Win32 SDK Notes

Simplified By:
DR. VIJAY D. GOKHALE

Windows Programming Using Visual ' C '.
: INDEX :

(9)	**COMMON DIALOG BOX ::** **• PROGRAM OF OPEN AND PRINT COMMON DIALOG BOX.** **• THEORY.**
(10)	**GRAPHICS CONCEPTS.**
(11)	**MULTIPLE DOCUMENT INTERFACE. [MDI]**
(12)	**THE KEYBOARD ::** **• KEYBOARD AND CARET THEORY.**
(13)	**THE MOUSE.**
(14)	**THE TIMER.**
(15)	**THE CLIPBOARD ::** **• THEORY.** **• SETTING AND GETTING STRING FROM CLIPBOARD.** **• SETTING AND GETTING BITMAP FROM CLIPBOARD.**
(16)	**THE ATOM.**
(17)	**OPEN DATA BASE CONNECTIVITY. [ODBC]**

Date : 14 - 05 – 2002
Day : Tuesday.

1st Chapter - Introduction & History.

In Past, there was Character User Interface, means no use of display driver (Video) to its full extent. Also user was the programmer and programmer was the user. Take example of UNIX. Every UNIX user must be a profound 'C' Programmer.

As time goes on, Operating System was become easy and programmers start writing different utilities to make the **O.S.** more and more easy. Then a different class of people was involved, which use these utilities by unknowing the hidden program. This class was called as "Plain User".

The programming task was targeted for bring to a single thought that making operations more and more easy to user. Obviously, this leads to a universal truth that "making things easy to user is always difficult of making programs for the programmers." There were many limitations on Character User Interface:-

1] No full use of Video Display.
2] In consistent look and feel of the programs.
3] Alphabets and digits can be displayed in only one row form. There was no orientation and variation.
4] A plain user feels it difficult because many inputs are only from keyboard and commands are very long.

So the Information Technology tried out for a Graphical User Interface, and its evolution started in 1970.

¾ GUI (Graphical User Interface) and History of Microsoft Windows.

The GUI was targeted to make full use of video display and making things easier to end-user. The first try was made in mid 1970 by XPARC [**X**erox **P**alo **A**lto **R**esearch **C**entre] in a language Smalltalk.

The Apple Company influences this work and developed **LISA**. This software has not gain popularity but it gives Apple stimulation to develop the first and the best and still going GUI Operating System, **Macintosh**. **Macintosh** developed in January 1984.

Microsoft Incorporation influenced by the work of **LISA** and announced its own GUI operating system in 1983, but launched it 2 years later by naming it **"Windows 1.0"** in November 1985. Then **"Windows 2.0"** was developed in 1987. This system was mainly designed for 80286 Microprocessor and thus also called as **"Windows/286"**. Then as Intel's 80386 was launched, Windows also launched **"Windows/386"**.

Uptil now the O.S. was running in **"Real Mode"** only. Then a major change was made by **Windows 3.0** which was launched on 22nd May 1990 which was using protected mode of CPU instructions. Then in April 1992, **Windows 3.1** was launched, which has many advantages over Windows 3.0, but the most important **Object Linking and Embedding**, (This is a invention of Microsoft) called as **OLE 1.0**. From here, a new era starts (with the help of OLE) called as **IPC** (Inter-process Communication)

(Windows 3.1 was introduced in 1992. Windows NT was introduced in 1993, it is independent on DOS, and it uses 32-bit first time.)

A big step in the development in Windows O.S. is made by launching Windows NT (NT stands for New Technology). This was the first system of Microsoft which provides:-

1) Multi-User.
2) Multi-Threading.
3) Multi-Tasking.

The version up till known were only supportive, only to process based multi-tasking. A very important change was also made in this O.S. that is 32 Bit Flat Addressing.

Though, Windows NT has all operating features, it was mainly used for workstations, and networking for the home and office purpose. Microsoft develops Windows 95 by reducing security, networking concepts of NT. This was launched in 1995. Windows 95 is also termed as "Chicago" and "Windows 4.0". Windows 95 has all good features of Windows NT and all further versions are based on Windows 95 & NT.

e.g. Windows 98 (Based on Windows 95)
Windows 2000 (Based on Windows NT)
Windows XP (Based on Windows NT)

But a major change was made by Microsoft in Windows XP is 64 Bit flat addressing technique.

In between these versions, some commercial versions are introduced with nominal changes such as Windows Me, Windows CE, etc. (Me : Millennium Edition, CE: Century Edition) Whatever may be the development in GUI, the major aim was to provide a consistent interface with the prolog **"What You See Is What You Get"** - WYSIWYG.

¾ Advantage of Windows Operating System : -

1] Graphical User Interface :-

Full use of Video Display, many rectangular areas can be displayed on the screen, each having its own properties and own life called as **"Windows"** . (Definition of Windows). By GUI texts and shapes can be drawn with various formats and with various orientation.

2] Consistent User Interface : -

Windows has consistent look and feel, means if you understand to operate one window application, the learning of operating other window application in easy.

This made possible because all window uses Title Bar, System Menu, Close, Restore, Minimize, Scroll Bars, Areas, Menu Bars, etc.

Date : 15 - 05 - 2002.
Day : Wednesday.

3] Multi-Tasking : -

In contrast to DOS, Windows platform allows multiple programs to run at a time on the screen by sharing CPU time (CPU scheduling or CPU time slicing) . User can switch for one application to other, not even this only, but one application can communicate with other application by sending or receiving data. (Inter-process Communication).

4] Memory Management :-

Windows NT and 95 can use Virtual Memory and thus allowing more code to fit in the memory. This is possible because Windows run in protected mode. Several instances (occurrences) can run simultaneously having separate database but sharing code space. Windows Memory Management also allows an application to start another application (child process) which runs either in the memory space of parent space or in its own memory space. This facility is mainly used by DLL and OLE and COM.

5] Device Independent Graphics Interface :-

Windows application does not directly interact with hardware, means they do not directly interact with video display or printers, instead they call GDI *API*s which in turn communicate with hardware. This insulation or encapsulation allows Windows O.S. and its user to access any type of hardware by just knowing their respective device drivers. Windows O.S. maintains database or for these devices (.inf files) in system registry.

So many hardware like Display Card, Printer, Modem, Sound Card, TV Tuner Card, Video Capture Card can be used just by Plug & Play method.

- **Difference between old Windows O.S. (like 3.1) and New Windows O.S. (like NT, 95.... so on)**

Old Windows O.S. (Windows 3.1)	New Windows O.S. i.e. Windows 95
Differences in concern with the User	
1) There is program manager and file manager.	1) Program manager is replaced by start menu and file manager is replaced by Windows Explorer.
2) Windows 3.1 program can run just fine on Windows 95.	2) Windows 95 program can not run on Windows 3.1.
3) There is no task bar.	3) There is a task bar.
4) The look is clunky.	4) The look is snappy.
5) Windows 3.1 has only old control's radio buttons, check boxes, scroll bars, etc.	5) New controls like toolbar, spin-control, tree-view control, status bar, property sheet, tab control, etc.
6) Windows 3.1 requires DOS to run it. It is not a standalone O.S.	6) Though Windows 95 has DOS, it can start independently. But Windows NT is DOS independent and hence it is totally standalone O.S.
7) If you want to run Windows Program, then you have to switch from DOS to Windows by typing **Win** and pressing a Enter Key.	7) You can run a Windows Program directly through DOS prompt.
8) Windows 3.1 supports only 8 characters long filenames and 3 character long extensions.	8) Windows 95 supports only long character file names and long extensions.
9) Internet, networking and Plug & Play are not well supported in Windows 3.1.	9) Windows 95 gives full support to Internet, networking and Plug & Play.

Differences in concern with Programmers	
10) Windows 3.1 programs can run just fine on Windows 95.	10) Windows 95 can not run on Windows 3.1
11) Windows 3.1 uses 16-bit microprocessor, 16-bit values and 16-bit segmental addressing technique Win 16 *API*.	11) Windows 95 uses 32-bit microprocessor, 32-bit values and 32-bit flag addressing technique Win 32 *API*.
12) Only process based multi-tasking means multi-thread based multi-tasking is not supported.	12) Multi-tasking under Windows 95is under both process based and thread based.
13) Windows 3.1 has multi-tasking of non-primitive type.	13) Windows 95/NT has primitive multi-tasking.
14) Every process has its own and only one message input queue.	14) Every thread has its own input queue. Hence in multi-threading, there will be multiple input queues.
15) *API* is Win 16 *API*.	15) *API* is Win 32 *API*.
16) All *API*s are grouped in 3 main files – user.exe, GDI.exe, krnl386.exe.	16) All *API*s are grouped in 3 dynamic link libraries (.DLL) which are user32.dll, GDI32.dll & kernal32.dll.
17) Though these files are EXEs, they are not executable, means if you double-click them or execute them, **nothing** happens.	17) These are dynamic link libraries and non-executables.

Meanings :-

¾ **Primitive and Non-Primitive multi-tasking.**

Windows 3.1 offers non-primitive multi-tasking, means when more than one program are running, Windows 3.1 does not use CPU scheduling, means does not use CPU timer to allocate specific time to running programs, instead one program itself has to give up the controls to other program so that another program can run. Obviously, unless one program give up the control, another program can not run.

Whereas Windows 95 offers primitive multi-tasking means more than one program are running under Windows 95.

Note :- That control is completely in hands of Windows O.S. (it is not in hands of application program). Thus, Windows O.S. makes time slices by CPU scheduling and allocate a specific time slice to the currently running program. When time slice of one program finishes, it sleeps and the time slice of another program awakes. This happens very fast and thus looks like multiple programs are running, but actually it's just like a illusion. This becomes possible due to multi-threading capacity of Windows NT and 95.

¾ Dynamic Link Library (DLL)

DLL is short form of Dynamic Link Libraries. DLL is a binary code file which provides library of functions, classes, objects, macros, resources. All *API* s of NT and 95 are also written in DLLs. When we write a program code, during compilation the code is checked only for errors, but library - functions which we used in our program are not added in our code. After completion of compilation, we link our code to the libraries and then there are 2 options :-

i) The code of library function into your compile code will increase size of final EXE. This type of linking is called as "Static Linking". Generally ".lib" files are used for static linking. In DOS environment we have to use this type of linking.

ii) During linking the code of library function is not added in compile code, keeping size of final EXE shorter and when we execute program, then at runtime, library files are linked to our executable files. For such type of linking or run-time linking, DLL files are used. This method is used mainly for Windows O.S. Another advantage of making DLL, is you can break your code in small pieces making DLLs of each and even other programmers can use these DLLs as these are binary files, there is no fare code security. When you want to upgrade your program, you can upgrade the mail file without touching the DLL i.e. there is no need of recompilation of DLL.

¾ What is *API* ?

API is short form of "Application Programming Interface", which is a set of several hundreds of *API* through which you can directly talk with Windows O.S. These *API*s are one of core files of Windows and any installation must include these 3 files of *API* as stated before. To use their functions, you should only know the 3 things :-

1) Name of function.
2) Return Type.
3) Parameters.

Windows Programming is possible only with these *API*s. These *API*s barely written in 'C'. Though, it looks that MFC or VC++, VB, Visual FoxPro, Delphi creating Windows application without *API*, they internally calls *API*s. There are 2 types of *API* :-

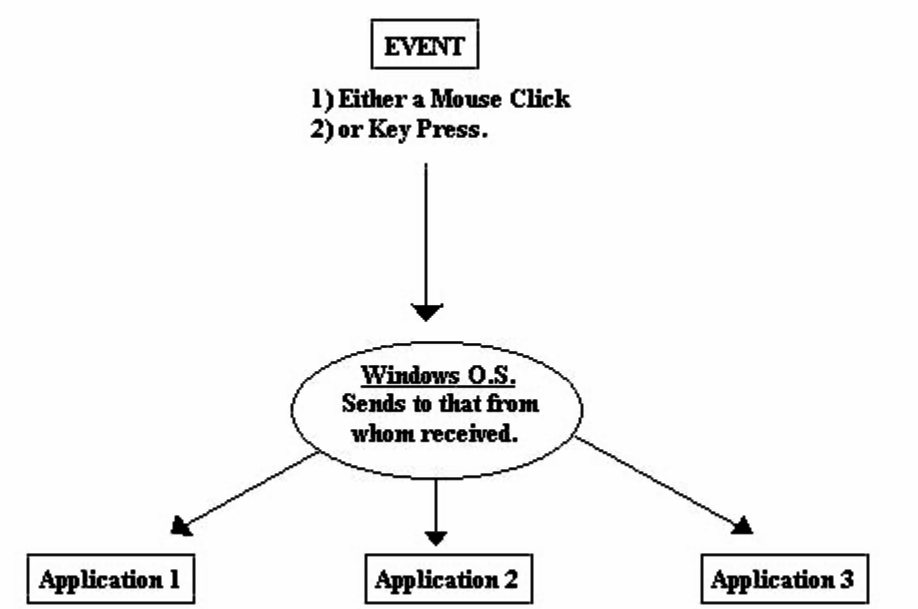
1) Win 16 *API* : Windows 3.1.
2) Win 32 *API* : Windows NT, 95.

