a list box.
LB_GETTOPINDEX	Gets the index of the first visible item in a list box. Initially the item with index 0 is at the top of the list box, but if the list box contents have been scrolled another item may be at the top. The first visible item in a multiple-column list box is the top-left item.
LB_INITSTORAGE	Allocates memory for storing list box items. This message is used before an application adds a large number of items to a list box.
LB_INSERTSTRING	Inserts a string or item data into a list box. Unlike the LB_ADDSTRING message, the LB_INSERTSTRING message does not cause a list with the LBS_SORT style to be sorted.
LB_ITEMFROMPOINT	Gets the zero-based index of the item nearest the specified point in a list box.
LB_RESETCONTENT	Removes all items from a list box.

LB_SELECTSTRING	Searches a list box for an item that begins with the characters in a specified string. If a matching item is found, the item is selected.
LB_SELITEMRANGE	Selects or deselects one or more consecutive items in a multiple-selection list box.
LB_SELITEMRANGEEX	Selects one or more consecutive items in a multiple-selection list box.
LB_SETANCHORINDEX	Sets the anchor item—that is, the item from which a multiple selection starts. A multiple selection spans all items from the anchor item to the caret item.
LB_SETCARETINDEX	Sets the focus rectangle to the item at the specified index in a multiple-selection list box. If the item is not visible, it is scrolled into view.
LB_SETCOLUMNWIDTH	Sets the width, in pixels, of all columns in a multiple-column list box.
LB_SETCOUNT	Sets the count of items in a list box created with the LBS_NODATA style and not created with the LBS_HASSTRINGS style.
LB_SETCURSEL	Selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string.
LB_SETHORIZONTALEXTENT	Sets the width, in pixels, by which a list box can be scrolled horizontally (the scrollable width). If the width of the list box is smaller than this value, the horizontal scroll bar horizontally scrolls items in the list box. If the width of the list box is equal to or greater than this value, the horizontal scroll bar is hidden.
LB_SETITEMDATA	Sets a value associated with the specified item in a list box.
LB_SETITEMHEIGHT	Sets the height, in pixels, of items in a list box. If the



	list box has the LBS_OWNERDRAWVARIABLE style, this message sets the height of the item specified by the wParam parameter. Otherwise, this message sets the height of all items in the list box.
LB_SETLOCALE	Sets the current locale of the list box. You can use the locale to determine the correct sorting order of displayed text (for list boxes with the LBS_SORT style) and of text added by the LB_ADDSTRING message.
LB_SETSEL	Selects an item in a multiple-selection list box and, if necessary, scrolls the item into view.
LB_SETTABSTOPS	Sets the tab-stop positions in a list box.
LB_SETTOPINDEX	Ensures that the specified item in a list box is visible.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

E.g.

Adding, Inserting and Deleting Items from a List Box

New stirng can be added to a list box control by sending LB_ADDSTRING message. The wParam parameter isn't used. The lParam parameter points to the string to be added.

```
SendMessage(hwndListBox, LB_ADDSTRING, 0, (LPARAM)(LPCSTR) String);
ListBox_Addstring(hwndListBox, String);
```

A list box add item to the end of the list. When the box has the LB_SORT style, the item is placed at the appropriate position in the list. You can also insert an item into a specified position in the list by sending an LB_INSERTSTRING message to the box. The box will place the new item in the position specified by the wParam parameter of the message.

```
SendMessage(hwndListBox, LB_INSERTSTRING, ItemIndex, (LPARAM)(LPCSTR)
            String);
```

Items can easily deleted from a list box by sending LB_DELETESTRING message:

```
SendMessage(hwndListBox, LB_DELETESTRING, ItemIndex, 0);
ListBox_DeleteString(hListBox, ItemIndex);
```

List Box Notifications

Notification	Description
LBN_DBLCLK	Notifies the application that the user has double-clicked an item in a list box. The parent window of the list box receives this notification code through the WM_COMMAND message.
LBN_ERRSPACE	Notifies the application that the list box cannot allocate enough memory to meet a specific request. The parent window of the list box receives this notification code through the WM_COMMAND message.
LBN_KILLFOCUS	Notifies the application that the list box has lost the keyboard focus. The parent window of the list box receives this notification code through the WM_COMMAND message.
LBN_SELCANCEL	Notifies the application that the user has canceled the selection in a list box. The parent window of the list box receives this notification code through the WM_COMMAND message.
LBN_SELCHANGE	Notifies the application that the selection in a list box has changed as a result of user input. The parent window of the list box receives this notification code through the WM_COMMAND message.
LBN_SETFOCUS	Notifies the application that the list box has received the keyboard focus. The parent window of the list box receives this notification code through the WM_COMMAND message.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.

Combo Box

A combo box combines an edit box or static text and a list. It's a unique type of control, defined by tge COMBOBOX class which combines the funcationality if Edit Control Class and List Box class.

E.g.

```
#define IDC_COMBO 106
HWND hwndCombo;
hwndCombo= CreateWindow("COMBOBOX", NULL, WS_CHILD | WS_VISIBLE | CBS_DROPDOWN, 100, 200, 300, 60, hwnd, IDC_COMBO, hInstance, NULL);
```

Combo Box Types and Styles

A combo box consists of a list and a selection field. The list presents the options that a user can select, and the selection field displays the current selection. If the selection field is an edit control, the user can enter information not available in the list; otherwise, the user can only select items in the list.

The common controls library includes three main styles of combo box, as shown in the following table.

Combo box type	Style constant	Description
Simple	**CBS_SIMPLE**	Displays the list at all times, and shows the selected item in an edit control.
Drop-down	**CBS_DROPDOWN**	Displays the list when the icon is clicked, and shows the selected item in an edit control.
Drop-down list (drop list)	**CBS_DROPDOWNLIST**	Displays the list when the icon is clicked, and shows the selected item in a static control.

The following screen shots each show the three kinds of combo box as they might appear in Windows Vista. In the first screen shot, the user has selected an item in the simple combo box. The user can also type a new value in the edit box of this control. The list has been sized in the Visual Studio resource editor and is only large enough to accommodate two items.

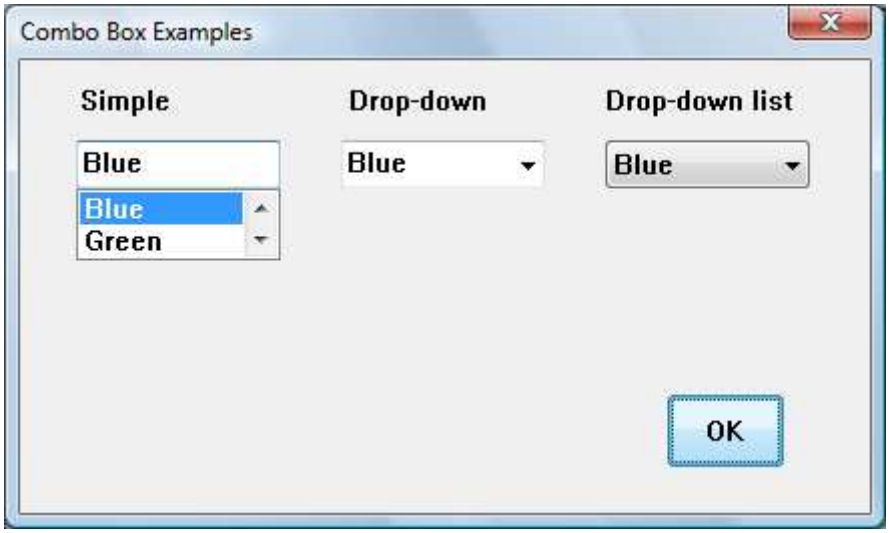


Figure: Screen shot showing an item selected in a simple combo box

In the second screen shot, the user has typed new text in the edit control of the drop-down combo box. The user could also have selected an existing item. The list box expands to accommodate as many items as possible.

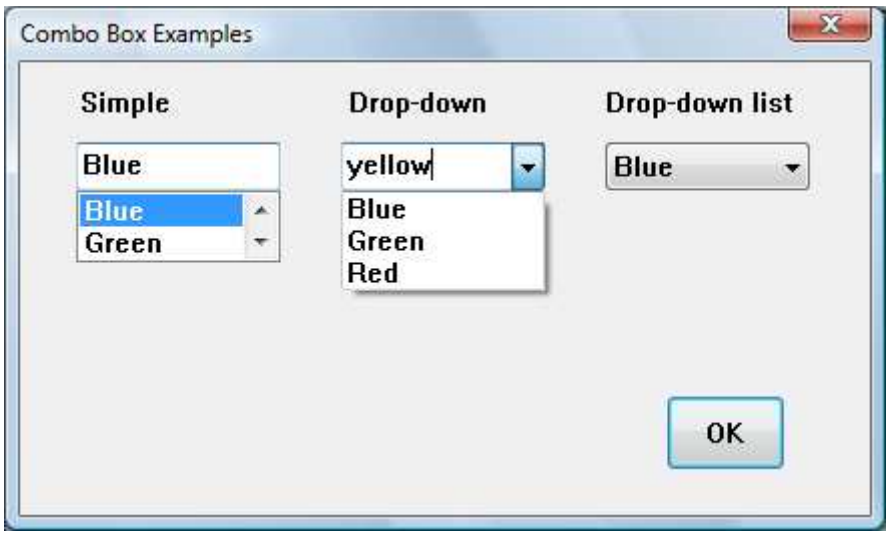


Figure: Screen shot showing text typed into a drop-down combo box

In the third screen shot, the user has opened the drop-down list combo box. The list box expands to accommodate as many items as possible. The user cannot enter new text.

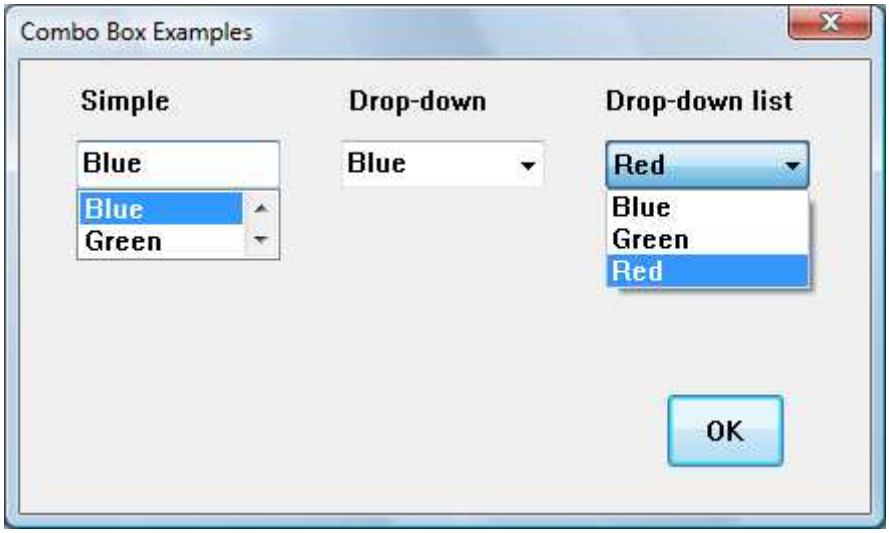


Figure: Screen shot showing an item selected in a drop-down list combo box

There are also a number of combo box styles that define specific properties. Combo box styles define specific properties of a combo box. You can combine styles; however, some styles apply only to certain combo box types.

Combo-Box Styles

CBS_AUTOHSCROLL Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.

CBS_DISABLENOSCROLL Shows a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.

CBS_DROPDOWN Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the edit control.

CBS_DROPDOWNLIST Similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item that displays the current selection in the list box.

CBS_HASSTRINGS Specifies that an owner-drawn combo box contains items consisting of strings. The combo box maintains the memory and address for the strings

so the application can use the CB_GETLBTEXT message to retrieve the text for a particular item.

CBS_LOWERCASE Converts to lowercase all text in both the selection field and the list.

CBS_NOINTEGRALHEIGHT Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, the system sizes a combo box so that it does not display partial items.

CBS_OEMCONVERT Converts text entered in the combo box edit control from the Windows character set to the OEM character set and then back to the Windows character set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the combo box to OEM characters. This style is most useful for combo boxes that contain file names and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN style.

CBS_OWNERDRAWFIXED Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are all the same height. The owner window receives a WM_MEASUREITEM message when the combo box is created and a WM_DRAWITEM message when a visual aspect of the combo box has changed.

CBS_OWNERDRAWVARIABLE Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a WM_MEASUREITEM message for each item in the combo box when you create the combo box and a WM_DRAWITEM message when a visual aspect of the combo box has changed.

CBS_SIMPLE Displays the list box at all times. The current selection in the list box is displayed in the edit control.

CBS_SORT Automatically sorts strings added to the list box.

CBS_UPPERCASE Converts to uppercase all text in both the selection field and the list.