¾ What is most important, major difference between DOS program and Windows Program ?

A DOS program has full control over CPU. Hence mostly CPU power is wasted. A DOS program calls O.S. whenever it needs O.S. When a DOS program runs, O.S. has no duty and just seats back.

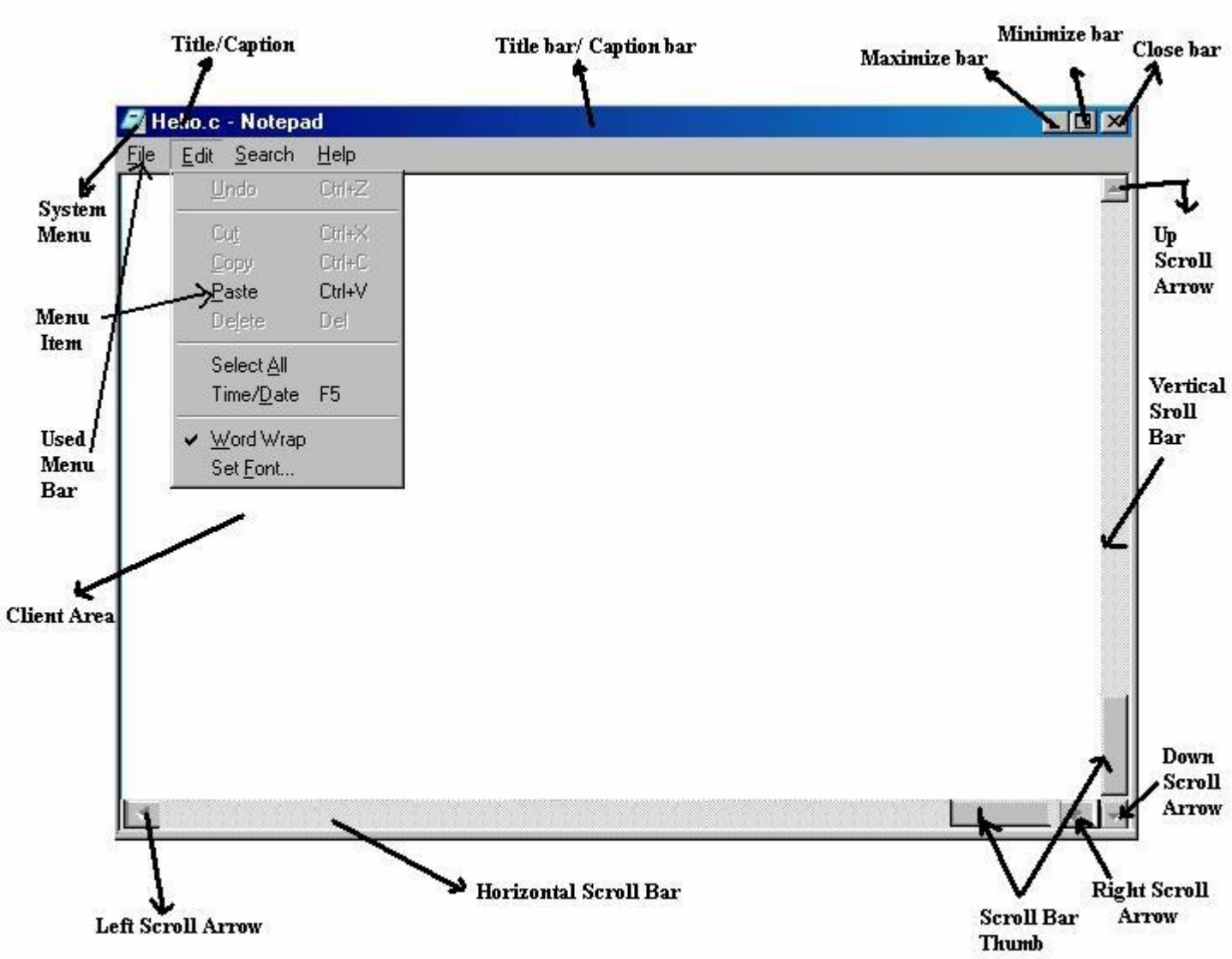
But Windows program is completely different. The whole control of program execution is never in hand of program (like in DOS), but in hands of O.S. When a mouse is clicked on a Window or a key is pressed then an event is generated and received by O.S., which in turn sends message to the program to take appropriate action. This is called as "Even Driven Architecture".

Diagram of Event Driver Architecture :



means, in DOS, O.S. is saying "Call me whenever you need me". In Windows O.S. is saying "Don't call me, I will call you".

Diagram of General Structure of Windows :



Date : 20 - 05 – 2002
Day : Monday.

First Window Program

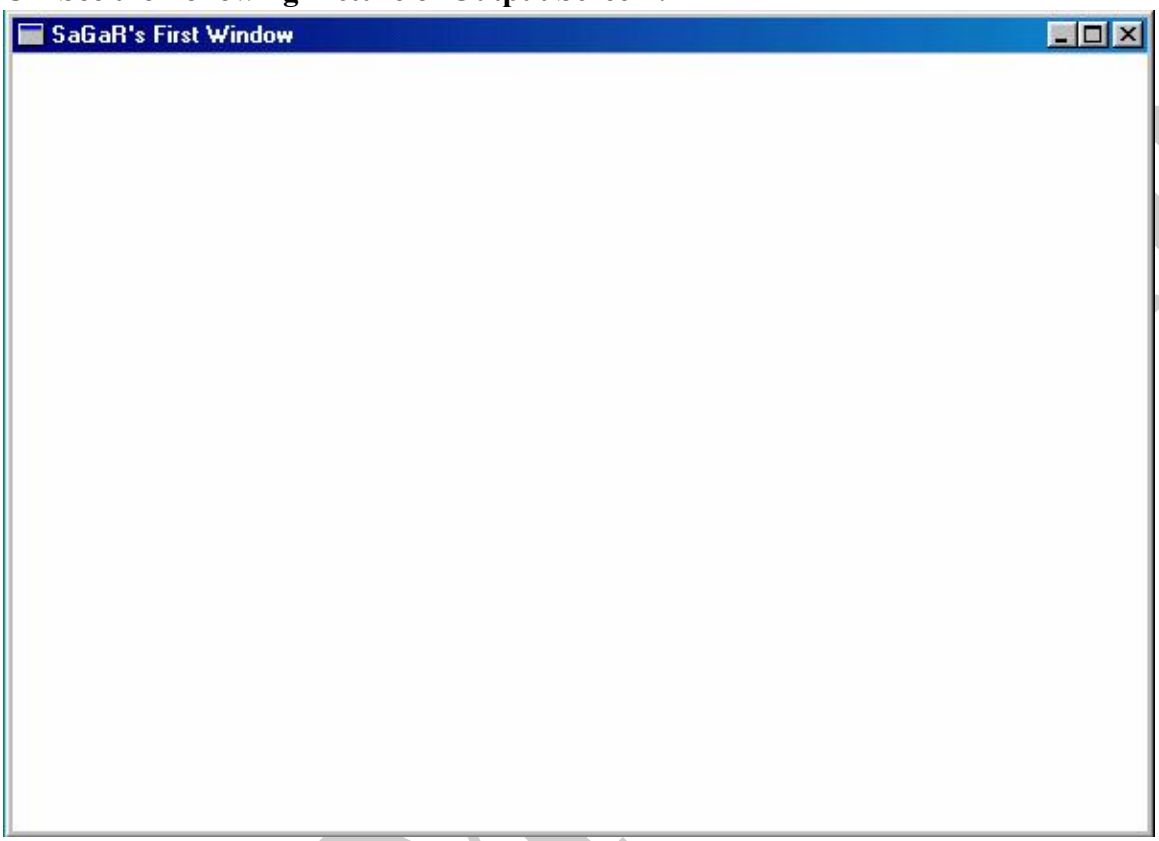
```
#include < windows.h >        // Step 1

LRESULT CALLBACK WndProc (  HWND,
                            UINT,
                            WPARAM,
                            LPARAM ) ;  // Step 2

int WINAPI WinMain ( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpszCmdLine,
                     int nCmdShow ) // Step 3

{
        // Local Variables

        WNDCLASSEX wndclass ; // Step 4
        HWND hwnd ;
        MSG msg ;
        char AppName[ ] = "Windows" ;

        // Code

        wndclass.cbSize = sizeof ( WNDCLASSEX ) ;
        wndclass.style = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc = WndProc ;
        wndclass.cbClsExtra = 0 ;
        wndclass.cbWndExtra = 0 ;
        wndclass.hInstance = hInstance ;
        wndclass.hIcon = LoadIcon ( NULL, IDI_APPLICATION ) ;
        wndclass.hCursor = LoadCursor ( NULL, IDC_ARROW ) ;
        wndclass.hbrBackground  =  ( HBRUSH )  GetStockObject  (  WHITE_BRUSH  )  ;
        wndclass.lpszClassName = AppName ;
        wndclass.lpszMenuName = NULL ;
        wndclass.hIconSm = LoadIcon ( NULL, IDI_APPLICATION ) ;

        // Registration

         RegisterClassEx ( &wndclass ) ; // Step 5
```

```
	// Creation of Window

	hwnd = CreateWindow ( AppName,
				"SaGaR's First Window",
				WS_OVERLAPPEDWINDOW,
				CW_USEDEFAULT,
				CW_USEDEFAULT,
				CW_USEDEFAULT,
				CW_USEDEFAULT,
				NULL,
				NULL,
				hInstance,
				NULL ) ; // Step 6

	// Displaying of Window

	ShowWindow ( hwnd, nCmdShow ) ; // Step 7
	UpdateWindow ( hwnd ) ; // Step 8

	// Message of Loop i.e. Step 9

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

	return (msg.wParam);


	// End of WinMain ( )
}

	// Window Procedure (Window Function) : Step 10

	LRESULT CALLBACK WndProc ( HWND hwnd,
					UINT iMsg,
					WPARAM wParam,
					LPARAM lParam )
	{

	// Code

	switch ( iMsg )
		{
			case WM_DESTROY :
			PostQuitMessage ( 0 ) ;
			break ;
		}
	 return ( DefWindowProc (hwnd, iMsg, wParam, lParam ) );
} // End of Complete Program
```

Double Click the icon below to see the output of the above program : -

First Window.exe

OR See the Following Picture of Output Screen :-



¾ **Step wise dissection of above program**

™ **Step 1 : #include <windows.h>**

This is standard header file for all windows application programs. Every windows **C** program must include this as the first executable statement of the program. This file contains 3 important things :

1) All predefined constants as MACROS like
CS_HREDRAW, **CS_VREDRAW, IDI_APPLICATION**, **WHITE_BRUSH**, **IDC_ARROW**, **WS_OVERLAPPEDWINDOW**, etc.

2) New data types as typedef of old data-types.

3) Prototypes of Windows *API*.

If you see this file in any editor, you'll see some other header files included in this header file, but those also have the same above 3 things.

Note *:- In old Win System Programming model, 2 other pre-processor directives were used, those were :-*

a) #define strict :- which was used before compiler to do strict type-checking, means though hwnd and hInstance are internally unsigned integers, you can't say :
HWND hInstance ; or **HINSTANCE hwnd ;**

b) #pragma argused :- If we see **WinMain ( )** code, we never used hPrevInstance and lpsz. In normal circumstances, compiler will generate warnings for those unused parameters. But if above pre-processor directive is used, such irritating warning can be suppressed.

™ **Global Prototype Declaration of Window Procedure Function :- (i.e. Step 2)**

Windows O.S. is such an O.S., which takes control of your program. In other words, your program doesn't call O.S., but O.S. call your program. Another important thing is that Windows itself is a Program (i.e. Win.com). Now to allow this O.S. program to call your program, there must be an entity which is visible to O.S. and which will allow O.S. to call it.

To make your program visible to O.S., you declare such a function calls as Window Procedure and declare its prototype globally.

At the same time, your **WinMain ( )**, also can call it. Besides this visibility, you should make your Window Procedure callable by O.S. and for this **CALLBACK** identifier is used. (**_far_pascal** is a typedef of **CALLBACK** & **WINAPI**).

In other words you can say that your Window Procedure is such a part of your program by which program can communicate with O.S. and most importantly O.S. can communicate with program.

This function returns **LRESULT** (long) type value, hence in prototype, we use **LRESULT**. The name of the function is not important. You can use any name here, but as a convention **"WndProc"** name is used. There are 4 parameters to this function.

1) Handle of the window - **HWND** type.
2) A 32 bit message number of **UINT** type.
3) A word parameter of **WPARAM** type.
4) and another long integer parameter of **LPARAM** type.

These 2 parameters are special or extra information of the message (of 2nd parameter).

Date : 22-05-2002
Day : Wednesday.