Messages to a combobox Control

Message	Description
CB_ADDSTRING	Adds a string to the list box of a combo box. If the combo box does not have the CBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted.
CB_DELETESTRING	Deletes a string in the list box of a combo box.
CB_DIR	Adds names to the list displayed by the combo box. The message adds the names of directories and files that match a specified string and set of file attributes. CB_DIR can also add mapped drive letters to the list.
CB_FINDSTRING	Searches the list box of a combo box for an item beginning with the characters in a specified string.
CB_FINDSTRINGEXACT	Finds the first list box string in a combo box that matches the string specified in the lParam parameter.
CB_GETCOMBOBOXINFO	Gets information about the specified combo box.
CB_GETCOUNT	Gets the number of items in the list box of a combo box.
CB_GETCUEBANNER	Gets the cue banner text displayed in the edit control of a combo box. Send this message explicitly or by using the ComboBox_GetCueBannerText macro.
CB_GETCURSEL	An application sends a CB_GETCURSEL message to retrieve the index of the currently selected item, if any, in the list box of a combo box.
CB_GETDROPPEDCONTROLRECT	An application sends a CB_GETDROPPEDCONTROLRECT message to

	retrieve the screen coordinates of a combo box in its dropped-down state.
CB_GETDROPPEDSTATE	Determines whether the list box of a combo box is dropped down.
CB_GETDROPPEDWIDTH	Gets the minimum allowable width, in pixels, of the list box of a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.
CB_GETEDITSEL	Gets the starting and ending character positions of the current selection in the edit control of a combo box.
CB_GETEXTENDEDUI	Determines whether a combo box has the default user interface or the extended user interface.
CB_GETHORIZONTALEXTENT	Gets the width, in pixels, that the list box can be scrolled horizontally (the scrollable width). This is applicable only if the list box has a horizontal scroll bar.
CB_GETITEMDATA	An application sends a CB_GETITEMDATA message to a combo box to retrieve the application-supplied value associated with the specified item in the combo box.
CB_GETITEMHEIGHT	Determines the height of list items or the selection field in a combo box.
CB_GETLBTEXT	Gets a string from the list of a combo box.
CB_GETLBTEXTLEN	Gets the length, in characters, of a string in the list of a combo box.
CB_GETLOCALE	Gets the current locale of the combo box. The locale is used to determine the correct sorting order of displayed text for combo boxes with the CBS_SORT style and text added by using the CB_ADDSTRING message.
CB_GETMINVISIBLE	Gets the minimum number of visible items in the

	drop-down list of a combo box.
CB_GETTOPINDEX	An application sends the CB_GETTOPINDEX message to retrieve the zero-based index of the first visible item in the list box portion of a combo box. Initially, the item with index 0 is at the top of the list box, but if the list box contents have been scrolled, another item may be at the top.
CB_INITSTORAGE	An application sends the CB_INITSTORAGE message before adding a large number of items to the list box portion of a combo box. This message allocates memory for storing list box items.
CB_INSERTSTRING	Inserts a string or item data into the list of a combo box. Unlike the CB_ADDSTRING message, the CB_INSERTSTRING message does not cause a list with the CBS_SORT style to be sorted.
CB_LIMITTEXT	Limits the length of the text the user may type into the edit control of a combo box.
CB_RESETCONTENT	Removes all items from the list box and edit control of a combo box.
CB_SELECTSTRING	Searches the list of a combo box for an item that begins with the characters in a specified string. If a matching item is found, it is selected and copied to the edit control.
CB_SETCUEBANNER	Sets the cue banner text that is displayed for the edit control of a combo box.
CB_SETCURSEL	An application sends a CB_SETCURSEL message to select a string in the list of a combo box. If necessary, the list scrolls the string into view. The text in the edit control of the combo box changes to reflect the new selection, and any previous selection in the list is removed.

CB_SETDROPPEDWIDTH	An application sends the CB_SETDROPPEDWIDTH message to set the maximum allowable width, in pixels, of the list box of a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.
CB_SETEDITSEL	An application sends a CB_SETEDITSEL message to select characters in the edit control of a combo box.
CB_SETEXTENDEDUI	An application sends a CB_SETEXTENDEDUI message to select either the default UI or the extended UI for a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.
CB_SETHORIZONTALEXTENT	An application sends the CB_SETHORIZONTALEXTENT message to set the width, in pixels, by which a list box can be scrolled horizontally (the scrollable width). If the width of the list box is smaller than this value, the horizontal scroll bar horizontally scrolls items in the list box. If the width of the list box is equal to or greater than this value, the horizontal scroll bar is hidden or, if the combo box has the CBS_DISABLENOSCROLL style, disabled.
CB_SETITEMDATA	An application sends a CB_SETITEMDATA message to set the value associated with the specified item in a combo box.
CB_SETITEMHEIGHT	An application sends a CB_SETITEMHEIGHT message to set the height of list items or the selection field in a combo box.
CB_SETLOCALE	An application sends a CB_SETLOCALE message to set the current locale of the combo box. If the combo box has the CBS_SORT style and strings are added using CB_ADDSTRING, the locale of a combo box affects how list items are sorted.

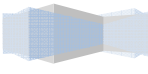
CB_SETMINVISIBLE	An application sends a CB_SETMINVISIBLE message to set the minimum number of visible items in the drop-down list of a combo box.
CB_SETTOPINDEX	An application sends the CB_SETTOPINDEX message to ensure that a particular item is visible in the list box of a combo box. The system scrolls the list box contents so that either the specified item appears at the top of the list box or the maximum scroll range has been reached.
CB_SHOWDROPDOWN	An application sends a CB_SHOWDROPDOWN message to show or hide the list box of a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Combo Box Notification Messages

Notification	Description
CBN_CLOSEUP	Sent when the list box of a combo box has been closed. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_DBLCLK	Sent when the user double-clicks a string in the list box of a combo box. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_DROPDOWN	Sent when the list box of a combo box is about to be made visible. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_EDITCHANGE	Sent after the user has taken an action that may have altered the text in the edit control portion of a combo box. Unlike the CBN_EDITUPDATE notification code, this notification code is sent after the system updates the screen. The parent window of the combo box receives this notification code through the WM_COMMAND message.

CBN_EDITUPDATE	Sent when the edit control portion of a combo box is about to display altered text. This notification code is sent after the control has formatted the text, but before it displays the text. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_ERRSPACE	Sent when a combo box cannot allocate enough memory to meet a specific request. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_KILLFOCUS	Sent when a combo box loses the keyboard focus. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_SELCHANGE	Sent when the user changes the current selection in the list box of a combo box. The user can change the selection by clicking in the list box or by using the arrow keys. The parent window of the combo box receives this notification code in the form of a WM_COMMAND message.
CBN_SELENDCANCEL	Sent when the user selects an item, but then selects another control or closes the dialog box. It indicates the user's initial selection is to be ignored. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_SELENDOK	Sent when the user selects a list item, or selects an item and then closes the list. It indicates that the user's selection is to be processed. The parent window of the combo box receives this notification code through the WM_COMMAND message.
CBN_SETFOCUS	Sent when a combo box receives the keyboard focus. The parent window of the combo box receives this notification code through the WM_COMMAND message.

Note: Refer Microsoft documentation to get information about wParam, lParam & Return Values.



Chapter 07: Managing Resources

Dialog Box

A dialog box is a secondary window that allows users to perform a command, asks users a question, or provides users with information or progress feedback.

A dialog box is similar to a popup window that contains controls. Dialog box is used to display information or request information from the user. The main difference between dialog boxes and pop-menu windows is that dialog boxes uses templates that define the controls created on the dialog box. These templates can be dynamically created in memory while parent application executes.

Dialog boxes consist of a title bar (to identify the command, feature, or program where a dialog box came from), an optional main instruction (to explain the user's objective with the dialog box), various controls in the content area (to present options), and commit buttons (to indicate how the user wants to commit to the task).

A task dialog is a dialog box implemented using the task dialog application programming interface (API). They consist of the following parts, which can be assembled in a variety of combinations:

- A title bar to identify the application or system feature where the dialog box came from.
- A main instruction, with an optional icon, to identify the user's objective with the dialog.
- A content area for descriptive information and controls.

- A command area for commit buttons, including a Cancel button, and optional more options and don't show this <item> again controls.
- A footnote area for optional additional explanations and help typically targeted at less experienced users.

Dialog box is displayed by calling one of the dialog box creation functions and passing a dialog box template and the address of a dialog function. A dialog box template describes the appearance of the dialog box: the size of the popup window, the window styles used to create the popup window, a list of the controls to create in the popup window, their sizes and locations, the font to be used for the text display in the controls and similar parameters. The dialog box creation function use the specified dialog box template to create a window based on the description in the template. The specified dialog function processes some of the messages for the pop-up window and the controls in the pop-up window.

When window creates the pop-up window for a dialog box, it creates the window based on the dialog window class. The dialog window class is predefined by windows, as are the button, list box and other control classes. The dialog window class specifies a function inside window itself as the window function for the windows of that class. That window function provides mechanism for program the controls.

There are three ways of creating a dialog box template:

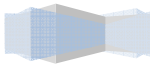
1. Use resources editor application provided with Microsoft Visual C++ and similar tools in other development environments allow you to graphically lay out the appearance of a dialog box. These tools allow saving the description of the dialog box in a text file.
2. You can directly use the text editor to create the dialog script file.
3. You can also build a data structure in memory and pass the address of that data structure to a dialog box creation function. The data structure describes the dialog box and each of controls to create the dialog box.

The first two methods produce a text file that forms the application's resource definition file. You compile the application's resource definition file using the Resource Compiler that produces an output (.res) file containing a binary version of the resources definitions.

Dialog boxes have two fundamental types:

Modal Dialog Box

- This is most common type of dialog box
- Modal dialog box disables its owner window when the dialog box is displayed. So when a modal dialog is being displayed user can't switch to another part of the same application.
- Modal dialog boxes require users to complete and close before continuing with the owner window. These dialog boxes are best used for critical or infrequent, one-off tasks that require completion before continuing.
- Modal Dialog box is created by calling one of the: DialogBox(), DialogBoxParam(), DialogBoxIndirect() or DialogBoxIndirectParam() functions.
- These functions create and display a modal dialog box and do not immediately return control but seemingly suspend.
- Because the newly created dialog box disables input to its owner window, all the mouse and keyboard input for the application flows to the windows function for the dialog box window, which then passes most of the inputs to dialog function.
- The dialog function terminates the dialog box by calling EndDialog() function and



specifying a return value.

- It is important to realize that the EndDialog() function doesn't actually terminate the dialog box. It sets a flag requesting the dialog box needed to be terminated. The EndDialog function then returns to its caller, the dialog function. Windows terminates the dialog box when the dialog function returns form processing the current message.
- The window style for a dialog box is specified in dialog box template. A dialog box must be a pop up window. It cannot have WS_CHILD style. Windows disables all child windows when their parent is disabled. A WS_CHILD style modal dialog box disables the parent which disables itself as well. Then because neither the parent nor the child window can receive input, the dialog box could never terminate.
- The recommended style for a modal dialog box are:
    - DS_MODALFRAME
    - WS_CAPTION
    - WS_SYSMENU

  Creating a window with these styles produces an overlapped window with a border that represents a modal dialog box.

Modeless Dialog Box

- Modeless Dialog box does not disable its owner window when it is created. Displaying a modeless dialog box does not stop the parent application and does not force the user to respond to the dialog box.
- Modeless dialog boxes allow users to switch between the dialog box and the owner window as desired. These dialog boxes are best used for frequent, repetitive, on-going tasks. User can interact to both owner and child windows.
- Modeless dialog box can obtain as well as loose the input focus at any time.
- Modeless dialog box can be creates by calling one of the CreateDialog(), CreateDialogIndirect(), CreateDialogParam() or CreateDialogIndirectParam() functions.
- These functions create and display a modeless dialog box, but unlike the functions for a model dialog box, they return immediately from the call. They do not wait for the dialog box to process input and terminate.
- Because the modeless dialog box remains on the screen, it must share messages with

the message loop. This requires modification to the message loop if the modeless dialog box is respond to keyboard selections using the TAB and arrow keys. A typical message loop containing a modeless dialog box is as follows:

```
while(GetMessage(&msg, NULL, 0, 0))
{
        if(hDlgModeless || !IsDialogMessage(hDlgModeless, &msg))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
}
```

hDlgModeless is HWND handle for the dialog box. This global variable is set to NULL if the modeless dialog box is not displayed and set to the dialog box handle if the modeless dialog box is currently on the screen.

IsDialogMessage() function determines whether a message is intended for the specified dialog box and, if it is the message is sent to the dialog box procedure and should not be processed by the normal TranslateMessage() and DispatchMessage() functions.

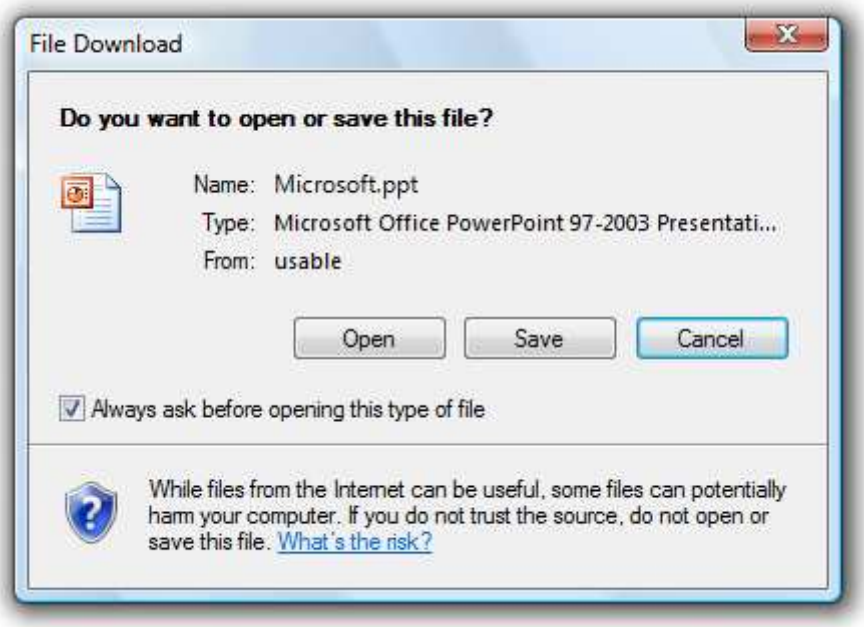
IsDialogMessage() function also converts keyboard messages into commands to dialog box control. For example, when the function detects a tab key press, it moves the input focus to the next control in the dialog box.