- **Due to CALLBACK identifier, O.S. can call your program, but how WinMain ( ) will call WndProc ( ) ?**

:- If you see the code of **WinMain ( )**, you'll found a while loop called as messaging. When **WinMain ( )** wants to call the **WndProc ( )** [in posting of messages which we'll see shortly] if enters into this loop & **DispatchMessage ( )** indirectly calls the **WndProc ( )**. Now again a question arises **"How DispatchMessage ( ) knows the name of your Window Procedure i.e. WndProc ( ) ?"**

DispatchMessage ( ) first checks your **WND** structure declaration, in which there is one member **lpfnWndProc**, which is nothing but the pointer to your Window Procedure. You assign name of your Window Procedure to this member and thus **DispatchMessage ( )** knows it.

Before studying further lines, we'll first get familiar with same new concepts :-

1) Derived data-types.
2) Hungarian Notation.
3) What is Handle ?

1) Derived data-types :- By using **typedef** facility of 'C'. Windows O.S. designers derived new data-types for -

i) Meaningfulness.
ii) To avoid mentioning of multiple keywords in single line.

iii) To create a complete different data terminology based on the old data types.

In short you can say that all the unknown uppercase words in the above program (except constants), like **LRESULT**, **CALLBACK**, **HINSTANCE**, etc. are nothing but the typedef based on old 'C' data types.

Some important typedefs are as follows : -

LRESULT	:-	_far_pascal
WINAPI	: -	_far_pascal
HINSTANCE	:-	HANDLE
HANDLE	:-	UINT
UINT	:-	unsigned integer.

Same applies to all the uppercase word beginning with "H", like **HWND**, **HDC**, **HGLOBAL**, **HBRUSH**, **HFONT.**

PSTR	:-	(pointer to string) char *
LPSTR	:-	(long pointer to string) char _far *
WORD	:-	unsigned integer (same like handle)
DWORD	:-	(double WORD) unsigned long.
WPARAM	:-	WORD
LPARAM	:-	LONG
LONG	:-	long
LPCTSTR	:-	long pointer to constant NULL terminated string (typedef is : **const char far ***)
BOOL	:-	unsigned integer.

2) Naming the identifiers (Hungarian Notation) :-

Identifiers are any user-defined names. Windows Programming follows conventional rules for naming identifiers. Programmers are not forced to follow them strictly, but they are encouraged to do so to increase the meaningfulness of source code. This naming convention is given by the Microsoft's respectful programmer **"Charles Simonyi"**. His birth-place was Hungary and hence this method of naming identifiers is called as "**Hungarian Notation**".

By this convention :-

1) The identifier name should begin with the lower case letter or with lower case letter of data type.

2) If the identifier name is made up of 2 or 3 words, then first letter if each word should be capital. These rules are followed by variable names, structure names, etc.

3) For function name, first letter should be capital and every first letter of each word should be capital.

For this convention, the 'data-type prefixes' are standardized as follows :-

01) **c** : character e.g. **cAppName**
2) **by**: BYTE (unsigned char)
03) **n** : short e.g. **nCmdShow**
04) **i** : integer e.g. **iMsg**
5) **x**: int used as X axis co-ordinate.
6) **y**: int used as Y axis co-ordinate.
7) **cx**, **cy** : Count from X axis, Count from Y axis.
8) **b**: BOOL (Boolean : typedef is : unsigned int)
09) **l** : LONG (long) e.g. **lParam**
10) **w** : WORD (unsigned short) e.g. **wParam**
11) **dw** : DWORD (unsigned long)
12) **fn**: function
13) **lp** : long pointer e.g of 12 & 13 is **lpfnWndProc**
14) **p**: pointer
15) **s**: string
16) **sz** : string terminated by 0 character e.g. **lpszCmdLine**
17) **cb** : count of bytes e.g. **cbSize**, **cbClsExtra**, **cbWndExtras**
18) **f**: bit flag
19) **h**: handle
20) **pt**: point on co-ordinate system (data-type will be structure)
21) **rgb** : RedGreenBlue (COLOREF structure variable for Red Green Blue)
22) **br** : brush e.g. **hbrBackground**

If we observe above conventions, we know that understanding is more clearer, we have twelve *API*s, one **WinMain ( )** , and one Window Procedure in our first program code.

All these name follow Hungarian Notation having their names beginning with capital letters and when they are multi-word format, then first letter of each word is also capital.

3) What is handle ?

:- Handle is 32 bit number which is used to reference an object of window like menu, icon, brush, window, hwnd, Device Context. It is not a pointer, but it can be indirectly referenced as a pointer, means you can not directly referenced as a pointer by hwnd, because it is a number and not the address. Handle is very important concept in Windows Programming, because many *API*s either returns handle or take handle as their parameter.

™ **WinMain ( ) (i.e. Step 3)**

This is just like **main ( )** in **C** and **C++**. Every windows program must have this function as a starting body except all which is DllMain ( ). This function has four parameters. First two are handle to instances of the program. **First parameter** is Present Instance and **Second parameter** is Previous Instance.

In 16 bit Windows Programming (Windows 3.1) there were non-primitive multi-tasking. Hence, hPrevInstance always have some non-zero value, but from Windows 95 and above, there is primitive multi-tasking and hence hPrevInstance is always NULL. Then too, it is kept in WinMain ( ) for backward compatibility, making easier to part 16 bit program to new this 32 bit platform. **The third parameter** is NULL terminated string pointer **lpszCmdLine, which** is for any command line argument to your program. Such a command line argument can make programs run :- 1) from command prompt by giving complete command line. 2) from run option.
3) when parent process executes child process by giving command line argument to child process otherwise (means, if your program does not have command line), then lpszCmdLine will be assigned to NULL by O.S.

The fourth parameter is **nCmdShow, which is of integer type,** and this parameter specifies how the application's window should be display (to O.S.).

If no value is specified by you, then by default Windows O.S. say **SW_NORMAL** value of this parameter.

All four parameters can be specified by programmer in code of **WinMain ( )**. You can specify values of these parameters, but those who passed to **WinMain ( )** only for Windows O.S.

™ **WNDCLASSEX Structure (i.e. Step 4)**

This is a structure. Don't misunderstand it with C++ class. No connection with C++. In class, same structure named as **WNDCLASS** with 10 members, but new version adds two extended members (hence **EX** is post fixed) making member numbers to twelve. This structure defines basic characteristics of your application's window. The datatypes and members are as follows :-

1) cbSize :- unsigned int.

This is of **UINT** type, which is assigned by **sizeof** either **WNDCLASSEX** structure type of variable of structure type.

2) cbStyle :- UINT type.

Common styles are :- **CS_HREDRAW** & **CS_VREDRAW**

Here, word **CS** means Class Style. HREDRAW & VREDRAW for horizontal & vertical redraw. It is used when window needs to be repainted.

Date : 30-05-2002
Day : Thursday.

3) lpfnWndProc :-

This is a long pointer to Window Procedure. As name of the function is itself its address, we'll assign name of our Window Procedure to this member. Our Window Procedure's name is **WndProc**, which we will assign to this member. This name may be any user-defined name.

4) cbClsExtra :-

This member is for Extra Information about the Class. Usually this member is assigned to zero.

5) cbWndExtra :-

This member is for Extra Information about the Window. Usually this member is assigned to zero.

6) hInstance :-

This member is the handle of the current instance of the program. In our **WinMain ( )** we have hInstance as our current instance of the program. Hence we will assign this handle to this member.

7) hIcon : -

This is the handle of the main icon of the window. We will call an *API* **LoadIcon ( )** to get the handle of the icon and then assign to this member. Prototype of **LoadIcon ( )** is :-

HICON LoadIcon (HINSTANCE, LPCTSTR) ;

8) hCursor :-

This is the handle of the cursor of the window. We will call an *API* **LoadCursor ( )** to get the handle of the cursor and then assign to this member. Prototype of **LoadCursor ( )** is :-

HCURSOR LoadCursor (HINSTANCE, LPCTSTR) ;

9) hbrBackground :-

This is the handle of the window background. We will call an *API* **GetStockObject ( )** to get the handle of the window background and then assign to this member. Prototype of **GetStockObject ( )** is :-

HGDIOBJ GetStockObject (int) ;

We pass **WHITE_BRUSH** integer value to this function, which returns handle of the type **HGDIOBJ**. But to point the background of our window, we want the **GDIOBJECT** Brush. Hence we will typecast the return value to **HBRUSH** type.

To see the other values of brushes like **WHITE_BRUSH**, see **MSDN**.

10) lpszClassName :-

This is the application name which is may be the any name defined by user as a string. Hence in variable declaration we should have such a string like

char szAppName [ ] = "Windows" ;

Now we will assign szAppName to this member of a structure.

11) lpszMenuName :-

Our first program does not have any user-defined name, hence this member will be assigned to NULL. When we will create our own menu in resource file, then this member will be assigned to the name of our menu.

12) hIconSm (Sm stands for Small) **:-**

This is the handle of the small icon of the window. We will call an *API* **LoadIcon ( )** to get the handle of the icon and then assign to this member. Prototype of **LoadIcon ( )** is :-

HICON LoadIcon (HINSTANCE, LPCTSTR) ;

If you include this member, then the reduced (shrinked) size of your main icon will be displayed in the task-bar when you minimize your program.

The necessity of **WNDCLASSEX** structure is for the creation of mainframe window. If you want to create small child windows like button, bar, box, then there is no need of **WNDCLASSEX** structure. These windows are pre-defined. You just have to pass respective class name of your required window as first parameter of the **CreateWindow ( )**. Thus almost in all window programs, **WNDCLASSEX** structure is declared only once.

™ **Registration of WNDCLASSEX Structure (i.e. Step 5)**

As this a mainframe window, you have to register **WNDCLASSEX** structure variable to the O.S. The *API* is **RegisterClassEx ( )**. The prototype of this function is :-

atom RegisterClassEx (CONST WNDCLASSEX *) ;

This function requires address of above structure variable as its only parameter. Our structure variable is **wndclass**. Hence we will pass **&wndclass** to this function.

™ **Creation of the Window. (i.e. Step 6)**

To create the window, we will use the *API* **CreateWindows ( )**. This function takes eleven parameters and returns handle to the window. To get its return value we must have a variable of **hwnd** type. Hence in variable declaration we wrote a statement **HWND hwnd ;**

The eleven parameters of this function is as follows :-

1st Parameter is Name of the class :-

In our program **szAppName** string. For child windows this parameter will be Pre-defined standard window class name.

2nd Parameter is Title of the window :-

This is a user-defined string enclosed in double-quotes. This name is going to appear in the Caption bar (Title bar) of your window.

3rd Parameter is Style of the window :-

As our program's window is going to be top-most window on the desktop, this parameter is usually a constant **WS_OVERLAPPEDWINDOW**. For further constants, see **MSDN**.

4th, 5th, 6th & 7th Parameters :- are the position of the window on the screen.

4th parameter is x co-ordinate of window.
5th parameter is y co-ordinate of window.
6th parameter is width of window.
7th parameter height of window.

for default values of this parameter, we used **CW_USEDEFAULT**.

8th Parameter is Handle of the Parent Window :-

When we use this function to create our first mainframe window, then there is no parent window, hence this parameter is NULL. But for child window this parameter will be **hwnd**.

9th Parameter is Handle of the Menu :-

Our first program does not have any user-defined menu, hence this parameter will be NULL.

10th Parameter is Current Instance Handle :-

Obviously this parameter will be **hInstance**.

11th Parameter is called as Window Creation Parameter, which is usually NULL. For more information about this parameter, see **MSDN**.

Prototype can be written as follows :-

HWND CreateWindow (LPCTSTR, LPCTSTR, DWORD, int, int, int, int, HWND, HMENU, HINSTANCE, LPVOID) ;

You may use **CreateWindowEx ( )**, it has one extra parameter as the first parameter. This parameter is for Extended Style of **dword** type. For more information about this parameter, see **MSDN**.

™ **Showing of Window (i.e. Step 7)**

CreateWindow ( ) creates the window in memory. To show the window, you have to use the handle returned by **CreateWindow ( )** in **ShowWindow ( )** *API*.

The prototype of **ShowWindow ( )** is :-

BOOL ShowWindow (HWND, int) ;

We will use **hwnd** variable as first parameter, **nCmdShow** variable as the second parameter. For other values, see **MSDN**.

Here, **nCmdShow** is 4th parameter of **WinMain ( )** , which is a command line argument. We start our application by double clicking or pressing ENTER on the program's executable's name, means obviously we don't explicitly pass any command line arguments. Then too, window appears on the screen with some definite size. This shows that O.S. itself passes default normal size of our window as the 4th parameter. This O.S. defined value is **SW_NORMAL**.

You can over-ride this default value by other **SW** values, e.g. **SW_MAXIMIZE**, **SW_MINIMIZE** (For other values of this parameter, see **MSDN**) as the second parameter of **ShowWindow ( )** *API*.

™ **Painting the Window on the screen (i.e. Step 8)**

Above function displays the window on the screen, but yet it is not visible. Because it is still not painted. To paint it, we will call **UpdateWindow ( )** *API*, which has only one parameter handle of our window. The prototype is :-

BOOL UpdateWindow (HWND) ;

Date : 31 - 05 – 2002
Day : Friday.