Once you retrieve a message, you must check to see whether it's for a modeless dialog box. Only if the modeless dialog box exists you pass the message to the IsDialogMessage() function. The IsDialogMessage() function process only those messages intented for the specified dialog box. It returns a non-zero value when it process a message and 0 otherwise. You must not pass a message processed by the IsDialogMessage function to the TranslateMessage() or DispatchMessage() functions. When the message isn't processed by the IsDialogMessage() function, it needs to be translated and dispatched as normal.

Creating dialog template in resource file

```
TESTDIALOG DIALOG DISCARDABLE 20, 20, 180, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
BEGIN
        // CONTROLS to CREATE
END
```

This statement starts the dialog box template definition. All statements from BEGIN until the END statement is reached are part of the dialog box template.



Syntax:

NameID **DIALOG** [load-option] [memory option] x, y, width, height, [options statement]

NameID	The name of the dialog box. This is character string or 16 bit unsigned integer that is used with DialogBox() to specify which dialog box control should be loaded and executed.
Load-option	Can be either PRELOAD or LOADONCALL. PRELOAD specifies that the dialog box resource data be loaded when

	the application starts. Doing so takes up memory but makes the dialog box appear quickly. LOADONCALL is the normal (default) option, in which the dialog box resource data is not loaded into memory until needed.
Memory option	A combination of FIXED, MOVABLE and DISCARDABLE. Normally, both the MOVABLE and DISCARDABLE styles are chosen.
x, y, width, height	The position and size of the dialog box.
Option statements	These include STYLE, EXSTYLE, CAPTION, CHARACTERISTICS, MENU, CLASS, LANGUAGE, VERSION and FONT. All statements after DIALOG, up to the BEGIN statement in the dialog box definition, are assumed to be option statement.

Following all of these specifications, BEGIN and END statements delimit the beginning and ending of the list of controls that are contained in dialog box. A dialog box can have up to 255 controls. Although the control IDs themselves are 16 bit values, the control limit is 255.

There are many different dialog control statements.

Statement	Window Class	Implied Windows Style
AUTOCHECKBOX	Button	BS_AUTOCHECKBOX | WS_TABSTOP
AUTO3STATE	Button	BS_AUTO3STATE | WS_TABSTOP
AUTORADIOBUTTON	Button	BS_AUTORADIOBUTTON | WS_TABSTOP
CHECKBOX	Button	BS_CHECKBOX | WS_TABSTOP
COMBOBOX	Combobox	CBS_SIMPLE | WS_TABSTOP
CONTROL	User-Specified	WS_CHILD | WS_VISIBLE
CTEXT	Static	SS_CENTER | WS_GROUP
DEFPUSHBUTTON	Button	BS_DEFPUSHBUTTON | WS_TABSTOP

EDITTEXT	Edit	ES_LEFT | WS_BORDER | WS_VSCROLL
GROUPBOX	Button	BS_GROUP | WS_TABSTOP
ICON	Static	SS_ICON
LISTBOX	Listbox	LBS_NOTIFY | WS_BORDER | WS_SCROLL
LTEXT	Static	SS_LEFT | WS_GROUP
PUSHBOX	Button	BS_PUSHBOX | WS_TABSTOP
PUSHBUTTON	Button	BS_PUSHBUTTON | WS_TABSTOP
RADIOBUTTON	Button	BS_RADIOBUTTON | WS_TABSTOP
RTEXT	Static	SS_RIGHT | WS_GROUP
SCROLLBAR	ScrollBar	SBS_HORZ
STATE3	Button	BS_3STATE | WS_TABSTOP

All control statement except COMBOBOX, CONTROL, EDITTEXT, LISTBOX and SCROLLBAR have following form:

Control-Type "Text", id, x, y, width, height, style, extended style

The COMBOBOX, EDITTEXT, LISTBOX and SCROLLBAR statements have following form:

Control-Type id, x, y, width, height, style, extended style

When you want to specify very possible parameter for a control, you need to use CONTROL statement. It allows specifying the text and the window class for the control in addition to all the normal parameters:

CONTROL "Text", id, class, style, x, y, width, height, extended-style

E.g.

CONTROL "OK", IDOK, "BUTTTON", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 116, 26, 50, 14

DialogBox Functions

Modal Dialog Box Creation

- **DialogBox**

  Creates a modal dialog box from a dialog box template resource. DialogBox does not return control until the specified callback function terminates the modal dialog box by calling the EndDialog function.

```
INT_PTR WINAPI DialogBox(
        HINSTANCE hInstance,
        LPCTSTR lpTemplate,
        HWND hWndParent,
        DLGPROC lpDialogFunc
);
```

Parameters

hInstance - A handle to the module whose executable file contains the dialog box template.

lpTemplate - The dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent - A handle to the window that owns the dialog box.

lpDialogFunc - A pointer to the dialog box procedure.

Return Value

If the function succeeds, the return value is the nResult parameter in the call to the EndDialog function used to terminate the dialog box.

If the function fails because the hWndParent parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous

versions of Windows. If the function fails for any other reason, the return value is –1.

This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- The WH_CBT hook is installed and returns a failure code
- if one of the controls in the dialog template is not registered, or its window procedure fails WM_CREATE or WM_NCCREATE

- **EndDialog**

  Destroys a modal dialog box, causing the system to end any processing for the dialog box.

```
BOOL WINAPI EndDialog(
        HWND hDlg,
        INT_PTR nResult
);
```

  Parameters

  hDlg - A handle to the dialog box to be destroyed.

  nResult - The value to be returned to the application from the function that created the dialog box.

  Return Value

  If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

- **DialogProc**

  Application-defined callback function used with the CreateDialog and DialogBox families of functions. It processes messages sent to a modal or modeless dialog box. The DLGPROC type defines a pointer to this callback function. DialogProc is a placeholder for the application-defined function name.

```
INT_PTR CALLBACK DialogProc(
        HWND hwndDlg,
        UINT uMsg,
        WPARAM wParam,
        LPARAM lParam
);
```

Parameters

hwndDlg - A handle to the dialog box.

uMsg - The message.

wParam - Additional message-specific information.

lParam - Additional message-specific information.

Return Value

Typically, the dialog box procedure should return TRUE if it processed the message, and FALSE if it did not. If the dialog box procedure returns FALSE, the dialog manager performs the default dialog operation in response to the message.

- **DialogBoxParam**

  Creates a modal dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
INT_PTR WINAPI DialogBoxParam(
        HINSTANCE hInstance,
        LPCTSTR lpTemplateName,
         HWND hWndParent,
        DLGPROC lpDialogFunc,
        LPARAM dwInitParam
);
```

  Parameters

  hInstance - A handle to the module whose executable file contains the dialog box template.

lpTemplateName - The dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent - A handle to the window that owns the dialog box.

lpDialogFunc - A pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam - The value to pass to the dialog box in the lParam parameter of the WM_INITDIALOG message.

Return Value

If the function succeeds, the return value is the value of the nResult parameter specified in the call to the EndDialog function used to terminate the dialog box.

If the function fails because the hWndParent parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is –1.

- **DialogBoxIndirect**

{Self Study}

- **DialogBoxIndirectParam**

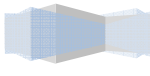
{Self Study}

Modeless Dialog Box

- **CreateDialog**

Creates a modeless dialog box from a dialog box template resource. The CreateDialog macro uses the CreateDialogParam function.

```
HWND WINAPI CreateDialog(
        HINSTANCE hInstance,
        LPCTSTR lpTemplate,
        HWND hWndParent,
```

```
        DLGPROC lpDialogFunc
);
```

Parameters

hInstance - A handle to the module whose executable file contains the dialog box template.

lpTemplate - The dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent - A handle to the window that owns the dialog box.

lpDialogFunc - A pointer to the dialog box procedure.

Return Value

If the function succeeds, the return value is the handle to the dialog box. If the function fails, the return value is NULL.

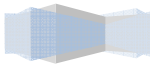
This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- The WH_CBT hook is installed and returns a failure code
- If one of the controls in the dialog template is not registered, or its window procedure fails WM_CREATE or WM_NCCREATE.

- **CreateDialogParam**

Creates a modeless dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
HWND WINAPI CreateDialogParam(
        HINSTANCE hInstance,
        LPCTSTR lpTemplateName,
```

```
    HWND hWndParent,
    DLGPROC lpDialogFunc,
    LPARAM dwInitParam
);
```

Parameters

hInstance - A handle to the module whose executable file contains the dialog box template.

lpTemplateName - The dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent - A handle to the window that owns the dialog box.

lpDialogFunc - A pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam - The value to be passed to the dialog box procedure in the lParam parameter in the WM_INITDIALOG message.

Return Value

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is NULL.

- **CreateDialogIndirect**

  {Self Study}

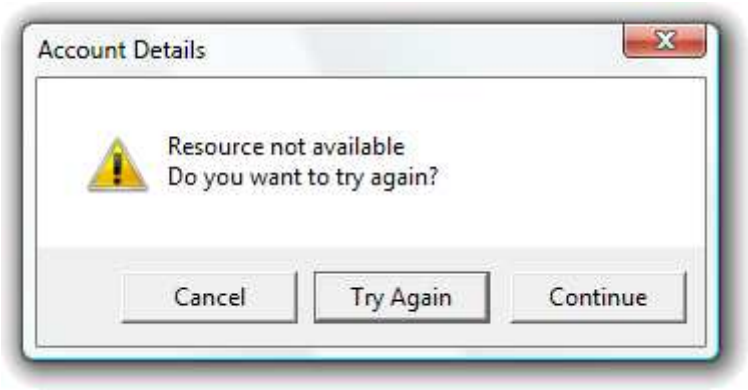
- **CreateDialogIndirectParam**

  {Self Study}

- **MessageBox**

  Many times you won't need to create a dialog box to prompt the user for simple replies. Windows supplies a message box for those times when you need to display a line or two of text to the user get a simple response, such as Yes, No or Retry.

  A message box is a window that contains a caption and a message that you supply along with a number of predefined icons and push buttons. The user responds to the message box clicking on of the push buttons. The MessageBox function returns a value indicating which button was pressed.



Message Box

- Displays a message to the user and waits for an acknowledgement
- Simply call the MessageBox() API function
- No need for DC's and WM_PAINT
- An excellent debugging tool
- Contains text, caption, pushbuttons etc.

Message Box function

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

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

Parameters

hWnd - A handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText - The message to be displayed. If the string consists of more than one line, you can separate the lines using a carriage return and/or linefeed character between each line.

lpCaption - The dialog box title. If this parameter is NULL, the default title is Error.

uType - The contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values.

Value	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_CANCELTRYCONTINUE	The message box contains three push buttons: Cancel, Try Again, and Continue. Use this message box type instead of MB_ABORTRETRYIGNORE.
MB_HELP	Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner.
MB_OK	The message box contains one push button: OK. This is the default.
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.

To display an icon in the message box, specify one of the following values.

Value	Meaning
MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
MB_ICONWARNING	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION	An icon consisting of a lowercase letter **i** in a circle appears in the message box.
MB_ICONASTERISK	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. The question-mark message icon is no longer recommended because it does not clearly represent a specific type of message and because the phrasing of a message as a question could apply to any message type. In addition, users can confuse the message symbol question mark with Help information. Therefore, do not use this question mark message symbol in your message boxes. The system continues to support its inclusion only for backward compatibility.
MB_ICONSTOP	A stop-sign icon appears in the message box.
MB_ICONERROR	A stop-sign icon appears in the message box.
MB_ICONHAND	A stop-sign icon appears in the message box.

To indicate the default button, specify one of the following values.

Value	Meaning
MB_DEFBUTTON1	The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.
MB_DEFBUTTON2	The second button is the default button.
MB_DEFBUTTON3	The third button is the default button.
MB_DEFBUTTON4	The fourth button is the default button.

Return Value

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

If the function fails, the return value is zero. If the function succeeds, the return value is one of the following menu-item values.

Return code/value	Description
IDABORT – 3	The Abort button was selected.
IDCANCEL – 2	The Cancel button was selected.
IDCONTINUE – 11	The Continue button was selected.
IDIGNORE – 5	The Ignore button was selected.
IDNO – 7	The No button was selected.
IDOK – 1	The OK button was selected.
IDRETRY – 4	The Retry button was selected.
IDTRYAGAIN – 10	The Try Again button was selected.
IDYES – 6	The Yes button was selected.