Instance

There are three words concerned with a program :-

1) **Program.**
2) **Process.**
3) **Instance.**

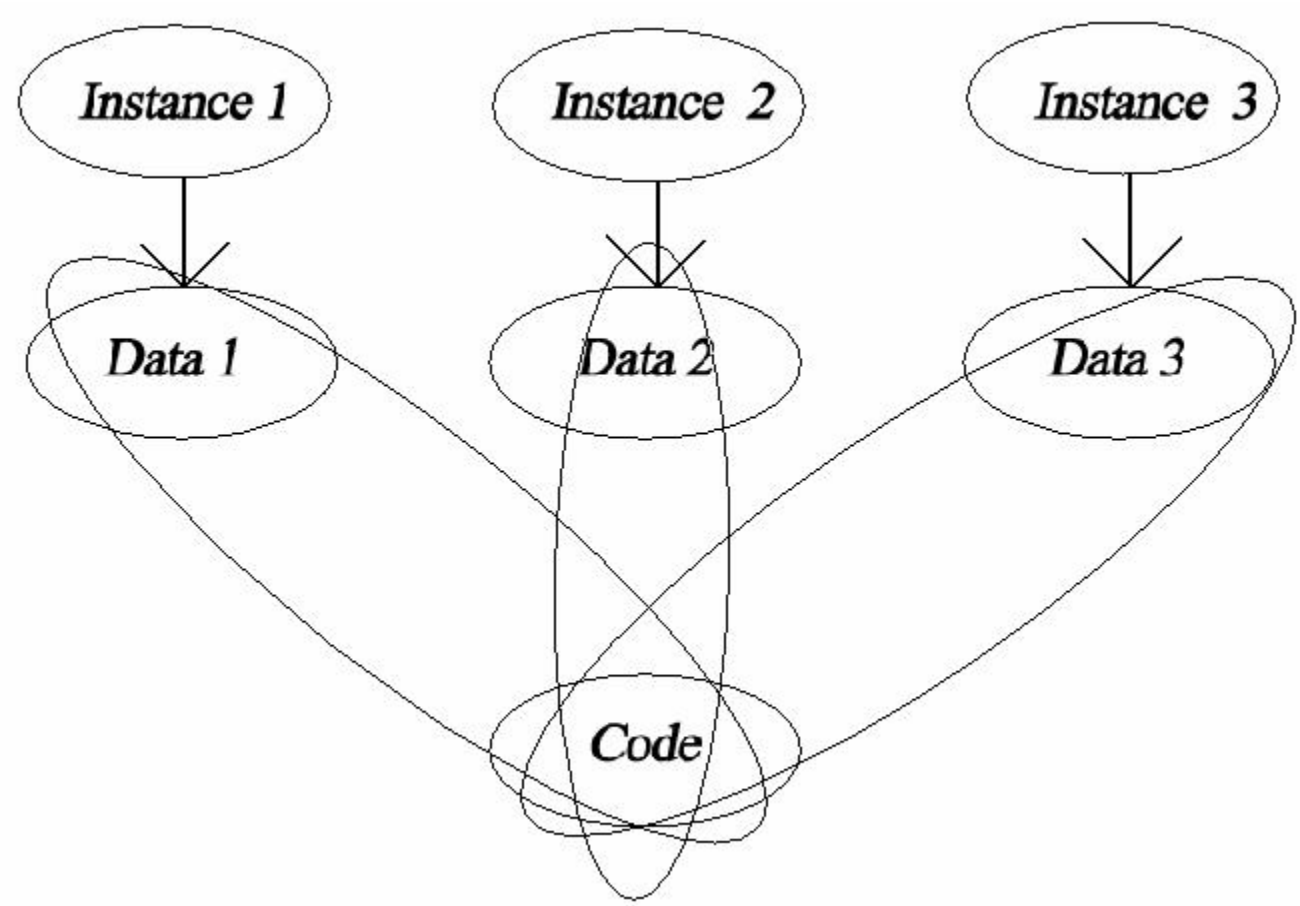
1) **Program :-** Executable source code on the disk is called as a program.

2) **Process :-** When you start a program, (either by typing its name & pressing Enter or double-clicking on it) it runs. This running entity is called as Process.

3) **Instance :-** When a process is running, then it is called as the occurrence or a instance of a program. Without destroying the running process if you again start a same program then it will be a second instance. The active instance will called as Current Instance and the past instance is called as Previous Instance. As we've seen before, in 16-bit Windows (i.e. Windows 3.1) the previous instance is a definite value, but in Windows NT & Windows 9x, this previous instance is always NULL. Still it is kept in the declaration of **WinMain ( )** for a backward compatibility.

   Concept of instance shows that, there may be the multiple instances of the same program running simultaneously, having different data-storage spaces but sharing same code, means if you've **int i**, in the program then every instance will reserve two bytes for **i**, but the code will be same to all.

Below Diagram can explain very well "Concept of instance shows that, there may be the multiple instances of the same program running simultaneously, having different data-storage spaces, but sharing same code."



In Windows Programming Instance is identified by handle called as HINSTANCE.

™ **Message Loop & Window Procedure. (i.e. Step 9 & 10)**

In the difference between DOS & Windows Program, we'd seen that Window's Program control is not at all in the hands of program, but completely in hands of Windows O.S., means O.S. determines :

a) Which program is running.
b) What things are happening with the program i.e. Events.
c) What responses should O.S. give to those events.

It is said that Windows Programming Model is **Even Driven Programming Model (EDP)**. This is just like **big brother's watching**.

Some questions are left unanswered when we first think about EDP :-

1) If user presses a key, who handles this operation ?
2) When user clicks a mouse button, who decides which window is going to receive that click ?

3) If two or three programs running, simultaneously wanted to access the printer, then who decides which should go first ?

All these questions have common answer, there must be a super program or big brother who is going to handle all these things & that Super Program is Windows Operating System.

When a window program starts, O.S. takes full control of that program in self-hand & passes control to the program only when any event occurs with that program.

(See the figure in the topic of Differences between DOS and Window's Program.)

Event Driven Programming Process

Suppose there are three windows on screen, means either three different applications are running or three instances of one application are running.

Now an event like a mouse click or a key press made by the user.

This event is received by Windows O.S. with following details :-

1) For which of the three windows, the event had occurred. O.S. identifies that window by its handle (**HWND**).
2) Event is translated into a message called as Window Message. There all in all about 200 messages, all are 32-bit integers & are defined in **windows.h** file with pre-fixing of every message name with **WM_** (e.g. **WM_DESTROY**).
3) Extra information about message, which is unsigned integer type, means of **WPARAM** type.
4) Another information about the message, which is of long integer type, means of **LPARAM** type.

(**3** & **4** are used for extra information about the message and hence called as **"Message Parameters"**.)

5) X & Y co-ordinate of the window for which the event is occurred, these co-ordinate taken in a structure of a point type.

```
struct point
{
        long x;
        long y;
};
```

6) The time, when the O.S. is going to dispatch the message.

After getting all this information about the event, Windows O.S. dispatch the corresponding

-message with respect to occurred event only to that window for which the event was generated.

In other words, we can say that Windows O.S. communicates with the program of that window, means obviously in that program there must be an entity which is visible to O.S. and O.S. can communicate with it. That entity is called as Window Procedure or Window Function. (In our first program, the name of this Window Procedure is **WndProc ( )**)

To communicate with this Window Procedure, Windows O.S. calls it. To allow this the programmer must make Window Procedure as **CALLBACK** & globally.

Windows O.S. communicates with this Window Procedure by dispatching message to it (**WM_**). This dispatching is done by two ways :-

1) By sending of a message. 2) By posting of a message.

1) Sending of a message :-

This is just like making a phone call and thus journey of these messages is very fast. Those messages are sent which has to be dealt very quickly. Out of two hundred messages, about 190 messages are sent by this type.

e.g. **WM_CREATE**, **WM_SIZE**, **WM_MOVE**, **WM_DESTROY**, etc.

The process of sending is as follows :-

Windows O.S. calls Window Procedure by posting four parameters to it

1) Handle of the window, to which message is to be passed (**HWND hwnd**)
2) Message (**UINT, uInt**)
3) Extra word parameter (**WPARAM**, **wParam**)
4) Extra long parameter (**LPARAM**, **lParam**)

In other words, we can say that while sending a message, Windows O.S. calls Window Procedure directly.

2) Posting of a message :-

This is just like mailing a letter. Obviously, it takes more time that making a phone call. Those messages are posted which has to be translated to a lower level code like Keyboard-messages, Mouse-messages, Graphic Video Card-messages, etc. In the words of Window Messages, these messages are WM_KEYDOWN, WM_KEYUP, WM_LBUTTONDOWN, WM_RBUTTONDOWN, WM_PAINT, WM_QUIT, etc.

The process of posting message is slightly different & difficult than the process of sending of message, which is as follows :-

Window O.S. puts the message in data- structure called as Message Queue. Now it is the duty of the application to retrieve the respective message from the message queue. As queue is going to be added by more & more messages, the application must be ready to get all messages, means in the words of programming we have to use a Loop. The message retrieving API is **GetMessage ( )** whose prototype is

BOOL GetMessage (MSG *, HWND, UINT, UINT);

1st parameter is address of **MSG** structure variable. When we pass it, it is empty or garbage. But when **GetMessage ( )** returns this structure is filled with the six point information of the retrieved message (as seen before).

The 2nd parameter is handle of the window, we made it NULL, to get all messages from the message queue.

Note that every application or every instance has its own message queue in main memory maintained by O.S.

3rd parameter is 1st message in the queue (i.e. Minimum value of the range of message).

4th parameter is last message in the queue (i.e. Maximum value of the range of the message).

Date : 06 – 06 – 2002
Day : Thursday.

GetMessage ( ) retrieves message related information in **MSG** type pointer (i.e. msg). This structure has six members as explained before (i.e. handle of the window, message value, WPARAM, LPARAM, point, time). While doing this, it gets all messages of current process because **hwnd** parameter as NULL.

Also we don't specify any range by making 3rd & 4th parameter zero, which also helps in retrieving all messages.

Retrieved message is then passed to **TranslateMessage ( )**, which converts virtual key-codes to character-codes on low level. Translated or non-translated messages ultimately passed to **DispatchMessage ( )**, which internally calls Window Procedure i.e. **WndProc ( )**.

By all about discussion of sending and posting messages, we came to a very important conclusion that whether message is sent or posted, it ultimately passed to Window Procedure.

Now, some important questions may arrive, "**Then how GetMessage loop is exited ?** ", as explained before for all messages except **WM_QUIT**, **GetMessage ( )** function returns non-zero value, hence loop continues. When user selects close from system menu or selects close box, then windows sends **WM_DESTROY** message to Window Procedure. Every window program must process **WM_DESTROY** message by **'case'** statement in **switch** construct. Obviously, when this message is processed by Window Procedure a very important *API* function is called, **PostQuitMessage ( )** by passing zero to it. This function tells Windows O.S. to post **WM_QUIT** message to the application by putting it in the application's message queue. When **GetMessage ( )** retrieves **WM_QUIT** from the queue, it returns zero and exits the loop. Also not that at the same time, it puts the zero of the **PostQuitMessage ( )** as the **wParam** member of the **MSG** structure which is used further as return value of **WinMain ( )** (i.e. **msg.wParam**) making exit status of application successful.

Another question is that **"Are sending & posting method runs concurrently ?"** in other words **"Are message loop & WndProc ( ) runs concurrently ?"**

The answer is NO. Because when a message is posted in message loop **DispatchMessage ( )** does not return until **WndProc ( )** completes the processing of posted message.

"What are the different return conditions of GetMessage ( ) ?"

:- There are four return conditions of **GetMessage ( )**

1) For all messages except **WM_QUIT**, it returns **1**.
2) For **WM_QUIT**, it returns **0** with successful exit status code.
3) If the condition is idle, means there is not a single message in message queue, then **GetMessage ( )** does not return and gives control to O.S.
4) When error occurs, during the operation of **GetMessage ( )**, then **GetMessage ( )** returns zero, but exit status code is erroneous.

⟶ Return Value of WinMain ( )

This indicates the exit status of application. This value comes from **MSG** type variable of Message Loop, means indirectly you can say that this value completely depends on the message information. **WinMain ( )** will return to O.S. with successful status only when **msg.wParam** is zero, which is possible only when **msg.message** member is **WM_QUIT**.

⟶ WndProc ( )

As stated before whether send or posted message has to come here, the duty of **WndProc ( )** is to process the required messages by **case-break** construct. Code bar in between the **case** statement & **break** statement of a particular message is the **Message Handler** of that particular message. Even your smallest application must have at least one message handler of **WM_DESTROY**. In **case-break** construct of **switch ( )**, we use messages as **MACROS** with beginning by **WM_**. It has definite 32-bit integer value which corresponds to **iMsg** of **WndProc ( )** (UINT parameter).