E.g.

```
BUTTONID = MessageBox(hwnd, “Like to exit?”, “Application”, MB_YESNO);
if(BUTTONID == IDYES)
{
        …
}
Else
{
        …
}
```

Menu

A menu is a list of items that specify options or groups of options (a submenu) for an application. Clicking a menu item opens a submenu or causes the application to carry out a command.

The Menu control presents a list of items that specify commands or options for an application. Typically, clicking a Menu-Item opens a submenu or causes an application to carry out a command. The following graphic shows the three different states of a menu control. The default state is when no device such as a mouse pointer is resting on the Menu. The focus state occurs when the mouse pointer is hovering over the Menu and pressed state occurs when a mouse button is clicked over the Menu.

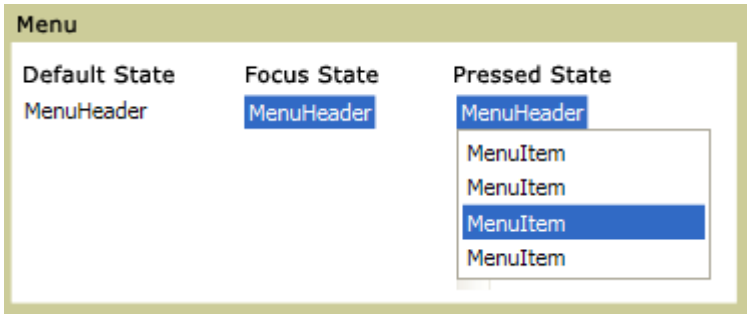


Figure: Different Menu states

When the user selects an item from a menu, windows sends a WM_COMMAND message to the window function associated with the window that owns the menu. The message specifies the menu ID, which is similar to a control ID, for the menu item that the user selected.

You can also define certain keystrokes as shortcuts to select a menu item. These are called accelerator keys. When the user press an accelerator key, windows sends the same message to the window function as it does when the user selects the menu item with the same menu ID.

A top level menu is the horizontal list of items displayed just below the caption of a window. Entries in a menu are displayed either as text or bitmaps. An entry in a tip level menu can be either a **command menu item** or **pop-up menu item**. The windows documentation provided by Microsoft refers top-level menu as the menu bar.

Selecting a label corresponding to the pop-up items produces a vertically displayed menu that pops up below the selected label. For example: the **File** label presents on the menus of windows application represents a pop-up menu, when the user selects File, its pop-up appears offering file related, additional choices such as New, Open, Close… etc.

Command menu items can be present on top-level menu although they are most often found on pop-up menus. Command menu items represent a final selection. When the user selects a command menu item, windows sends a WM_COMMAND messages to the window function identifying the menu item that is selected.

You can also create menus with more than one level of pop-up menus. Multiple levels of pop-ups are called cascading menus. A pop-up menu and cascading pop-up menu are normally owned by another menu. The user selects a pop-up menu item from the owner menu and windows display the pop-up menu. You can also explicitly request windows to display a pop-up menu at an arbitrary location on the screen whenever you wish. Generally, you'll display this pop-up menu at fixed location, at the current cursor position or a position relative to a location in the application's client area, usually in response to a right-mouse button notification. This particular type of pop-up menu is not associated with another menu. Such pop-up menu is called floating pop-up menu. These are the menus that are normally invoked with a right mouse button click and are application and context specified.

Defining and Creating a Menu

We can define a menu in the following three ways:

1. Define a menu resource file.
2. Create a menu in a resource file
3. Create a menu from a memory template

The most common method is defining an application's menus in the applications resource definition file. This allows you to change a menu – for example, into a different language – by changing only the resource definition file. You don't to make any changes to the application source code or to re-link the application.

What are resources?

- Objects that are used inside the program but defined outside the program
- Custom type of objects such as menus, icons, dialog box, bitmaps etc.
- Created separately from the program but added to the .exe when the program is linked.

What are resource files?

- Files containing the resources of menu, icon, dialog box, bitmap etc.
- Generally resources file should have file names same as that of .exe but with the extension of .rc.
- Text resources for menus as well as Icons, cursor made by resource editor are referred from the .rc file.

Compilation of the resources files

- .rc files are not like c/cpp source files but the resource scripts.
- Compiled using resource compiler (rc.exe) that converts the resource files (.rc) in to .res files.
- .res file is then linked to application
- We don't need to compile the .rc file, IDE like Visual C++ automatically compiles it.

Defining a Menu Template in your Resource Definition File

A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands and open submenus from a list of menu items.

Menus of all types are often defined in an application's resource definition (*.rc) file by one or more MENU statement. Each MENU statement must be followed by BEGIN and END keywords, one or more MENUITEM and/or POPUP statements and terminated by the END keyword. Like dialogs, a menu resource is named either by a string or by an integer. The MENU or MENUX statement begins the menu resource definition.

The MENU resource definition statement specifies the contents of a resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands and open submenu from a list of menu items.

The MENU statement has following syntax:

menuID MENU [optional-statements] [memory-option]

BEGIN // or {

item-definitions

END // or }

The MENUX statement has following syntax:

menuID MENUEX

BEGIN // or {

item-definitions

END // or }

Note: These contents are written in *.rc file.

Parameters

menuID - Number that identifies the menu. This value is either a unique string or a unique 16-bit unsigned integer value in the range of 1 to 65,535.

optional-statements - This parameter can be zero of more of the following statements.

Statement	Description
CHARACTERISTICS	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE	Language for the resource.

VERSION	User-defined version number for the resource that can be used by tools that read and write resource files.

Example:

//The following is an example of a complete MENU statement:

```
sample MENU
{
     MENUITEM "&Soup", 100
     MENUITEM "S&alad", 101
     POPUP "&Entree"
     {
          MENUITEM "&Fish", 200
          MENUITEM "&Chicken", 201, CHECKED
          POPUP "&Beef"
          {
               MENUITEM "&Steak", 301
               MENUITEM "&Prime Rib", 302
          }
     }
     MENUITEM "&Dessert", 103
}
```

The menuID parameter specifies a name of integer that you use to identify the menu resource. The load option parameter has no meaning in win32 programming. The memory option parameter, if present is DISCARDABLE.
The option statements can be any of LAGUAGE, CARACTERISTICS or VERSION and apply to the menu declaration but not MENUEX declaration.

Each POPUP statement defines a pop-up menu. A POPUP statement contains a text string that specifies the name of the popup menu and optionally, one or more keywords that control the appearance of the popup menu name. A POPUP statement must be

followed by a BEGIN (or {), one or more MENUITEM and/or POPUP statements and terminated by the END (or }) keyword just like MENU statement.

A POPUP statement in a MENU declaration has following syntax:

POPUP "Text" [Option-List]
BEGIN
Items Declaration
END

MENUITEM declaration has following syntax:

MENUITEM "Text", ItemID [Option-List]

The text parameter specifies the character string that is displayed for a pop-up menu. An '&' in the string causes the character that follows it to be underlined. Windows considers an underlined character in text for a menu to be mnemonic for the item.

The ItemID parameter of MENUITEM statement specifies the menu item's ID, an integer that is similar to the controlID paramenter of the control. This integer value is sent to the window that owns the menu when the user selects the item's name. When the user selects an enabled MEUITEM (Other than the separator) from a menu, windows sends a WM_COMMAND message to the window function for the window that owns the menu. The wParam parameter of the message contains the ItemID value.

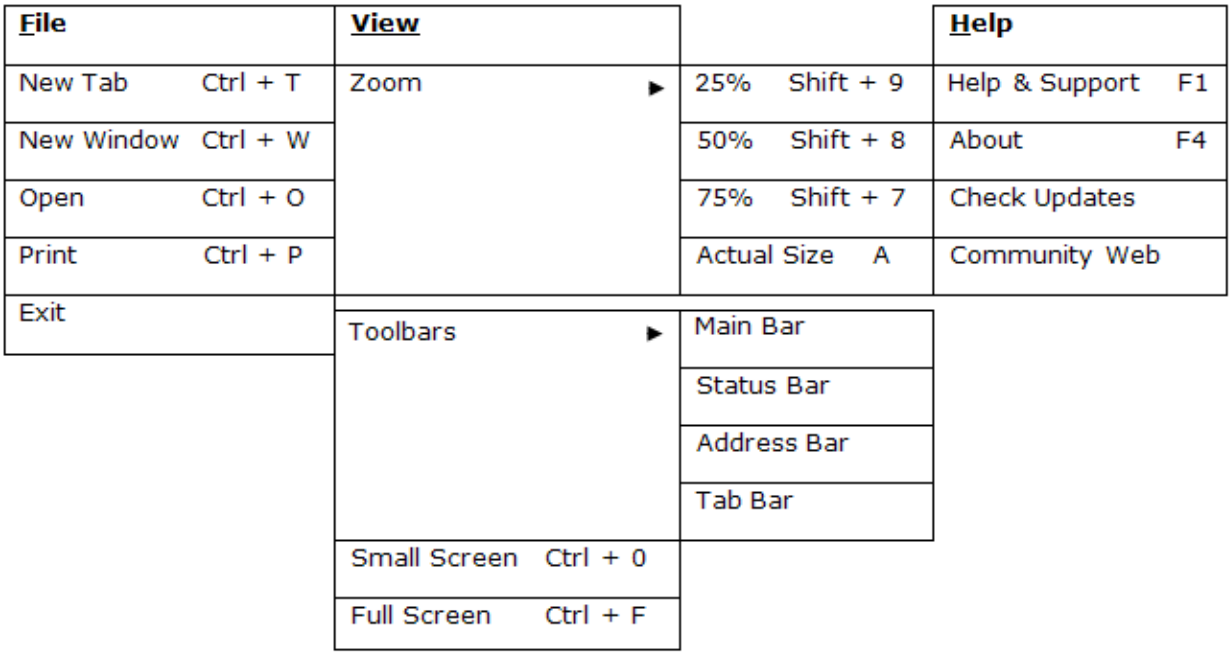
The option-list controls the appearance of the menu. It can be specified for MENUITEM as well as POPUP statements, although all options are valid in all cases.

Option	Description
CHECKED	The item is initially displayed with a check mark to the left of the text.
GRAYED	The item is initially inactive, cannot be selected and appears on the menu in a gray or lightened color. This option cannot be combined with INACTIVE.
HELP	The item has a vertical separator to the left of the text. This

	option is valid only on MENUITEM statements.
INACTIVE	The option is initially inactive and cannot be selected but appears on the menu in the normal menu-text color. This option cannot be combined with GRAYED.
MENUBREAK	The item begins on a new line when used on top-level menu items. When used on an entry in a POPUP menu, this item begins a new column in the POPUP menu.
MENUBARBREAK	This option is identical to MENUBREAK except that a vertical bar separates a new column from the old when used on a entry in a POPUP menu.
SEPARATOR	A item is an inactive item, represented be the horizontal line drawn across the width of the menu. This attribute cannot be combined with any other attributes and has no text or menuID.

Question:

Write the necessary code fragment to display the following Menu.



Then write windows procedure to handle the WM_COMMAND messages for the selection of menu Items. Use Message Box to tell the user about menu item that is clicked. If the 'Exit' is selected, the application should be closed.

Steps to remember while creating and displaying menu

1. Define the MENUITEMS in *.h file then it will be easier to use later on. This can also be done in source file itself. We can also use INTEGER values directly without defining any where.

   A sample of the header file having macro definitions of the MenuID's used in the resource file.

   e.g.

   mymenu.h

   ```
   #define     IDM_ONE     100
   #define     IDM_TWO     101
   #define     IDM_HELP    102
   ```

   - ID values must be unique
   - Valid range of values is 0-65565
   - This Header File **must** be added to the source file as well as the Resource File.

2. **Create resource template with required POPUP and Menu items.**

   e.g.

   mymenu.rc

   ```
   #include "mymenu.h"
   MYMENU MENU
   {
       POPUP "&First"
       {
           MENUITEM "&ItemOne" , IDM_ONE        ;This is comment
           MENUITEM "I&temTwo" , IDM_TWO
       }
   MENUITEM "&Help", IDM_HELP
   }
   ```

Remember

- POPUP's do not have menuID's associated with them
- The menu ID's are defined in a separate header file as macros

- An **&** causes the succeeding character to become the shortcut key associated with that option (no duplicates in a single menu)
- Comments are allowed on any empty line. It can be used in c-style or '**;**' as the first character

3. Assign the **lpszMenuNamelpszMenuName** field of the **window class structure** a pointer to a string that contains the name of the menu.

E.g.

```
wcl.lpszMenuName = "MYMENU";
```

Menu Selection Handler

- WM_COMMAND Message is posted when a menu item is selected by the user
- **LOWORD(LOWORD(wParamwParam))** contains the menu item's ID constant corresponding to the value of the macro associated with the selected menu item
- To check which item was selected by the user, a **nested switch** statement is to be used inside the **WM_COMMAND Message Handler**
- The **LOWORD(LOWORD(wParamwParam))** is to be checked by the inner switch block to get the ID of the selected item of the menu.

Contd. ...

MENUEX

Defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands and open submenus from a list of menu items.

It also defines the following:

- Help identifiers on menus.
- Identifiers on menus.
- Use of the MFT_* type flags and MFS_* state flags.