[*Note that every message arrives with other two informative values,* ***WPARAM, LPARAM*** *which are different for different messages.* For e.g. when **WM_PAINT** message arrives at **WndProc ( )** (whether sent or posted), it comes with handle of the Device Context as its WPARAM. So the programmer's duty is to first decide which message are to be processed and what output is expected from a particular message, then decide the code & algorithm for that output and write it as message handler of the respective message, e.g. if you want an application to display a line of text after pressing a left mouse button then the possible **case-break** construct in **WndProc ( )** will be as follows **:**

```
case WM_LBUTTONDOWN :
hdc = GetDC ( hwnd ) ;
GetClientRect ( hwnd, &rc ) ;
DrawText      ( hdc,
                "SaGaR",
                -1,
                &rc,
                DT_SINGLELINE | DT_CENTER | DT_VCENTER ) ;
ReleaseDC ( hdc );
break ;
```

After closing a brace of **switch** loop, there is a last statement of **WndProc ( )** which is one of the most important statement. This is the return statement, which returns long type variable. This value is usually returned not by you, but by O.S.

For this purpose, you call **DefWindowProc ( )** *API* in return statement of **WndProc ( )** which has same four parameters as **WndProc ( )**. Obviously its return value is also **LONG**. But the difference between prototype of **DefWindowProc ( )** & **WndProc ( )**, **DefWindowProc ( )** is not **CALLBACK**.

The purpose of this function is very important and details of its internal working are kept hidden by **Microsoft** as its **Trade-Secret**. But conceptually this function has following main functions :

1) It handles all those messages, which are not processed by application's **WndProc ( )**.
2) It returns the control from the application to O.S.
3) Default Window Procedure works as **WndProc ( )** for many pre-defined windows of O.S. itself.

Date : 07 – 06 – 2002.
Day : Friday.

Figure of Window Messages and WndProc ( )
(messages & **case-break** construct with message handler)

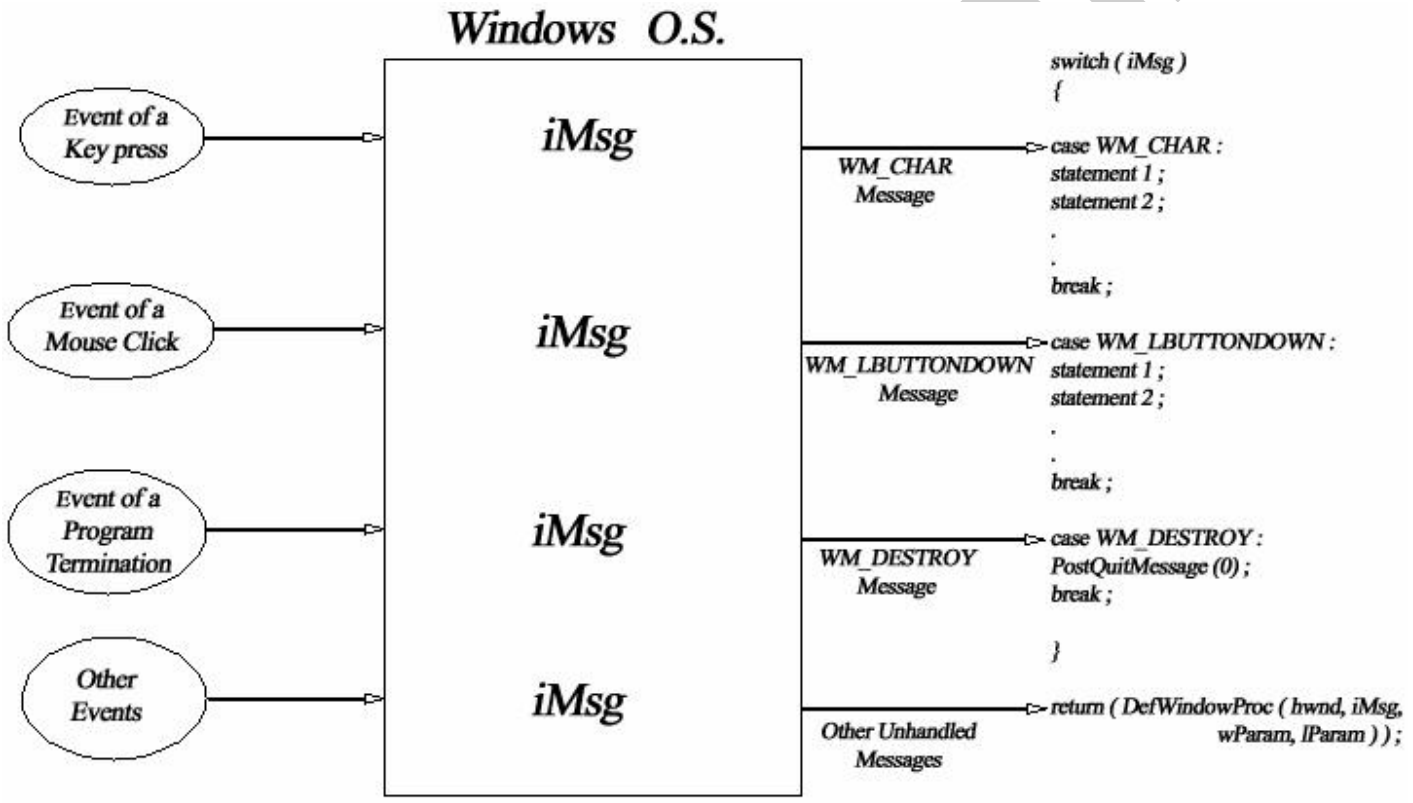


Figure of Sending of Message.

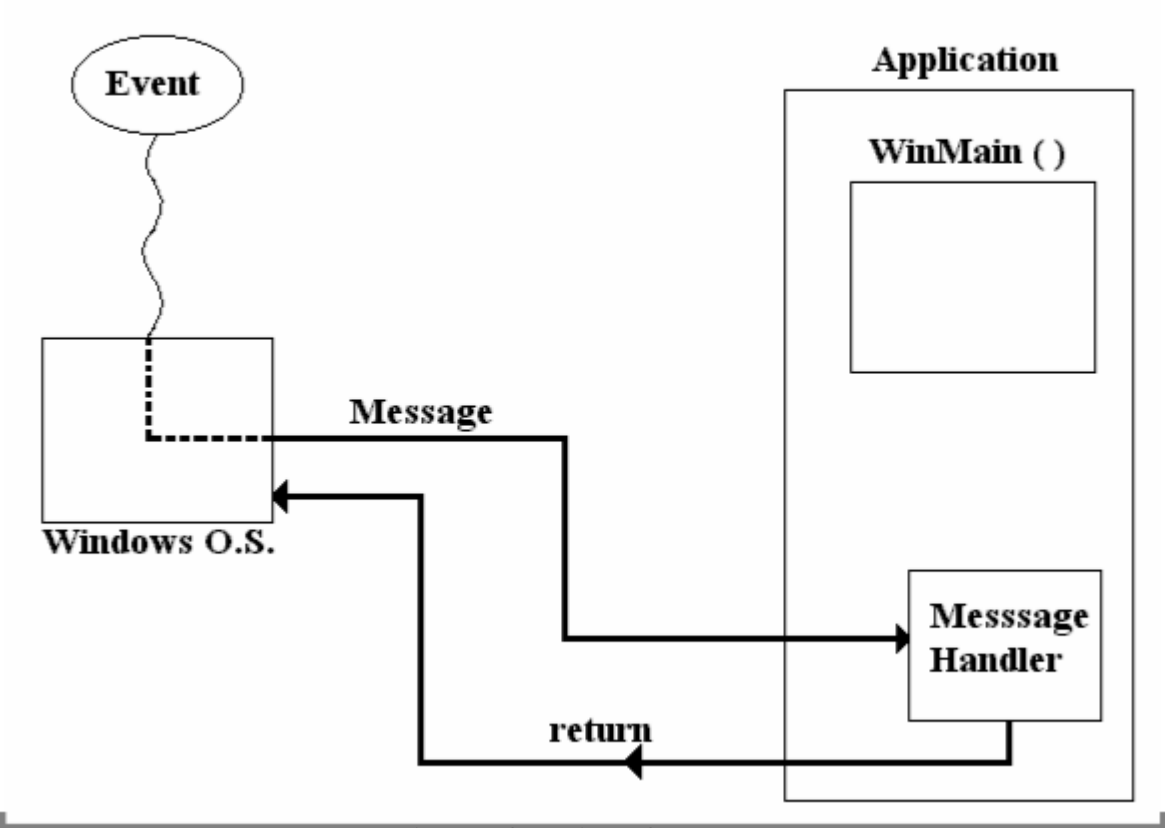


Figure of Posting of Message :

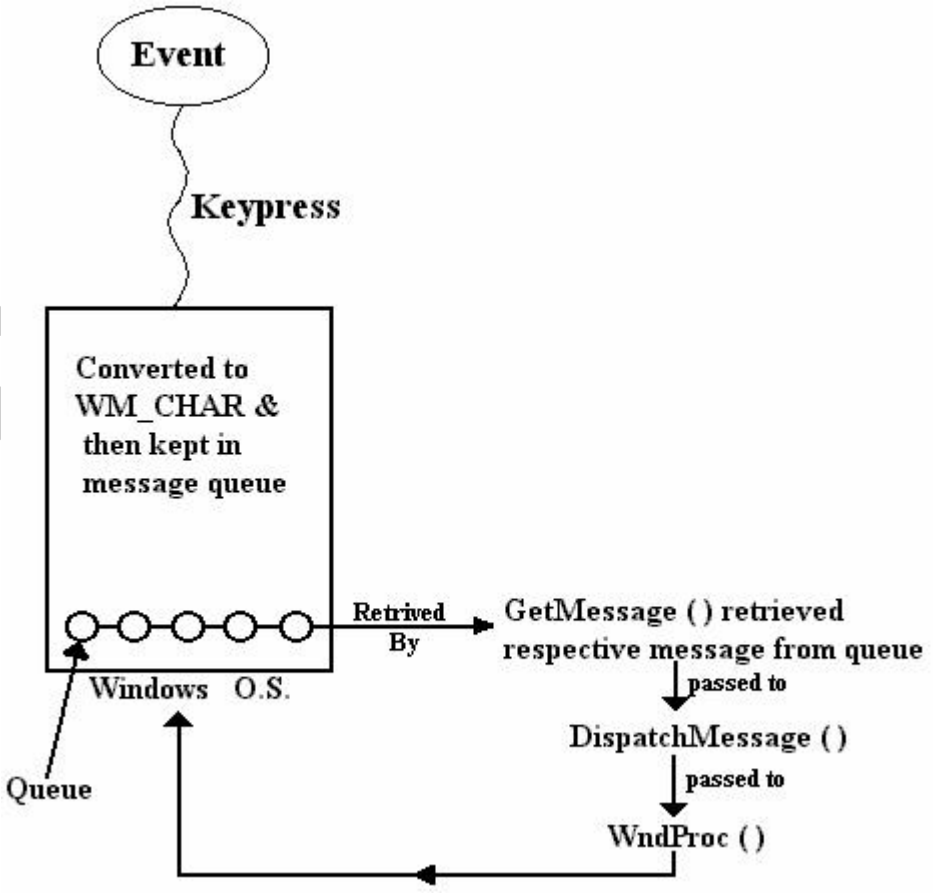
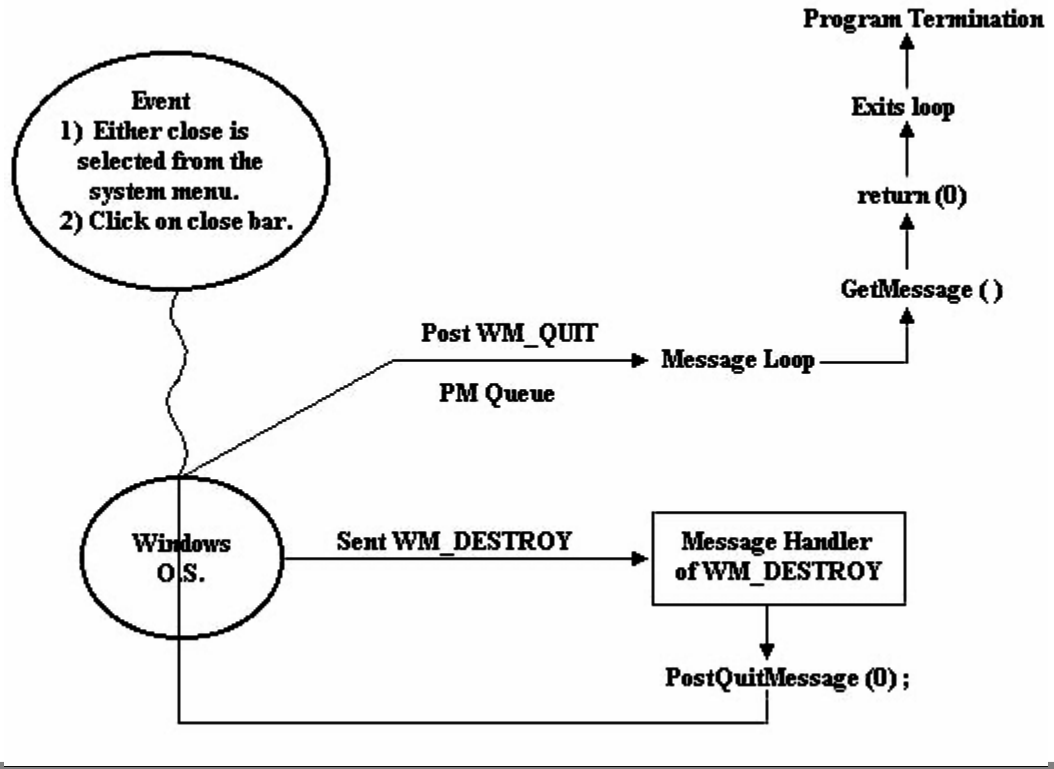


Figure of Program Termination :



Understanding of some important messages

From previous discussion, we know following things about messages: -

- Window messages are passed to your application by O.S. in Window Procedure.
- All messages are 32-bit integers (of UINT type) which are pre-defined by O.S. designers and which are made available to us as MACRO constant defined in **window.h**. All these values are written in HEX form. For e.g. **WM_QUIT** is **0X0012** and **CW_USEDEFAULT** is **0X080000000**. All messages are unique values.