MENUITEMINFO Structure

```
typedef struct tagMENUITEMINFO {
  UINT     cbSize;
  UINT     fMask;
  UINT     fType;
  UINT     fState;
  UINT     wID;
  HMENU    hSubMenu;
  HBITMAP  hbmpChecked;
  HBITMAP  hbmpUnchecked;
  ULONG_PTR dwItemData;
  LPTSTR   dwTypeData;
  UINT     cch;
  HBITMAP  hbmpItem;
} MENUITEMINFO, *LPMENUITEMINFO;
```

MENUEX Syntax

```
menuID MENUEX
{
        MENUITEM itemText [id], [type], [state]
        POPUP itemText [id], [type], [state], [helpID]
        {
                popupBody
        }
}
```

Parameters

MENUITEM - Defines a menu item.

itemText - String containing the text for the menu item.

id - Numeric expression indicating the identifier of the menu item.

type - Numeric expression indicating the type of the menu item, to use the predefined MFT_* type values, include the following statement in your .rc file:

#include "winuser.h"

state - Numeric expression indicating the state of the menu item To use the predefined MFS_* state values, include the following statement in your .RC file:

#include "winuser.h"

POPUP - Defines a menu item that has a submenu associated with it. A POPUP statement in the context of MENUEX declaration is also quite different from a POPUP statement in a MENU declaration.

A POPUP statement in the context of MENU declaration supports only text and a set of simple options. A POPUP statement in the context of MENUEX declaration suppots an extensive set of options, including the ability to attach an ID to the POPUP and also the helped.

itemText - String containing the text for the menu item.

id - Numeric expression indicating the identifier of the menu item.

type - Numeric expression indicating the type of the menu item To use the predefined MFT_* type values, include the following statement in your .RC file:

#include "winuser.h"

state - Numeric expression indicating the state of the menu item To use the predefined MFS_* state values, include the following statement in your .rc file:

#include "winuser.h"

helpID - Numeric expression indicating the identifier used to identify the menu during WM_HELP processing.

Menu Types for MENUITEM

MFT_BITMAP	The menu item will be bitmaps.
MFT_MENUBARBREAK	Places the menu item on a new line if it is in a menu bar or in a new column if it is a POPUP menu. When a POPUP menu is broken into multiple columns, a vertical bar will separate the column.
MFT_MENUBREAK	Places the menu item on a new line if it is in a menu bar or in a new column if it is a POPUP menu. When a POPUP menu is broken into multiple columns, there is no vertical bar separator between the columns.
MFT_OWNERDRAW	The menu item will be drawn by the window that owns the menu.
MFT_RADIOCHECK	Displays a checked menu item using the radio button mark, provided the checked bitmap has not been over-ridden by an explicit bitmap.
MFT_RIGHTJUSTIFY	Right justifies the menu item and all to its right. This is valid only for top level menus.
MFT_SEPARATOR	The menu item is a horizontal line drawn across the width of the menu. This option is a valid only for POPUP menus, not top-level menus.
MFT_STRING	The menu is displayed using a string.

Menu States

Flag	Description
MFS_CHECKED	The menu item is checked.
MFS_DEFAULT	The menu item will be the default. There is only one default menu item in any POPUP menu.
MFS_DISABLED	The menu item is disabled, but not grayed out.
MFS_ENABLED	The menu item is not enabled.
MFS_GRAYED	The menu item is disabled and is also grayed.
MFS_HILITE	The menu item is highlighted.
MFS_UNCHECKED	Removes the check mark.

MFS_UNHILITE	Removes any highlighting from the menu item. This is the default state.

popupBody - Contains any combination of the MENUITEM and POPUP statements.

Remarks

The valid arithmetic and Boolean operations that can be contained in any of the numeric expressions in the statements of MENUEX are as follows:

- Add ('+')
- Subtract ('-')
- Unary minus ('-')
- Unary NOT ('~')
- AND ('&')
- OR ('|')

Some of the Menu Related functions

- **SetMenu**

  Assigns a new menu to the specified window.

```
BOOL WINAPI SetMenu(
        HWND hWnd,
        HMENU hMenu
);
```

Parameters

hWnd - A handle to the window to which the menu is to be assigned.

hMenu - A handle to the new menu. If this parameter is NULL, the window's current menu is removed.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

- The window is redrawn to reflect the menu change. A menu can be assigned to any window that is not a child window.
- The SetMenu function replaces the previous menu, if any, but it does not destroy it. An application should call the DestroyMenu function to destroy the menu.

- **Radio Button Menu Item**

Radio button items are variant on the check mark style for a menu item intended to show a set of disjoint choices.

- **CheckMenuRadioItem**

Checks a specified menu item and makes it a radio item. At the same time, the function clears all other menu items in the associated group and clears the radio-item type flag for those items.

```
BOOL WINAPI CheckMenuRadioItem(
        HMENU hmenu,
        UINT idFirst,
        UINT idLast,
        UINT idCheck,
        UINT uFlags
);
```

Parameters

hmenu - A handle to the menu that contains the group of menu items.

idFirst - The identifier or position of the first menu item in the group.

idLast - The identifier or position of the last menu item in the group.

idCheck - The identifier or position of the menu item to check.

uFlags - Indicates the meaning of idFirst, idLast, and idCheck. If this parameter is MF_BYCOMMAND, the other parameters specify menu item identifiers. If it is MF_BYPOSITION, the other parameters specify the menu item positions.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the

return value is zero. To get extended error information, use the GetLastError function.

Remarks

The CheckMenuRadioItem function sets the MFT_RADIOCHECK type flag and the MFS_CHECKED state for the item specified by idCheck and, at the same time, clears both flags for all other items in the group. The selected item is displayed using a bullet bitmap instead of a check-mark bitmap.

- **CheckMenuItem**

  Sets the state of the specified menu item's check-mark attribute to either selected or clear.

```
DWORD WINAPI CheckMenuItem(
        HMENU hmenu,
        UINT uIDCheckItem,
        UINT uCheck
);
```

Parameters

hmenu - A handle to the menu of interest.

uIDCheckItem - The menu item whose check-mark attribute is to be set, as determined by the uCheck parameter.

uCheck - The flags that control the interpretation of the uIDCheckItem parameter and the state of the menu item's check-mark attribute. This parameter can be a combination of either MF_BYCOMMAND, or MF_BYPOSITION and MF_CHECKED or MF_UNCHECKED.

Value	Meaning
MF_BYCOMMAND	Indicates that the uIDCheckItem parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default, if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.

MF_BYPOSITION	Indicates that the uIDCheckItem parameter gives the zero-based relative position of the menu item.
MF_CHECKED	Sets the check-mark attribute to the selected state.
MF_UNCHECKED	Sets the check-mark attribute to the clear state.

Return Value

The return value specifies the previous state of the menu item (either MF_CHECKED or MF_UNCHECKED). If the menu item does not exist, the return value is -1.

Floating POPUP Menu

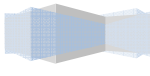
Nearly all POP-UP menus you use in windows application are owned by the menu on the menu bar. However, windows also provide the **TrackPopupMenu** and **TrackPopMenuEx** functions that allow you to display a popup menu anywhere on the screen. Because the POPUP is not attached to a menu item on the menu bar but can float around and be displayed anywhere on the screen, it's called floating POPUP Menu. These are most often in response to a WM_CONTEXTMENU message being received.

To use the TrackPopupMenu or TrackPopupMenuEx functions, the first thing required is a handle to a POPUP menu. The easiest way to get it is to define a menu in a resource file that contains the desired POPUP menu. Then it can be loaded and handle can be retrieved using GetSubMenu function.

GetSubMenu

Retrieves a handle to the drop-down menu or submenu activated by the specified menu item.

```
HMENU WINAPI GetSubMenu(
        HMENU hMenu,
        int nPos
);
```

Parameters

hMenu - A handle to the menu.

nPos - The zero-based relative position in the specified menu of an item that activates a drop-down menu or submenu.

Return Value

If the function succeeds, the return value is a handle to the drop-down menu or submenu activated by the menu item. If the menu item does not activate a drop-down menu or submenu, the return value is NULL.

For example: The following code retrieve the handle to the POPUP File menu contained in MyMenu template.

```
HMENU hMenu;
HMENU hPopup;
hMenu=LoadMenu(hInstance, "MyMenu");
hPopup=GetSubMenu(hMenu, 0);
```

TrackPopupMenu

Displays a shortcut menu at the specified location and tracks the selection of items on the menu. The shortcut menu can appear anywhere on the screen.

```
BOOL WINAPI TrackPopupMenu(
        HMENU hMenu,
        UINT uFlags,
        int x,
        int y,
        int nReserved,
        HWND hWnd,
        const RECT *prcRect
);
```

Parameters

hMenu - A handle to the shortcut menu to be displayed. The handle can be

obtained by calling CreatePopupMenu to create a new shortcut menu, or by calling GetSubMenu to retrieve a handle to a submenu associated with an existing menu item.

uFlags - Use zero of more of these flags to specify function options. Use one of the following flags to specify how the function positions the shortcut menu horizontally.

Value	Meaning
TPM_CENTERALIGN	If this flag is set; the function centers the shortcut menu horizontally relative to the coordinate specified by the x parameter.
TPM_LEFTALIGN	If this flag is set, the function positions the shortcut menu so that its left side is aligned with the coordinate specified by the x parameter.
TPM_RIGHTALIGN	Positions the shortcut menu so that its right side is aligned with the coordinate specified by the x parameter.

Use one of the following flags to specify how the function positions the shortcut menu vertically.

Value	Meaning
TPM_BOTTOMALIGN	If this flag is set, the function positions the shortcut menu so that its bottom side is aligned with the coordinate specified by the y parameter.
TPM_TOPALIGN	If this flag is set; the function positions the shortcut menu so that its top side is aligned with the coordinate specified by the y parameter.
TPM_VCENTERALIGN	If this flag is set, the function centers the shortcut menu vertically relative to the coordinate specified by the y parameter.

Use the following flags to determine the user selection without having to set up a parent window for the menu.

Value	Meaning
TPM_NONOTIFY	If this flag is set, the function does not send notification messages when the user clicks on a menu item.
TPM_RETURNCMD	If this flag is set, the function returns the menu item identifier of the user's selection in the return value.

Use one of the following flags to specify which mouse buttons the shortcut menu tracks.

Value	Meaning
TPM_LEFTBUTTON	If this flag is set, the user can select menu items with only the left mouse button.
TPM_RIGHTBUTTON	If this flag is set, the user can select menu items with both the left and right mouse buttons.

Use any reasonable combination of the following flags to modify the animation of a menu. For example, by selecting a horizontal and a vertical flag you can achieve diagonal animation.

Value	Meaning
TPM_HORNEGANIMATION	Animates the menu from right to left.
TPM_HORPOSANIMATION	Animates the menu from left to right.
TPM_NOANIMATION	Displays menu without animation.
TPM_VERNEGANIMATION	Animates the menu from bottom to top.
TPM_VERPOSANIMATION	Animates the menu from top to bottom.

For any animation to occur, the SystemParametersInfo function must set SPI_SETMENUANIMATION. Also, all the TPM_*ANIMATION flags, except TPM_NOANIMATION, are ignored if menu fade animation is on, See the SPI_GETMENUFADE flag in SystemParametersInfo.

Use the TPM_RECURSE flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

To have text layout from right-to-left, use TPM_LAYOUTRTL. By default, the text layout is left-to-right.

x - The horizontal location of the shortcut menu, in screen coordinates.

y - The vertical location of the shortcut menu, in screen coordinates.

nReserved - Reserved; must be zero.

hWnd - A handle to the window that owns the shortcut menu. This window receives all messages from the menu. The window does not receive a WM_COMMAND message from the menu until the function returns. If you specify TPM_NONOTIFY in the uFlags parameter, the function does not send messages to the window identified by hWnd. However, you must still pass a window handle in hWnd. It can be any window handle from your application.

prcRect - Ignored.

Return Value

If you specify TPM_RETURNCMD in the uFlags parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.

If you do not specify TPM_RETURNCMD in the uFlags parameter, the return value is nonzero if the function succeeds and zero if it fails.

TrackPopupMenuEx

Displays a shortcut menu at the specified location and tracks the selection of items on the shortcut menu. The shortcut menu can appear anywhere on the screen.

```
BOOL WINAPI TrackPopupMenuEx(
        HMENU hmenu,
        UINT fuFlags,
        int x,
        int y,
```

```
HWND hwnd,
LPTPMPARAMS lptpm
);
```

[Description: Study Assignment]

Accelerators

Accelerators are closely related to menus – both provide the user with access to an application's command set. Typically users rely on an application's menus to learn the command set and then switch over to using accelerators as they become more proficient with the application. Accelerators provide faster and more direct access to commands then menu do. At a minimum, an application should accelerators for the more commonly used commands. Although accelerators typically generate commands that exist as menu, they can also generate commands that have no equivalent menu items.

Using an accelerator is the same as choosing a menu item: Both actions cause the system to send a WM_COMMAND or WM_SYSCOMMAND message to the corresponding window procedure. The WM_COMMAND message includes an identifier that the window procedure examines to determine the source of the message. If an accelerator generates the WM_COMMAND message, the identifier is that of the accelerator. Similarly, is a menu item generated the WM_COMMAND message; the identifier is that of the menu item. Because an accelerator provides a shortcut for choosing a command from a menu, an application usually assigns the same identifier to the accelerator and the corresponding menu item.

An ASCII character code or a virtual key code can be used to define the accelerator. An ASCII character code makes the accelerator case sensitive. Thus, using the ASCII 'C' character defines the accelerator ALT + C rather than ALT +c. However, case sensitive accelerators can be confusing to use. For example, the ALT + C accelerator will only be generated if the caps lock or shift key is pressed but not if both are down.

Typically accelerators don't need to be case sensitive, so most applications use virtual key codes for accelerators rather than ASCII character codes.

Avoid accelerators that conflict with an application's menu mnemonics, because the accelerator overrides the mnemonics, which can confuse the user.

If an application defines an accelerator that is also defined in the system accelerator table, the application defined the system accelerator, but only within the context of the application. Avoid this practice, however, because it prevents the system accelerator from performing its standard role in the user interface. Some system wide accelerators are described in the following list.

Accelerator	Desscription
ALT + ESC	Switches to the next application.
ALT + F4	Closes an application or a window.
F1	Help.
CTRL + F4	Close the active group or document window.
ALT + TAB	Switches to the next application.
CTRL + ESC	Switch to Start menu
ALT + SPACEBAR	Opens the window menu for the application's main window.
PRINT SCREEN	Copies an image of the screen onto the clipboard
ALT + PRINT SCREEN	Copies an image of the active window onto the clipboard

After you create the menu that informs the user of the accelerator keys, you must create an accelerator table. An accelerator table maps the accelerator keys to their corresponding command menu items.

Several steps are required to create an accelerator table for an application. First, a resource compiler is used to create accelerator table resource and to add them to the application's executable file. At run time, the LoadAccelerators function is used to load the accelerator into the memory and retrieve the handle of the accelerator table. This handle is passed to the TranslateAccelerator() function to activate the accelerator table.

An accelerator table can also be created for an application at run time by passing an array of ACCEL structure to the CreateAcceleratorTable() function. This method supports user-defined accelerators in the application. Like the LoadAccelerators()

function, CreateAcceleratorTable() returns an accelerator table handle that can be passed to TranslateAccelerator() to activate the accelerator table.

The system automatically destroys accelerator tables loaded by LoadAccelerators(). However, an accelerator table created by CreateAcceleratorTable() must be destroyed before an application closes; otherwise, the table continues to exist in the memory after tha application has closed. An accelerator table is destroyed by calling the DestroyAcceleratorTable() function.

LoadAccelerators Function

Loads the specified accelerator table.

```
HACCEL WINAPI LoadAccelerators(
        HINSTANCE hInstance,
        LPCTSTR lpTableName
);
```

Parameters

hInstance - A handle to the module whose executable file contains the accelerator table to be loaded.

lpTableName - The name of the accelerator table to be loaded. Alternatively, this parameter can specify the resource identifier of an accelerator-table resource in the low-order word and zero in the high-order word. To create this value, use the MAKEINTRESOURCE macro.

Return Value

If the function succeeds, the return value is a handle to the loaded accelerator table. If the function fails, the return value is NULL.

Remarks

If the accelerator table has not yet been loaded, the function loads it from the specified executable file. Accelerator tables loaded from resources are freed automatically when the application terminates.

TranslateAccelerator Function

Processes accelerator keys for menu commands. The function translates a WM_KEYDOWN or WM_SYSKEYDOWN message to a WM_COMMAND or WM_SYSCOMMAND message (if there is an entry for the key in the specified accelerator table) and then sends the WM_COMMAND or WM_SYSCOMMAND message directly to the specified window procedure. TranslateAccelerator does not return until the window procedure has processed the message.

```
int WINAPI TranslateAccelerator(
        HWND hWnd,
        HACCEL hAccTable,
        LPMSG lpMsg
);
```

Parameters

hWnd - A handle to the window whose messages are to be translated.

hAccTable - A handle to the accelerator table. The accelerator table must have been loaded by a call to the LoadAccelerators function or created by a call to the CreateAcceleratorTable function.

lpMsg - A pointer to an MSG structure that contains message information retrieved from the calling thread's message queue using the GetMessage or PeekMessage function.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Creating Accelerator

- Accelerator keys are special keystrokes defined by the user that, when pressed, automatically select a menu option even though the menu in which that option resides is not displayed
- Allows to select an item directly bypassing the menu entirely

ACCELERATORS Resource

Defines one or more accelerators for an application. An accelerator is a keystroke defined by the application to give the user a quick way to perform a task.

Accelerator Key Table is added to the Resource File by:

acctablename ACCELERATORS [optional-statements] {event, idvalue, [type] [options]... }

Parameters

acctablename - Unique name or a 16-bit unsigned integer value that identifies the resource.

optional-statements - Zero or more of the following statements.

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language, sublanguage*	Specifies the language for the resource.
VERSION dword	User-defined version number for the resource that can be used by tools that read and write resource files.

event - Keystroke to be used as an accelerator. It can be any one of the following character types.

Type	Description
"char"	A single character enclosed in double quotation marks ("). The character can be preceded by a caret (^), meaning that the character is a control character.
Character	An integer value representing a character. The type parameter must be ASCII.

virtual-key character	An integer value representing a virtual key. The virtual key for alphanumeric keys can be specified by placing the uppercase letter or number in double quotation marks (for example, "9" or "C"). The type parameter must be VIRTKEY.

idvalue - a 16-bit unsigned integer value that identifies the accelerator.

type - Required only when the event parameter is a character or a virtual-key character. The type parameter specifies either ASCII or VIRTKEY; the integer value of event is interpreted accordingly. When VIRTKEY is specified and event contains a string, event must be uppercase.

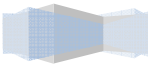
options - options that define the accelerator. This parameter can be one or more of the following values.

Option	Description
NOINVERT	Specifies that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If NOINVERT is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used. This attribute is obsolete and retained only for backwards compatibility with resource files designed for 16-bit Windows.
ALT	Causes the accelerator to be activated only if the ALT key is down. Applies only to virtual keys.
SHIFT	Causes the accelerator to be activated only if the SHIFT key is down. Applies only to virtual keys
CONTROL	Defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the event parameter. Applies only to virtual keys.

The following example demonstrates the use of accelerator keys.

```
#define IDM_AMENU 107
        IDM_AMENU ACCELERATORS
        {
                "^C",  IDDCLEAR                          ; control C
                "K",   IDDCLEAR                           ; shift K
                "k",   IDDELLIPSE, ALT                   ; alt k
                98,    IDDRECT, ASCII                    ; b
                66,    IDDSTAR, ASCII                    ; B (shift b)
                "g",   IDDRECT                            ; g
                "G",   IDDSTAR                           ; G (shift G)
                VK_F1, IDDCLEAR, VIRTKEY                 ; F1
                VK_F1, IDDSTAR, CONTROL, VIRTKEY         ; control F1
                VK_F1, IDDELLIPSE, SHIFT, VIRTKEY        ; shift F1
                VK_F1, IDDRECT, ALT, VIRTKEY             ; alt F1
                VK_F2, IDDCLEAR, ALT, SHIFT, VIRTKEY     ; alt shift F2
                VK_F2, IDDSTAR, CONTROL, SHIFT, VIRTKEY  ; ctrl shift F2
                VK_F2, IDDRECT, ALT, CONTROL, VIRTKEY    ; alt control F2
        }
```


Contd. ...



Icon

Most windows applications display icons as part of their graphical interface. An Icon is a small image to represent a component of your application. Most applications use an icon to represent the application's main window when the window is minimized. Some applications display icons in the client area of their window.

Defining an Icon Resources

A program typically defines the icon that it uses in the application's resources definition file. The ICON statements in this file define a name or integer ID for the icon and specify the name of a file that contains the ICON. An ICON statement has the following syntax:

ICON_ID ICON [MEMORY-OPTION] Filename

The ICON_ID specifies either a unique name or an integer value that identifies the resource. The load option field is ignored in win32 and is recognized only for compatibility with earlier resource. The memory options can be DISCARDABLE. The file name field is the name of the file that contains an icon resource.

E.g.

WarnIcon ICON DISCARDABLE 'warn.ico'

An Icon may contain a number of bitmap images. Each image in the file represents the same icon but they can have different resolution. When you load icon from application resources, windows selects the image of the icon that best matches the current display device.

ICONs with different Style

- A standard icon, 32 X 32 pixels, in 16 Colors
- A High-Resolution color icon, 32 X 32 pixels, in 256 colors
- A Large, high resolution color icon, 48 X 48 pixels, in 256 colors
- A Small icon, 16 X 16 pixels, in 16 colors

Loading an Icon Resource

You load an icon from the application's resources using the LoadIcon function. The following code shows two methods to load the same icon. Which one to select depends upon how you identified the icon resource definition file (*.rc file).

```
#define        IDI_WARN     101
HICON          hIcon;
hIcon= LoadIcon(hInstance, 'WarnIcon');
hIcon= LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WARNICON));
```

Drawing own Icon

DrawIcon

Draws an icon or cursor into the specified device context.

```
BOOL WINAPI DrawIcon(
        HDC hDC,
        int X,
        int Y,
        HICON hIcon
        );
```

Parameters

hDC – A handle to the device context into which the icon or cursor will be drawn.

X - The logical x-coordinate of the upper-left corner of the icon.

Y - The logical y-coordinate of the upper-left corner of the icon.

hIcon - A handle to the icon to be drawn.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

DrawIcon places the icon's upper-left corner at the location specified by the X and Y parameters. The location is subject to the current mapping mode of the device context.

DrawIcon draws the icon or cursor using the width and height specified by the system metric values for icons; for more information, see GetSystemMetrics.

Chapter 08: Printing

Printing in the page similar to printing on the display in some way because in both we determine the colors and the size if the display, create a Device Context (DC) for the device (Video Device or Printer) and print text or draw any graphical items with GDI Functions which are same functions used in the display.

However in some way it is quite different. Multiple applications can access printer concurrently but they cannot use it only one application can use the printer at a time. Output from different application must be collected separately. The printer can printer only one page at time and the application must indicate when printing will be finished but it is not necessary to graphical device. Printers are extremely slow compared to graphical devices.

An Overview of the printing Process

When an application program wants to begin using a printer, it first obtains a handle to the printer device context using CreateDC or PrintDlg. This causes the printer device driver library module to be loaded into memory and to initialize itself. The program then calls the StartDoc function, which signals the beginning of a new document. The StartDoc function is handled by the GDI module. The GDI module calls the Control function in the printer device driver, telling the device driver to prepare for printing. The process ends when the program calls EndDoc. Each page is itself enclosed by a call to StartPage to begin a page and EndPage to end the page.

1. Display PageSetup Dialog (***PageSetupDlg***) that allows the user to Choose the paper size, margins and other features. If this feature is not provided we need to assume default values.
2. Display printer selection Dialog Box to give the user the ability to select one of the installed printer. ***PrintDlg*** function is used to get the printer DC.
3. After the creation of DC for printer an abort function need to be established by calling ***SetAbortProc*** function. This function is the query handler for abort.
4. Then the small modeless dialog box will be created for showing which page is printing and accept cancellation request. The dialog is created by ***CreateDialog*** function.
5. When the small dialog box is created the main application will be disabled.

6. Beguining of new document is signaled by calling ***StartDoc*** Function. This function initializes the prnter device driver and invokes whatever other operations required for initializing the printer.
7. After start of the document we need to call ***StartPage*** function at the beguining of each page. After this function call device mode such as paper orientation, paper size etc. cannot be changed.
8. Drawing on the page done through GDI calls (TextOut, Rectangle, Ellipse etc.)
9. When the printing of the first page is finished then EndPage will be called. Then the printing process repeats form **step 7** until the end of all pages.
10. If the printing process is canceled by the user ***AbortDoc*** function will be called and if the printing process is finished successfully ***EndDoc*** function will be called then the small dialog box created in **step 4** will be terminated by calling ***DestroyWindow*** and main window will be activated with the function call ***EnableWindow***.
11. At last the printer's DC will be freed by ***DeleteDC*** function call.

Contd. ...