Either sent or posted all window messages come to WndProc ( ) with three things which are also unique, means different for different messages. These three things are :

1) Message number as explained above.
2) Extra information about message of **WORD** type.
3) Extra information about message of **LONG** type.

e.g. **lParam** of **WM_CREATE** will point to the structure **CREATESTRUCT**, while lParam of **WM_LBUTTONDOWN**, we have screen co-ordinates of mouse positions.

There are in all about 200 pre-defined messages. Out of them, some of important messages are :-

01) **WM_CHAR :** When user presses a key on keyboard.
2) **WM_LBUTTONDOWN :** When user clicks a left mouse button.
3) **WM_RBUTTONDOWN :** When user clicks a right mouse button.
4) **WM_CREATE :** When a window is created, means when **CreateWindow ( )** *API* is called.
5) **WM_SHOWWINDOW :** When **ShowWindow ( )** *API* is called.
6) **WM_SIZE :** When window size is changed.
7) **WM_MOVE :** When window is moved.
8) **WM_COMMAND :** When your window wants to send message to its child window like Dialog Box, Radio Buttons, Push Buttons.
9) **WM_DESTROY :** When application is going to be terminated.

10) **WM_PAINT :** Whenever you want to do something with client area i.e. displaying text, diagrams, redrawing, etc.

Our Second Program :

Write following code into the **WndProc ( )** in switch construct before **"case WM_DESTORY :"**

```
case WM_CREATE :
MessageBox ( ) ( hwnd, "WM_CREATE message is sent", "Messages", MB_OK ) ;
break ;

case WM_MOVE :
MessageBox ( ) ( hwnd, "WM_MOVE message is sent", "Messages", MB_OK )
; break ;

case WM_SIZE :
MessageBox ( ) ( hwnd, "WM_SIZE message is sent", "Messages", MB_OK )
; break ;

case WM_KEYDOWN :
MessageBox ( ) ( hwnd, "WM_KEYDOWN message is sent", "Messages", MB_OK ) ;
break ;

case WM_LBUTTONDOWN :
MessageBox ( ) ( hwnd, "WM_LBUTTONDOWN message is sent", "Messages", MB_OK
); break ;
```

then our first program will look like this :-

```
#include < windows.h >        // Step 1

LRESULT CALLBACK WndProc (  HWND,
                            UINT,
                            WPARAM,
                            LPARAM ) ;  // Step 2

int WINAPI WinMain ( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpszCmdLine,
                     int nCmdShow ) // Step 3

{

        // Local Variables

        WNDCLASSEX wndclass ; // Step 4
        HWND hwnd ;
```

```
        MSG msg ;
        char AppName[ ] = "Windows" ;

        // Code

        wndclass.cbSize = sizeof ( WNDCLASSEX ) ;
        wndclass.style = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc = WndProc ;
        wndclass.cbClsExtra = 0 ;
        wndclass.cbWndExtra = 0 ;
        wndclass.hInstance = hInstance ;
        wndclass.hIcon = LoadIcon ( NULL, IDI_APPLICATION ) ;
        wndclass.hCursor = LoadCursor ( NULL, IDC_ARROW ) ;
        wndclass.hbrBackground = ( HBRUSH ) GetStockObject ( WHITE_BRUSH ) ;
        wndclass.lpszClassName = AppName ;
        wndclass.lpszMenuName = NULL ;
        wndclass.hIconSm = LoadIcon ( NULL, IDI_APPLICATION ) ;

        // Registration

        RegisterClassEx ( &wndclass ) ; // Step 5

        // Creation of Window

        hwnd = CreateWindow ( AppName,
                              "SaGaR's First Window",
                               WS_OVERLAPPEDWINDOW,
                               CW_USEDEFAULT,
                               CW_USEDEFAULT,
                               CW_USEDEFAULT,
                               CW_USEDEFAULT,
                               NULL,
                               NULL,
                               hInstance,
                               NULL ) ; // Step 6

        // Displaying of Window

        ShowWindow ( hwnd, nCmdShow ) ; // Step 7
        UpdateWindow ( hwnd ) ; // Step 8

        // Message of Loop i.e. Step 9

        while ( GetMessage ( &msg, NULL, 0, 0 ) )
                {
                        TranslateMessage ( &msg ) ;
                        DispatchMessage ( &msg ) ;

                }
        return (msg.wParam); // End of WinMain ( )
}
```

```
// Window Procedure (Window Function) : Step 10

LRESULT CALLBACK WndProc ( HWND hwnd,
                           UINT iMsg,
                           WPARAM wParam,
                           LPARAM lParam )
{

// Code

switch ( iMsg )
        {
                case WM_CREATE :
                MessageBox ( ) ( hwnd, "WM_CREATE message is sent", "Messages",
                MB_OK ) ;
                break ;

                case WM_MOVE :
                MessageBox ( ) ( hwnd, "WM_MOVE message is sent", "Messages",
                MB_OK ) ;
                break ;

                case WM_SIZE :
                MessageBox ( ) ( hwnd, "WM_SIZE message is sent", "Messages",
                MB_OK ) ;
                break ;

                case WM_KEYDOWN :
                MessageBox ( ) ( hwnd, "WM_KEYDOWN message is sent",
                "Messages", MB_OK ) ;
                break ;

                case WM_LBUTTONDOWN :
                MessageBox ( ) ( hwnd, "WM_LBUTTONDOWN message is sent",
                "Messages", MB_OK );
                break ;

                case WM_DESTROY :
                PostQuitMessage ( 0 ) ;
                break ;
        }
return ( DefWindowProc (hwnd, iMsg, wParam, lParam ) );
} // End of Complete Program
```

Output :-

When you execute the program, the first Message Box will be of **WM_CREATE** message handler. **Which indicates that WM_CREATE is first message, sent to your application by O.S. Another thing is that this message box does not get displayed indicating WM_CREATE is sent only once.**

After clicking "**OK**" of **WM_CREATE** Message Box further Message Boxes will be displayed.

- ¾ When you move the window, **WM_MOVE** Message Box will be displayed.
- ¾ When you change the size of the window, **WM_SIZE** Message Box will be displayed.
- ¾ When you press any key, say spacebar, then **WM_KEYDOWN** Message Box will be displayed.
- ¾ When you press left mouse button on the client area of your window, then **WM_LBUTTONDOWN** will be displayed.
- ¾ If you click on restore box, then sizing and move is done simultaneously, hence **WM_MOVE** & **WM_SIZE** Message Boxes are displayed one after the other.

Date : 08 – 06 – 2002
Day : Saturday.

MessageBox ( ) *API.*

This is very useful *API* function which is mainly used for three purposes :

1) To display informational messages to the user.
2) To display error messages to the user.
3) For stepwise debugging of a program.

In our second program, we use the message material directly in double quoted string form, but you can display variables in string form too, by using **sprintf ( )** & **MessageBox ( )** simultaneously. For these purpose you will need a temporary string variable.

e.g.

sprintf (STR, "%d", a) ;
MessageBox ( ) (hwnd, STR, "Value of a", MB_OK) ;

Output :

In above example, STR is temporary string variable and **a** is integer value whose value we want to display in Message Box.

The **MessageBox ( )** has four parameters :

1st parameter is Handle of the window :- means now Message Box will be displayed as child window of your window. But if you specify NULL, then O.S. considers desktop as Message Box's parent window.

2nd parameter is message string which we want to display in the Message Box.

3rd parameter is caption's string which displays a caption of the Message Box.

4th parameter is for two purposes :-

i) To display one, two or three push buttons, which may be one of the follows :

"OK"	**:**	**MB_OK**
"YES NO"	**:**	**MB_YESNO**
"ABORT RETRY CANCEL"	**:**	**MB_ABORTRETRYCANCEL**
"RETRY CANCEL"	**:**	**MB_RETRYCANCEL**
"OK CANCEL"	**:**	**MB_OKCANCEL**
"YES NO CANCEL"	**:**	**MB_YESNOCANCEL**

ii) You may display informative pre-defined icon which is displayed at the right side of the message.

Some important icons are :-
MB_ICONINFORMATION
MB_ICONERROR
MB_ICONEXCLAMATION
MB_ICONQUERY
MB_ICONSTOP

Official prototype of MessageBox ( )

int MessageBox (HWND, LPCTSTR, LPCTSTR, UINT) ;

When you want to use both button & icon together, then use bit-wise operator (|).
e.g. **MB_OK | MB_ICONINFORMATION**.

The return value is an integer of click button. e.g. If you click on "OK", then return value will be "**IDOK**", if you click on "CANCEL", then return value will be "**IDCANCEL**", for other values see **MSDN**.

Short Information about some important messages :

I) WM_CREATE : This is the first message sent to your window, when **CreateWindow( )** is called. This message is sent only once. When this message is sent, its **lParam** points to **CREATESTRUCT** structure whose members are exactly the same as of parameters of **CreateWindow ( )**, but in reverse order. You can use these members in **WndProc ( )** avoiding global declaration. e.g. Suppose you want to use Instance handle of your application in **WndProc ( )**, but as it is local for **WinMain ( )**, you should create another global variable, say **hInst** and **hInst = hInstance ;**

Now you can use, **hInst** in **WndProc ( )**, because **hInst** is global.

But though above way is easier, it creates one global handle. You can avoid this, because the 10th parameter of **CreateWindow ( )** is **hInstance** and you get it in **lParam** of **WM_CREATE** in **WndProc ( )**, you can get it locally in **WndProc ( )**.

Declare a local variable :
HINSTANCE hInst ;
Then write a following line :
hInst = (HINSTANCE) (CREATESTRUCT FAR *) wParam) -> hInst
OR
hInst = (HINSTANCE) (LPCREATESTRUCT lParam) -> hInst
By the same way, you can get any value of your window (from **CreateWindow ( )**) in WndProc ( ).

¾ When to use WM_CREATE message handler in your WndProc ( ) ?

: Above example is one of the reason to use **WM_CREATE** in your application, but this is not only one, there are other two important reasons :
1) Whenever you want to do some initializations, which are going to be used throughout the

application, then **WM_CREATE** is best space for it. *But Note That if you want to use these initializations in other message handlers, then don't anticipate that they will work if declared locally.* Hence to avoid this mishap follow one of the following way :-

a) Either declare such variable globally or
b) Declare them locally with static storage specifier.

2) As this message is always sent and only once and it is the first message, the initialization done in this message handler and initialized variables are global or local static then there is no need of re-initialization of same variables in other messages.

II) WM_MOVE :

This message is sent to your application, whenever a window is moved or restore/maximize box is used.]

Note : *While moving a window entire window with its frame, its contents move to different locations. Here you don't have to worry about the logic and code for moving all these contents. Windows O.S. itself takes care of whole logic.*

III) WM_SIZE :

This message is sent to your application, whenever a window is resized.

IV) WM_VSCROLL :

This message is sent to your application, whenever you will click on vertical scroll, bar , button or arrow.

V) WM_HSCROLL :

This message is sent to your application, whenever you will click on horizontal scroll, bar, button or arrow.

Date : 12 – 06 – 2002
Day : Wednesday.

VI) WM_PAINT : (Just explanation done about this topic.)

Date : 13 – 06 – 2002
Day : Thursday.

WM_PAINT is supposed to be a heart of Event Driven Program, because it is the message of painting either on the screen or on the printer. The key sentence of **WM_PAINT** is "**Windows O.S. sends/post WM_PAINT message to your application whenever screen is to be redrawn.**" In other words **WM_PAINT** message is sent in following eight situations :-

1) When a cursor of the mouse moves across the client area of the window.
2) When the whole window is moved to other location on the screen.
3) When icon is dragged & dropped across the client area.
4) Whenever the client area is over-lapped by another window (fully or partially.)
5) Whenever scrolling operation is done.
6) Resizing of Window. (Provided you should have **CS_HREDRAW** & **CS_VREDRAW** are assigned as class style in **WNDCLASSEX** structure.)
7) When the window is first created and displayed (**UpdateWindow ( )**).
8) Whenever menu is dragged and it overlaps the window.

(*Note : Selection and de-selection of the text in client area is also nothing but over-lapping of window.*)

Out of above eight jobs, 1,2,3,7 & 8 are done by O.S., you don't have to worry about its logic and code.

The job left to the programmer is thus limited for 4,5,6.

Out of above 8 situations, **WM_PAINT** is sent only when the **UpdateWindow ( )** is called (for 7th option).

For other conditions including 4,5,6, **WM_PAINT** is posted.

¾ Invalidate Rectangle or Invalidate Region :-

Whenever the other application's window overlaps your application's window partially, then overlapped region (which is to be repainted in future when uncovered) called as Invalidate Region. Invalidate rectangle is the smallest rectangle which is to be repainted, means in other words you can say that Invalidate Region is made of multiple Invalidate Rectangles.

¾ The Painting and Memory :-

From above discussion, now we know that the client area is going to be painted/repainted, either entirely (when application's window is minimized or overlapped completely) or partially (when another application's window covers some part for your application's window). When repainting of entire window is mercy, then the entire client area is repainted. But the partial area is to be repainted, then you have two choices :

1) Don't think about partial area but just repaint the entire client area. Though this is easy to done, it is CPU time consuming as well as Video Memory consuming.
2) Repaint only partial overlapped area. This is very much faster than the first option, also you can save Video Memory.

¾ How to implement WM_PAINT in your program ?

:- To implement **WM_PAINT** in your program, there must be a message handler :

```
case WM_PAINT :
        .
        .       \\ Painting Code
break ;
```

DEVICE CONTEXT

The job of painting can not be directly done, because the hardware i.e. Display Card or printer is encapsulated (insulated) from the direct access. So before any type of painting, you should request O.S. to give such a "**Specialist**" who is able to do painting on any type of display or printer hardware without caring about its Driver, Manufacturer, Memory, etc. This **Specialist** is called as **Device Context**.

As a programmer you can obtain handle to this Device Context by calling **GetDC ( )**, **BeginPaint ( )**, **CreateDC ( )**, **CreateCompatibleDC ( )**, etc.

In **WM_PAINT**, you will always get handle to the Device Context by using **BeginPaint ( )** *API* . Because **BeginPaint ( )** is capable of repainting the partial area or only the invalidate rectangle or invalidate region.

Device Context created by the other *API* are used mainly when you want handle of Device Context to pass as parameter to other *API*s.

Note that Device Context occupies much memory and hence it must be released or destroyed or deleted immediately after its use. Hence above Device Context creating *API*s also have their counter functions like :

EndPaint ( ) for **BeginPaint ( )**
ReleaseDC ( ) for **GetDC ( )**
DeleteDC ( ) for **CreateDC ( )**

After getting this Device Context, you can do any type of painting of the client area in **WM_PAINT** message handler. So the final **WM_PAINT** message handler :

```
case WM_PAINT :
hdc = BeginPaint ( hwnd, &ps ) ;
        .
        .
        .
EndPaint ( hwnd, &ps ) ;
break ;
```

Before using this case-construct you should have two variables declared :

```
HDC hdc ;
PAINTSTRUCT ps ;
```

Out of which **HDC** is handle to the Device Context and **ps** is the variable of **PAINTSTRUCT** structure whose address is the second parameter of **BeginPaint ( )** & **EndPaint ( )**.

PAINTSTRUCT is specialty of **BeginPaint ( )**, because it is the structure which is filled by O.S. and which with six members. Out of which three are of our concern :
HDC hdc, **BOOL fErase**, **RECT rcPaint**

Remaining three are reserved by Microsoft, we can not access them.

Hence **HDC** is the handle to the Device Context which is returned by **BeginPaint ( )**, means you can get **HDC** with **BeginPaint ( )** by two ways :-

1) **hdc = BeginPaint (hwnd, &ps) ;**
2) **BeginPaint (hwnd, &ps);**
   hdc = ps.hdc ;

Out of above, 1st one is preferable

The **fErase** member is for repainting of the background. It is of **BOOL** type, means if it is **YES** or **1**, then repainting of the background is done. If it **NO** or **0**, then repainting of background is not done.

The third member is the invalidate rectangle which is going to be repainted.

These are actually co-ordinates of the rectangle given in RECT structure variable which as four members :

```
int left ;
int top ;
int right ;
int bottom ;
```

(***Note :*** *Don't use one Device Context for longer time and don't use multiple Device Contexts are at the same time.* Some more functions regarding Device Context : **GetScreenDC ( )**, **GetWindowDC ( )**, **CreateMetafile ( )**,)

Date : 14 – 06 – 2002
Day : Friday.

If you need, you can force O.S. to send **WM_PAINT** message by using one of the following 3 functions :

1) **UpdateWindow ( )**
2) **InvalidateRect ( )**
3) **InvalidateRgn ( )**

Most commonly **InvalidateRect ( )** is used. This function can be used for repainting of the whole window or repainting of the invalidate rectangle only.

2nd Chapter - Text

(Explained in brief about TEXT)

==========

Date : 17 – 06 – 2002
Day : Monday.

(Explained in details about TEXT)

Date : 18 – 06 – 2002
Day : Tuesday.

Displaying a text is one of the important form of output in Windowing System. Our text is usually in following forms :

1) Single Line Centered Text.
2) Multi-line, but can fit in vertical window size.
3) Multi-Columnar, but can fit in horizontal window size.
4) Multi-line, but can not fit in vertical window size. (i.e. using scroll logic)
5) Multi-Columnar, but can not fit in horizontal window size (i.e. using scroll logic)

There are various text output functions, in WIN 32 *API*. Some of them are :

1) **DrawText ( )**
2) **TextOut ( )**
3) **TabbedTextOut ( )**
4) **ExtTextOut ( )**

Out of above 4, we are going to see first two. (Because, they are most common.) For remaining two, see **MSDN**.

1) DrawText ( ) :

It is mainly used for **Single Line Centered Text** of different fonts and different sizes.

It is not much used for regular multi-line & multi-columnar text output, because it does not deal with co-ordinates, but deals with string enclosing rectangle.

It has five parameters :

1) Handle of Device Context (Data-type : **hdc**)
2) Actual string or string variable or character pointer. (Data-type : **LPCTSTR**)
3) String length. (Data-type : **int**)
4) Address of **RECT** variable or long pointer to **RECT** structure. (Data-type : **LPRECT** or **RECT FAR ***)
5) Options for **DrawText ( )** (Hence, pre-fixed by **DT_**) (Data-type : **UINT**)

   Some options are :

   DT_SINGLELINE

DT_MULTILINE
DT_CENTER : For horizontal centering.
DT_VCENTER : For vertical centering.

2) TextOut ( ) :

This function also has five parameters. This function is mostly used for multi-line, multi-columnar textual output. Five parameters are as below :

1) Handle to Device Context. (Data-type : **hdc**)
2) X co-ordinate. (Data-type : int)
3) Y co-ordinate. (Data-type : int)
4) Actual string or string variable or character pointer. (Data-type : **LPCTSTR**)
5) String length. (Data-type : int)

™ CASE 1 : Code Procedure for displaying Single Line Centered Text using DrawText ( ) :

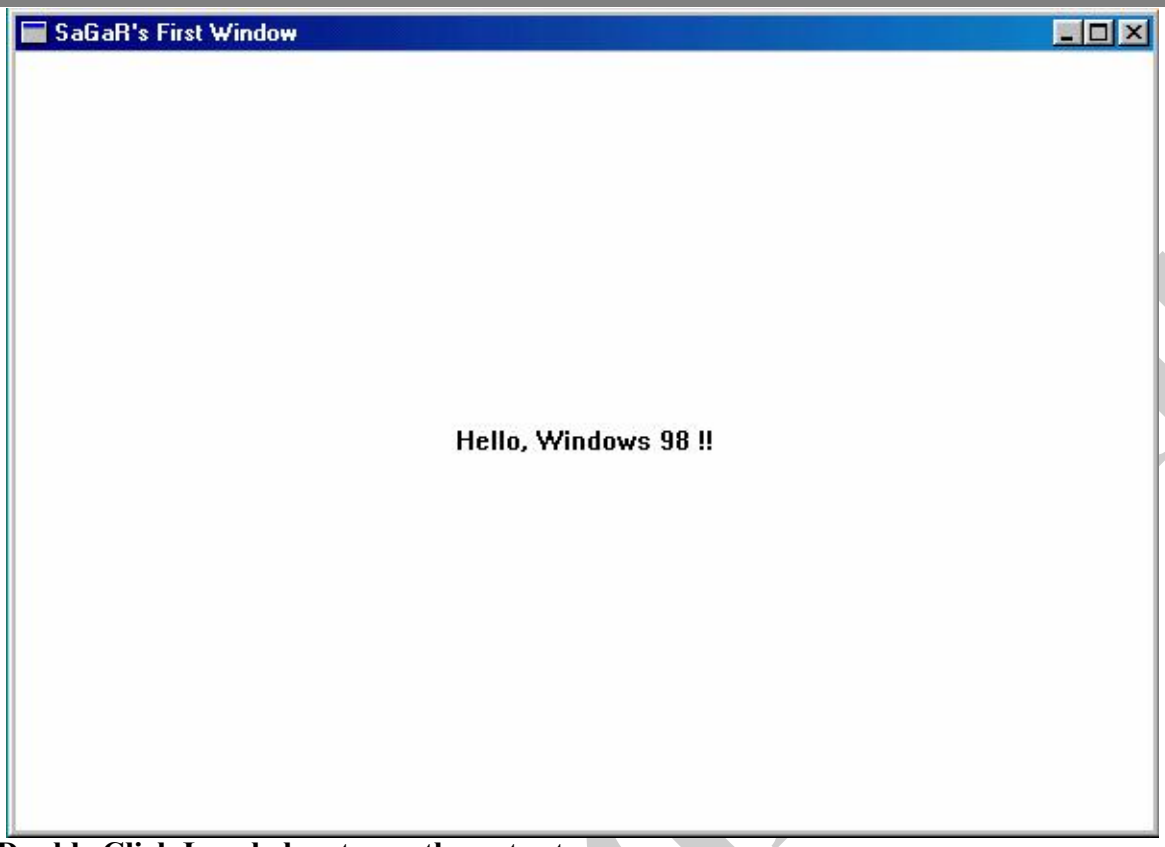
Step 1 : Go into **WM_PAINT** message handler.
Step 2 : Obtain Device Context Handle using **BeginPaint ( )**
Step 3 : Use **DrawText ( )** : Here, you can give co-ordinates of the rectangle in the middle of which your string is to be painted or if entire window is to be considered as the rectangle and you want to display the text in its middle, then use a non GDI-*API* i.e. **GetClientRect ( )** having two parameters : handle of the window and address of **RECT** structure.
Step 4 : EndPaint ( ).

Main Code is as follows :

```
HDC hdc ;
PAINTSTRUCT ps ;
RECT rc ;
switch (iMsg)
{
        case WM_PAINT :
        GetClientRect ( hwnd, &rc ) ;
        hdc = BeginPaint ( hwnd, &ps );
        DrawText      ( hdc,
                        “Hello, Windows 98 !!”,
                        -1,
                        &rc ,
                        DT_SINGLELINE | DT_CENTER | DT_VCENTER ) ;
        EndPaint ( hwnd, &ps ) ;
        break ;
        .
        .
        }
```

Note : *When you pass* ***–1*** *as string length parameter, whole string is considered for the output.*

Output of our program after adding above code, will look like below :



Or Double Click Icon below to see the output :


Window with Text.exe

™ CASE 2 : Code Procedure for displaying Multi-Line Text using TextOut ( ) :

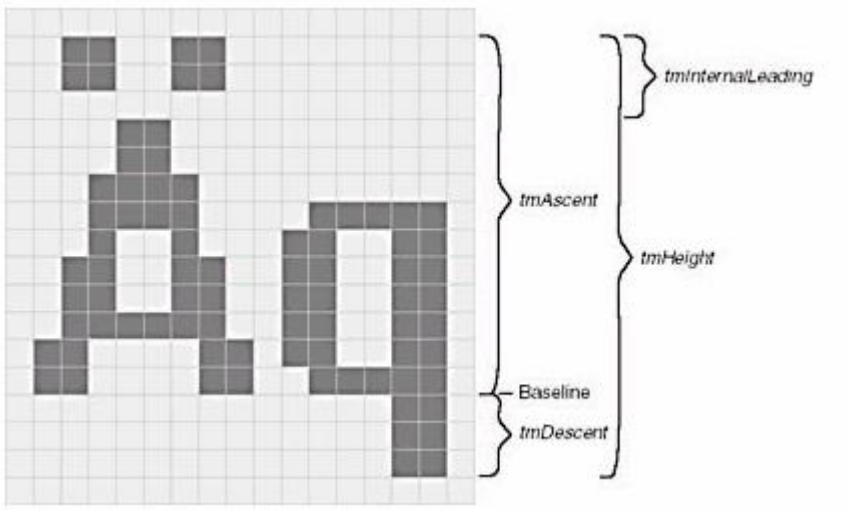
For this purpose, we are going to use **TextOut ( )**.

Here we are mainly dealing with continuously line by line changing Y co-ordinate keeping X co-ordinate constant.