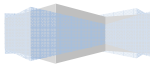
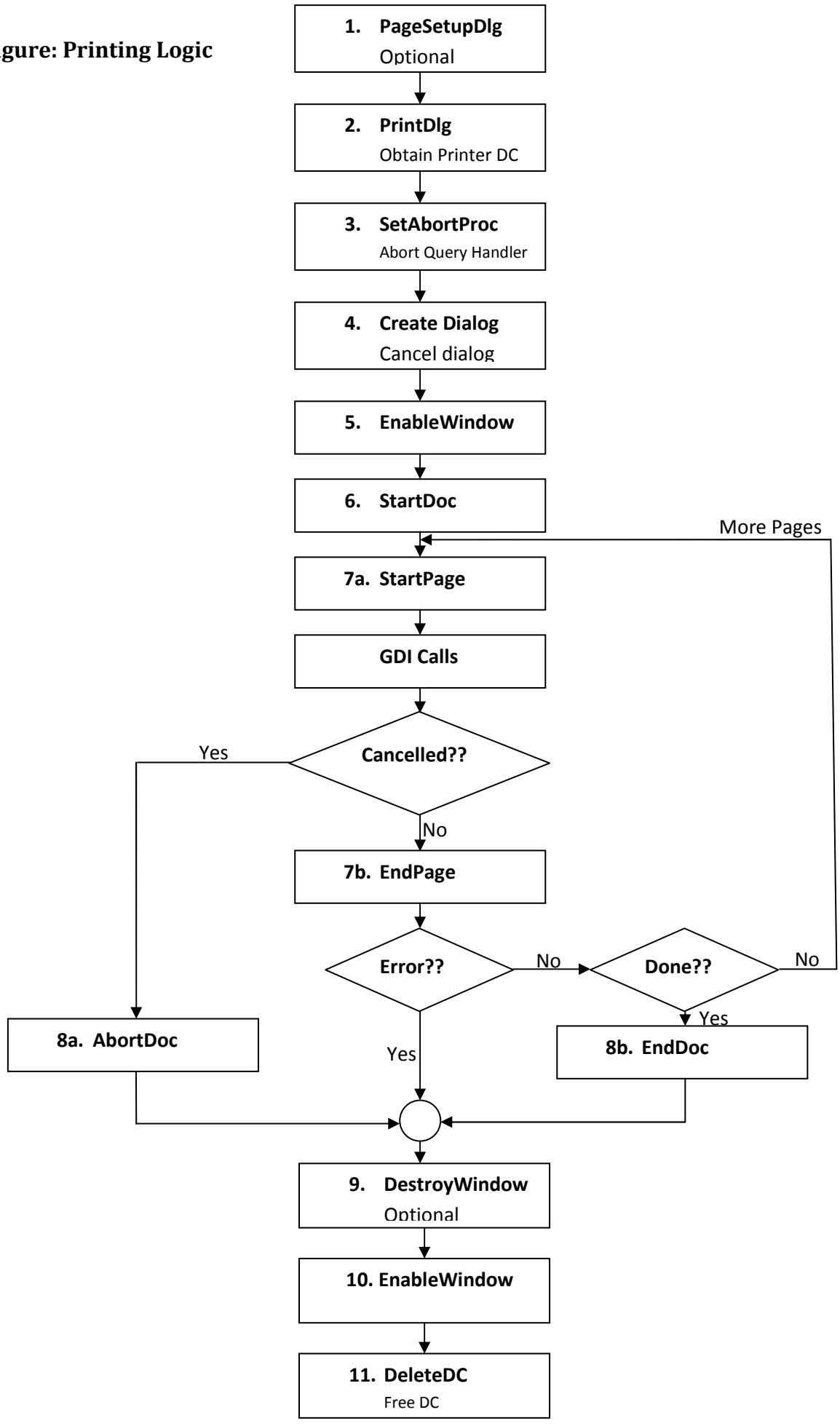


Figure: Printing Logic



Document printing and setup functions

Functions	Description
AbortDoc	Terminated a print Job.
EndDoc	End the print Job.
EndPage	Ends a Page.
PageSetupDlg	Allows the user to configure page setup such as choosing margin sizes, paper size, paper orientation etc.
PrintDlg	Allows the user to setup and configure printing options. This is important for obtaining the printer's DC.
SetAbortProc	Establishes a procedure that is called to check for user termination of the printing process.
StartDoc	Start the print job.
StartPage	Start a new page.
ResetDC	Updates a DC.
DeleteDC	Free the DC.

Chapter 09: Memory Management

Each process on 32-bit Microsoft Windows has its own virtual address space that enables addressing up to 4 gigabytes of memory. Each process on 64-bit Windows has a virtual address space of 8 terabytes. All threads of a process can access its virtual address space. However, threads cannot access memory that belongs to another process, which protects a process from being corrupted by another process. The 2 GB in low memory are available to the user and 2 GB in high is available to the kernel. The address used by a process does not represent the actual physical location in memory. The kernel maintains a page map for each process used to translate virtual address into corresponding physical address. A process cannot write outside its own address space to protect process from each other.

GlobalAlloc() and LocalAlloc() functions are available in the Win32 API. The memory allocated by these functions won't reside in segments as they did in windows 3.x instead, both type of memory are identical, allocated in the address space for the process and are accessible for using 32-bit pointers. The GlobalAlloc() & LocalAlloc() functions are provided for completeness and compatibility with 16 bit versions of windows.

Two new types of memory, virtual memory and a local heap are introduced in the Win32 environment. The virtual memory manager API is similar to the global memory manager API, except that memory can be reserved in large blocks and later actually allocated. The new heap manager allows the creation of multiple, separate heaps, giving you an efficient way to allocate small portions of memory.

Windows controls the allocation of system memory. Both windows itself and windows application allocate and release memory blocks frequently during their lifetimes. Because many applications can run concurrently under windows, dynamic storage requests can vary in amount instant by instant.

The performance of windows and applications running under it depends to a great extent on the amount of available memory. Excessive memory usage by an application can cause windows to discard and reload pages un-necessarily. Generally, this is an indication that you should allocate memory as needed ad free it as quickly as possible.

In win32 application we call GlobalAlloc to allocate the memory. When you are done with it, do a GlobalFree.

Windows uses the notion of handle for managing resources administered by the system. We have seen how the GDI uses handles to represent pens, brush, bitmaps and other resources. Windows also administers memory and represents a memory location with a memory handle. In Win32, all physical memory is handled by the virtual memory component of the operating system, and this process is invisible to the application.

Within the heap, allocated memory blocks are one of three types: fixed, movable or discardable. The type of a memory block greatly affects how your application must access the information contained in the block.

Dynamic Memory Allocation

Windows controls the allocation of system memory and represents a memory location with a memory handle. Both windows and windows application allocates and releases memory blocks frequently in their life time. In windows many applications can run concurrently so dynamic memory allocation is very mush essential. The performance of the windows and application depends upon the amount of physical memory available.

Allocated memory blocks are one of three types:

1. Fixed Memory Block
2. Moveable Memory Block
3. Discardable Memory Block

Fixed Memory Blocks

Fixed Memory blocks are the type used by most memory allocation schemes. This is same as what you get with malloc & new in C/C++ programming. Windows returns the address of the memory block when using malloc, new or calling GlobalAlloc with the GMEM_FIXED flag.

When you allocate the fixed memory block with GlobalAlloc, windows returns the address of the memory block. The address of the block never changes from the time you allocate it until you free it. There is no limit on how many fixed memory blocks you can have, up to the 2GB limit or the limit of your paging system.

Moveable Memory Blocks

The secondary type of memory block is the movable memory block. This is normally used for special allocations in Clipboard and multimedia functions. This type should be normally avoided, unless absolutely required, as far as possible because no more than 65,535 blocks are allowed. To allocate this you should use GlobalAlloc with the GMEM_MOVABLE flag. After allocation, windows doesn't return the address of the block, instead, it returns a handle to the memory block (HGLOBAL). The handle uniquely identifies that particular memory block. And when the address of the block is needed, the handle must be converted to address.

Locking a block has no effect on movable other than returning the address of the block. Normally a windows application will allocate a block and immediately lock it. The application then can use the block whenever it needs to. When the application no longer needs the storage, it should unlock and free the block.

Discardable Memory Blocks

The third kind of memory block is the discardable memory block. Windows can reallocate the discardable memory block to a 0 length (Releasing the Space) when it needs space to satisfy an allocation request. Doing this destroys all data contained in the memory block. Discardable block are typically used to hold information that is convenient to keep in memory but that can easily be recreated when necessary.

To get the discardable block you should use GlobalAlloc with the GMEM_MOVABLE and GMEM_DISCARDABLE flags.

When discardable block is allocated, windows returns a handle to the block, just like it does for a movable block. While locking the block prior to accessing it, windows returns the address of the block. The handle returned when allocating a discardable memory block remains valid even after the block is discarded. When discarded block is locked, windows returns a NULL pointer because the data in the block no longer exixts. That block must be reallocated from its 0 length to required size and recreate the content.

When insufficient memory is available to satisfy a memory allocation request, windows attempts to free blocks in order to create a large enough block. If this can't be done, Windows begins discarding discardable memory blocks one by one until either a large enough free block can be created or no more discardable blocks exist.

The original goal of using discardable memory blocks for Win32 is to allow the system to postpone for as long possible the need to allocate more virtual memory to the application.

Managing Memory Blocks using the Global- Functions

```
HANDLE hMem;
hMem  = GlobalAlloc  (flags, size);
```

The signature of the GlobalAlloc function is:

```
HGLOBAL WINAPI GlobalAlloc(
 UINT uFlags,
 SIZE_T dwBytes
);
```

- The flag parameter specifies one or more flags describing the type of memory
- The dwBytes parameter specifies the number of bytes to be allocated.

uFlags - The memory allocation attributes. If zero is specified, the default is GMEM_FIXED. This parameter can be one or more of the following values, except for the incompatible combinations that are specifically noted.

Value	Meaning
GHND	Combines GMEM_MOVEABLE and GMEM_ZEROINIT.
GMEM_FIXED	Allocates fixed memory. The return value is a pointer.
GMEM_MOVEABLE	Allocates movable memory. Memory blocks are never moved in physical memory, but they can be moved within the default heap. The **return value** is a handle to the memory object. To translate the handle into a pointer, use the GlobalLock function. This value cannot be combined with GMEM_FIXED.

GMEM_ZEROINIT	Initializes memory contents to zero.
GPTR	Combines GMEM_FIXED and GMEM_ZEROINIT.

dwBytes - The number of bytes to allocate. If this parameter is zero and the uFlags parameter specifies GMEM_MOVEABLE, the function returns a handle to a memory object that is marked as discarded.

Return Value

If the function succeeds, the return value is a handle to the newly allocated memory object. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If the heap does not contain sufficient free space to satisfy the request, GlobalAlloc returns NULL. Because NULL is used to indicate an error, virtual address zero is never allocated. It is, therefore, easy to detect the use of a NULL pointer.

Allocating a Fixed Memory Block

```
HGLOBAL  hMem;
hMem  = GlobalAlloc(GMEM_FIXED,  1024);
```

The handle of a global fixed block is always the address of the allocated memory. The malloc and HeapAlloc can also be used.

Allocating a Movable Memory Block

```
hMem  = GlobalAlloc(GMEM_MOVEABLE,  1024);
```

To obtain the address of the block, we lock the block by calling the

```
LPVOID lp;
lp  = GlobalLock(hMem);
```

-***it returns a pointer to the block.***

Allocating a Discardable Memory Block

hMem = GlobalAlloc(GMEM_DISCARDABLE|GMEM_MOVEABLE, 1024);

Windows does not discard this type of memory block when it is locked. The GlobalLock increases the lock count only when locking discardable blocks in order to prevent it from being discarded.The GlobalUnlock function decreases the lock count so that the block may be discarded if necessary.

GlobalReAlloc

Changes the size or attributes of a specified global memory object. The size can increase or decrease.

```
HGLOBAL WINAPI GlobalReAlloc(
  __in  HGLOBAL hMem,
  __in  SIZE_T dwBytes,
  __in  UINT uFlags
);
```

hMem - A handle to the global memory object to be reallocated. This handle is returned by either the GlobalAlloc or GlobalReAlloc function.

dwBytes - The new size of the memory block, in bytes. If uFlags specifies GMEM_MODIFY, this parameter is ignored.

uFlags - The reallocation options. If GMEM_MODIFY is specified, the function modifies the attributes of the memory object only (the dwBytes parameter is ignored.) Otherwise, the function reallocates the memory object.

You can optionally combine GMEM_MODIFY with the following value.

Value	Meaning
GMEM_MOVEABLE	Allocates movable memory. If the memory is a locked GMEM_MOVEABLE memory block or a GMEM_FIXED memory block and this flag is not specified, the memory can only be reallocated in place.

If this parameter does not specify GMEM_MODIFY, you can use the following value.

Value	Meaning
GMEM_ZEROINIT	Causes the additional memory contents to be initialized to zero if the memory object is growing in size.

Return Value

- If the function succeeds, the return value is a handle to the reallocated memory object.
- If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

- If GlobalReAlloc reallocates a movable object, the return value is a handle to the memory object. To convert the handle to a pointer, use the GlobalLock function.
- If GlobalReAlloc reallocates a fixed object, the value of the handle returned is the address of the first byte of the memory block. To access the memory, a process can simply cast the return value to a pointer.
- If GlobalReAlloc fails, the original memory is not freed, and the original handle and pointer are still valid.

Reallocating a Memory Block

HGLOBAL GlobalReAlloc(HGLOBAL handle, DWORD newsize, UINT flags)

It is used to change the size and type of the memory block.To change the memory flags for a block, we need to use GMEM_MODIFY flag.

hMemNew = GlobalReAlloc(hMem, 2048, GMEM_MODIFY| GMEM_MOVEABLE);

[**Note**: *MOVABLE or DISCARDABLE blocks cannot be converted into FIXED memory block*.]

GlobalDiscard

Discards the specified global memory block. The lock count of the memory object must be zero.

```
HGLOBAL WINAPI GlobalDiscard(
        HGLOBAL hMem
);
```

hMem - A handle to the global memory object. This handle is returned by either the GlobalAlloc or GlobalReAlloc function.

Return Value

If the function succeeds, the return value is a handle to the memory object. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Although GlobalDiscard discards the object's memory block, the handle to the object remains valid. The process can subsequently pass the handle to the GlobalReAlloc function to allocate another global memory block identified by the same handle.

Locking and Unlocking Memory Block

Locking and Unlocking is used only on Discardable and Movable block. GlobalLock function returns the address of the block specified handle. Returns 0 if lock fails. The function will fail when handle to the Discardable memory which is already discarded is given or any other invalid handle is given. The GlobalLock increases the lock count only when locking discardable blocks in order to prevent it from being discarded.The GlobalUnlock function decreases the lock count so that the block may be discarded if necessary.

After getting the address of the allocated storage block with GlobalLock it will not change as long as the block memory remains lock. We lock the block once at the start of an application and keep it locked the entire time the application is running. Once unlocked there is no guarantee that the next time when the memory is locked same address is locked.

GlobalLock

Locks a global memory object and returns a pointer to the first byte of the object's memory block.

```
LPVOID WINAPI GlobalLock(
        HGLOBAL hMem
);
```

hMem - A handle to the global memory object. This handle is returned by either the GlobalAlloc or GlobalReAlloc function.

Return Value

If the function succeeds, the return value is a pointer to the first byte of the memory block. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

- The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, GlobalLock increments the count by one, and the GlobalUnlock function decrements the count by one. Each successful call that a process makes to GlobalLock for an object must be matched by a corresponding call to GlobalUnlock. Locked memory will not be moved or discarded, unless the memory object is reallocated by using the GlobalReAlloc function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.
- Memory objects allocated with GMEM_FIXED always have a lock count of zero. For these objects, the value of the returned pointer is equal to the value of the specified handle.
- If the specified memory block has been discarded or if the memory block has a zero-byte size, this function returns NULL.
- Discarded objects always have a lock count of zero.

GlobalUnlock

Decrements the lock count associated with a memory object that was allocated with GMEM_MOVEABLE. This function has no effect on memory objects allocated with GMEM_FIXED.

```
BOOL WINAPI GlobalUnlock(
        HGLOBAL hMem
);
```

hMem - A handle to the global memory object. This handle is returned by either the GlobalAlloc or GlobalReAlloc function.

Return Value

- If the memory object is still locked after decrementing the lock count, the return value is a nonzero value. If the memory object is unlocked after decrementing the lock count, the function returns zero and GetLastError returns NO_ERROR.
- If the function fails, the return value is zero and GetLastError returns a value other than NO_ERROR.

Freeing a Memory Block

GlobalFree function is used for freeing the memory with the handle of the memory as a parameter.

GlobalFree

Frees the specified global memory object and invalidates its handle.

```
HGLOBAL WINAPI GlobalFree(
        HGLOBAL hMem
);
```

hMem - A handle to the global memory object. This handle is returned by either the GlobalAlloc or GlobalReAlloc function. It is not safe to free memory allocated with LocalAlloc.

Return Value

If the function succeeds, the return value is NULL. If the function fails, the return value is equal to a handle to the global memory object. To get extended error information, call GetLastError.

Remarks

- If the process examines or modifies the memory after it has been freed, heap corruption may occur or an access violation exception (EXCEPTION_ACCESS_VIOLATION) may be generated.
- The GlobalFree function will free a locked memory object. A locked memory object has a lock count greater than zero. The GlobalLock function locks a global memory object and

increments the lock count by one. The GlobalUnlock function unlocks it and decrements the lock count by one. To get the lock count of a global memory object, use the GlobalFlags function.

- If an application is running under a debug version of the system, GlobalFree will issue a message that tells you that a locked object is being freed. If you are debugging the application, GlobalFree will enter a breakpoint just before freeing a locked object. This allows you to verify the intended behavior, and then continue execution.

E.g.

To free the memory block of 'hMem' handle:

```
hMem=GlobalFree(hMem);
```

More functions related to windows memory management (*Self Study: eXtra*)

- GlobalFlags(…)
- GlobalHandle(…)
- GlobalSize(…)
- GlobalMemoryStatusEx(…)

WoW! You came till the end? That's Great ☺ !!!

'ALL THE BEST'

PURBANCHAL UNIVERSITY

IV SEMESTER FINAL EXAMINATION

LEVEL: Bachelor of Computer Application (B. C. A.)

SUBJECT: BCA255CS, Visual & Windows Programming

Full Marks: 60

TIME: *03:00 hrs* **Pass Marks**: 24

Candidates are required to give their answers in their own words as far as possible.Figures in the margin indicates full marks.

GROUP – A: LONG-ANSWER TYPE

Answer TWO questions. **[2X12=24]**

Q. [1] Write Complete Window Application Program & Modify it in following way: [7+4+1]

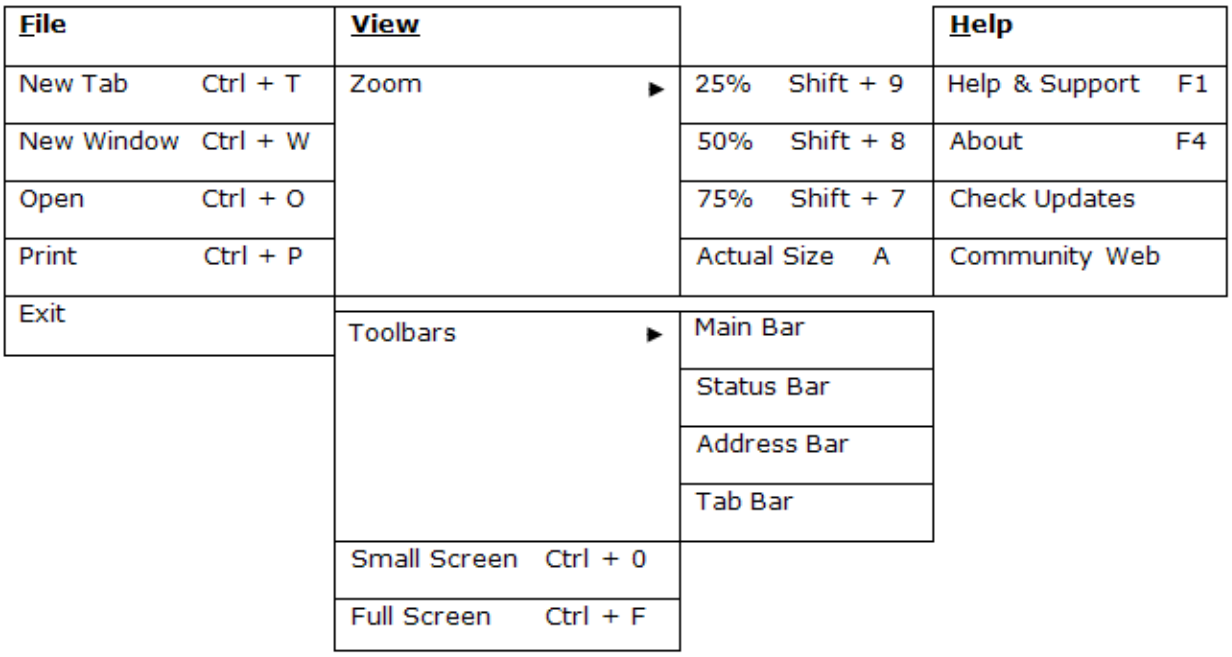
- Left Click on client area for the first time should start drawing then it should draw as the mouse is moved inside the program & stop drawing when left button is clicked again.
- Your drawing should be cleared in right button click.

Q. [2] What is Timer Input? Why the Timer is used in windows programs? Write a program (*windows procedure only*) that will show the following output:

Active Window Time
0 :: 0 :: 0 :: 0

- The time will start when 'SHIFT+S' is pressed, Abort When 'SHIFT+A' is pressed & reset when 'SHIFT+R' is pressed. On exit show the message box with current timer value. [2+2+8]

Q. [3] Create the following Menu and write the necessary code fragment to display the created Menu.



Then write windows procedure to handle the WM_COMMAND messages for the selection of menu Items. Use Message Box to tell the user about menu item that is clicked. If the 'Exit' is selected, the application should be closed. [6+6]

GROUP – B: SHORT-ANSWER TYPE

Answer SIX questions. **[6X6=36]**

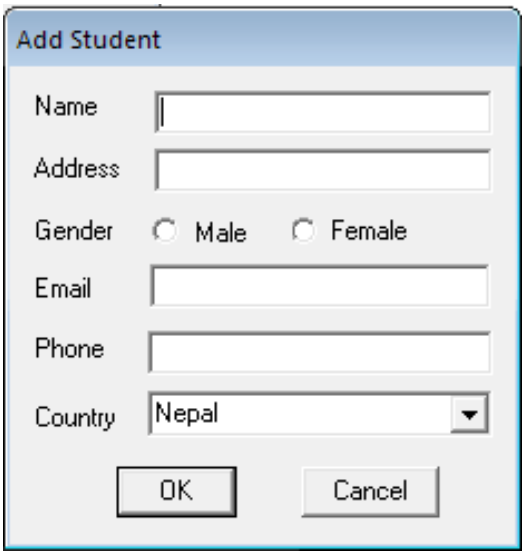
Q. [4] Define Windows in different Prospective. How WIN32 differs from WIN16? Explain with Example. [3+3]

Q. [5] Define 'Display Context' & 'WM_PAINT' message? Why the background of the window repainted on receipt of WM_PAINT Message? [2+4]

Q. [6] What is printing? Describe the overall printing process the windows program follow, with a neat flowchart. [1+5]

Q. [7] What is Memory Management? List the types of Dynamic Memory allocations used in Windows Programming. Why memory management is so important to Windows Programs? Give your view in context with Windows features. [2+1+3]

Q. [8] What is dialog box? Briefly explain its types. Write necessary dialog template which will create following Dialog Box. Use appropriate control style. [1+2+3]



Q. [9] Write necessary windows procedure code fragment to draw the following primitives: [2+2+2]

a. A rectangle with 10px, Dashed, Red boundary and filled with Green. Starting from (50, 35). Height of the rectangle is 110Px and width is 200Px.

b. Green colored, Straight Line with 10Px Width & 300Px Length in following style (*any starting position*):

c. A 4Px, Solid & Red Bezier Curve that goes from these points: (280,270), (140,200), (160,170) & (59,140).

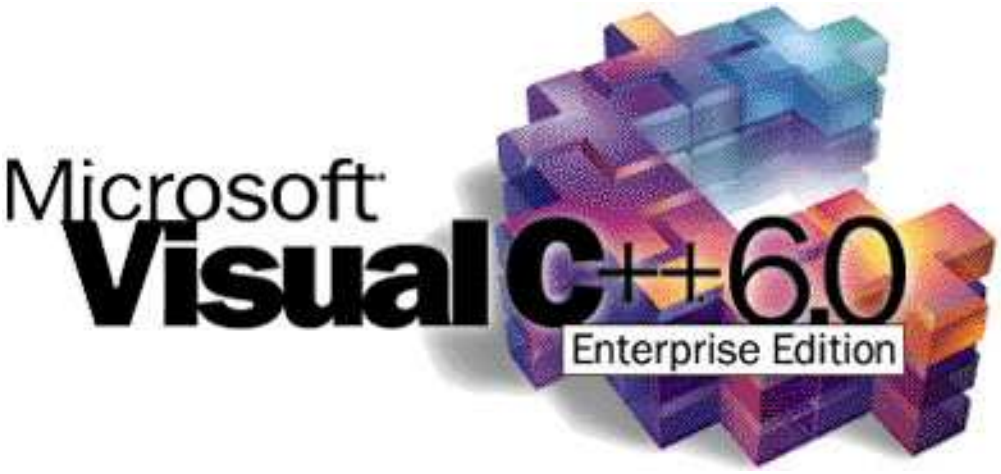
Q. [10] What is Static & Dynamic Linking? How Dynamic Linking is related with Dynamic Link Libraries, Justify with Example. [2+4]

Q. [11] **Write short notes on:** [3+3]

a. PAINTSTRUCT Structure.

b. Defining Color with Explicit RGB

IDE for Lab

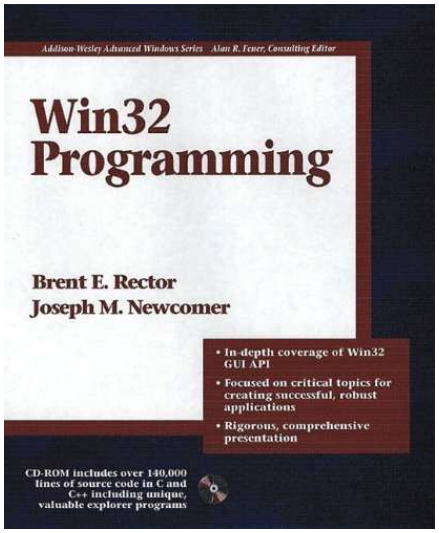


Labsheets & Online Resources

http://www.sarojpandey.com.np/win32

References

f. Brent E Rector and Joseph M Newcomer, Win32 programming, Addison Wesley.



g. Charles Petzold, Programming Windows 95. Microsoft Press. 1996.

h. Richard J. Simon. Windows NT, Win32 API Super Bible SAMS. 1997.

i. Microsoft Documentation on visual programming.

j. MSDN Reference Library.

~