For System Fonts and their properties, *API* uses a structure known as **TEXTMETRIC**, which has many members, but eight are of most important :

1) **tmAscent :** Height of the character above base-line.
2) **tmDescent :** Depth of the character below base-line.
3) **tmHeight :** Total height of the character, means tmAscent + tmDescent.
4) **tmExternalLeading :** Space left for over-line appendages like apostrophe s (’).
5) **tmInternalLeading :** It is subtraction (or difference) between maximum ascent of the upper-case letter and maximum ascent of the lower-case letter.
6) **tmAveCharWidth :** This is a average character width.
7) **tmMaxCharWidth :** This is a maximum character width. E.g. width of upper-case “W”.
8) **tmPitchAndFamily :** For fixed pitch font, its low bit is assigned to zero and for variable pitch font, its low bit is assigned to one.

Following diagram will explain in more details about font's Ascent, Descent, Height, Width, External & Internal Leading, Average Char Width and Pitch.



Date : 19 – 06 – 2002
Day : Wednesday.

Main Code :

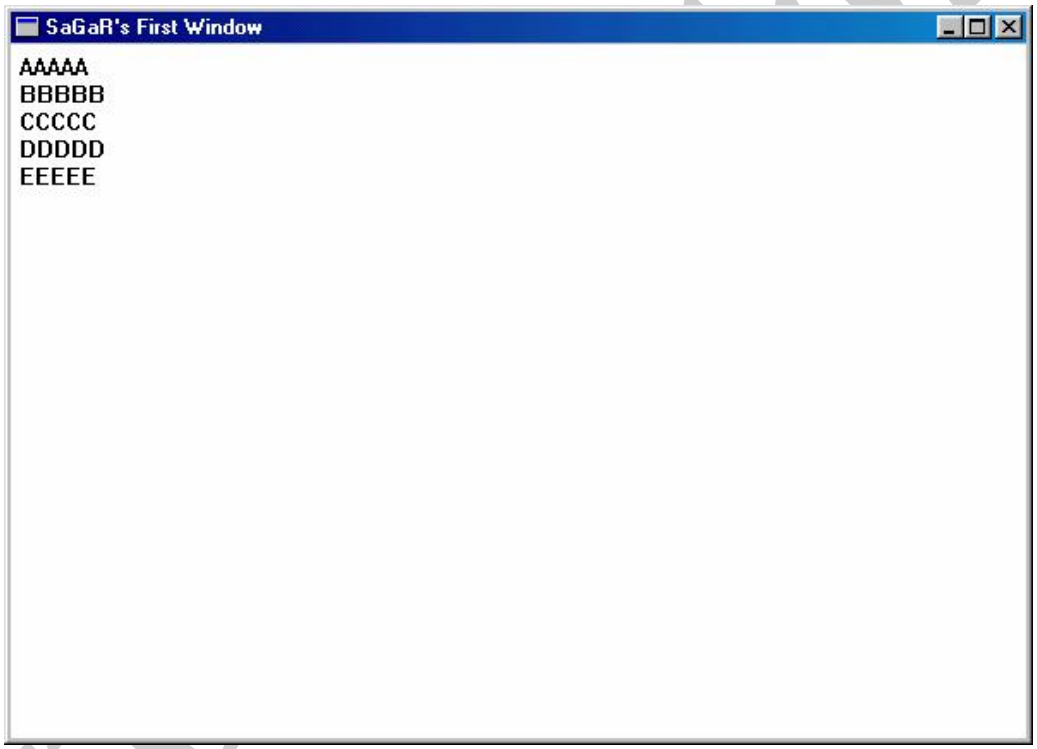
```
HDC hdc ;
TEXTMETRIC tm ;
static int cyChar ;
int i, X, Y ;
PAINTSTRUCT ps ;
char * str [ ] = { "AAAAA", "BBBBB", "CCCCC", "DDDDD", "EEEEE" } ;

case WM_CREATE :
hdc = GetDC ( hwnd ) ;
GetTextMetrics ( hdc, &tm );
ReleaseDC ( hwnd, hdc );
cyChar = tm.tmHeight + tm.tmExternalLeading ;
break ;

case WM_PAINT :
hdc = BeginPaint ( hwnd, &ps ) ;
X=5;
Y=5;
for ( i=0 ; i<=4 ; ++i )
                    ↑
                    └──   (because , we have 5 lines)
{
        TextOut ( hdc, X, Y, str [i], strlen ( str [i] ) ) ;
        Y = Y + cyChar ;
}

EndPaint ( hwnd, &ps ) ;
break ;
```

1) Here we increment Y co-ordinate every time by : **Y = Y + cyChar ;**
2) For the use of **strlen ( )**, it is better to include **stdio.h**.
3) Getting of **TextMetric** & **cyChar** is done in **WM_CREATE** , because we don't want them to be created every time, we want them to be created only once and as **WM_CREATE** is sent only once, we done these things here.
4) In **WM_CREATE** , to create **hdc**, we use **GetDC ( )**, as we don't want to paint, we want **hdc** only to pass to **GetTextMetrics ( )** as a parameter.

To see the output of program, double click on ICON :


Multi-Line Text.exe

Or see, the output screen snap :



™ CASE 3 : Code Procedure for displaying of Multi-Columnar, Single Line Text.

For multi-columnar output, Y co-ordinate is usually constant, if the output is multi-columnar single line. If the output is multi-columnar, multi-line, the both X & Y co-ordinates will change continuously.

While thinking about X co-ordinate, two terms are important. :

1) Width of the font : which is given by **tmAveCharWidth.**
2) Pitch of the font : which is given by **tmPitchAndFamily**.

The logic for the Y co-ordinate is same as the previous program of multi-line output.

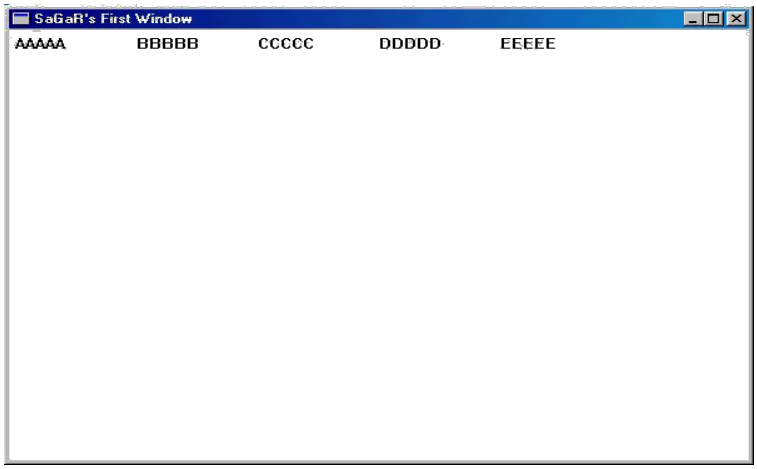
Main Code (assumed that multi-columnar but single line) :

```
HDC hdc ;
TEXTMETRIC tm;
static int, cxChar, cyChar, cxCaps ;
int i, X, Y ;
PAINTSTRUCT ps ;
char * str [ ] = { “AAAAA”, “BBBBB”, “CCCCC”, “DDDDD”, “EEEEE” } ;

case WM_CREATE :
hdc = GetDC ( hwnd ) ;
GetTextMetrics ( hdc, &tm ) ;
ReleaseDC ( hwnd, hdc ) ;
cxChar = tm.tmAveCharWidth ;
cyChar = tm.tmHeight + tm.tmExternalLeading ;

if ( ( tm.tmPitchAndFamily & 1 ) != 0 )
cxCaps = 3 × cxChar ÷ 2 ;
else
cxCaps = cxChar ;
break ;

case WM_PAINT :
hdc = BeginPaint ( hwnd, &ps ) ;
X=5;
Y=5;
for ( i = 0 ; i <= 4 ; ++i )
{
     TextOut ( hdc, X, Y, str [i], strlen ( str [i] ) );
     X = X + cxChar + (9 × cxCaps ) ;
}

EndPaint ( hwnd, &ps ) ;
break ;
```

Double click on the below icon to see the output of the above program :

Multi-Columnar Text.exe

or See the snap of the output :

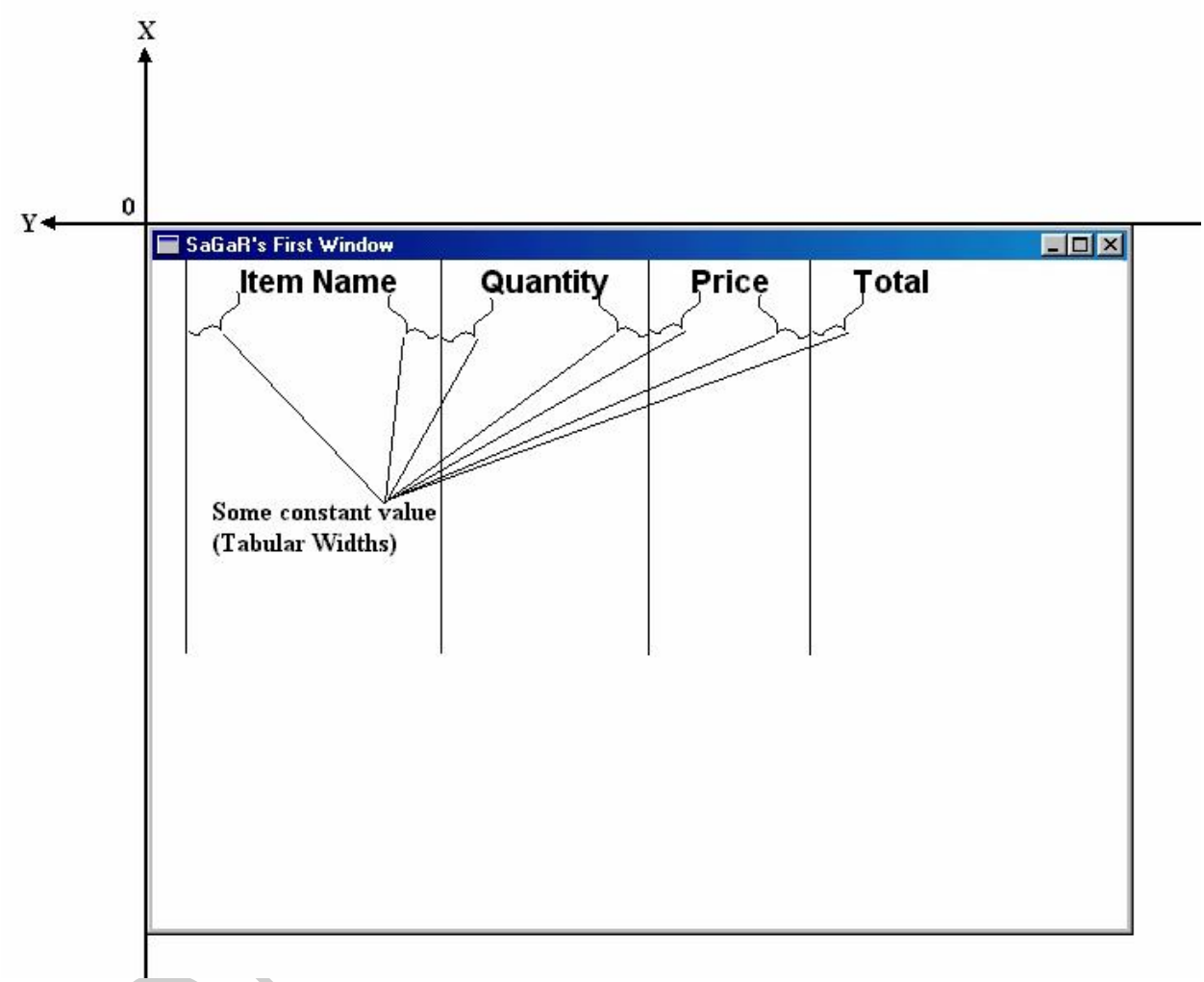


1) We declare **cxChar** & assign it to **tmAveCharWidth**.
2) According to the previous description of **tmPitchAndFamily** member, if the font is of variable pitch, we increase the width (i.e. **cxCaps**) by 150 percents (i.e. 1.5 times) and for fixed pitch, we keep it as average character width.
   Thus **if-else** code is written for this task.
3) While displaying consecutive columns, we have to know the maximum width of the previous column to get the X co-ordinate of the current column.
   In above example, the heading **Item_Name** is the maximum width column of 9 characters. Hence to get X co-ordinate of the **Price** column, we use following equation : X = X + cxChar + (9 × cxCaps) ;

   But to get X co-ordinate of succeeding column, you have to know maximum width of column.

   Our example does not support this, because we use fix column width as 9. But in actual practice, maximum column widths of all columns should be considered for neat output.
   Don't rely on the width of the column heading, but rely on the value and its width.

This is the ideal output :



Date : 24 – 06 – 2002
Day : Monday.

¾ Second method of placing Y co-ordinate for new line and X co-ordinate for new column.

:- In above method, we use **TEXTMETRIC** structure and we use its member for spacing of lines and columns.

But there is one *API*, which can do both these tasks at once. The name of the *API* is **GetTextExtentPoint32 ( )**. This function requires four parameters :

1) Handle of the Device Context.
2) The string.
3) String length
4) Pointer to an empty structure variable of **SIZE** type.

When this function returns, O.S. fills this structure variable by appropriate variable of length of the string in **cx** member and height of the string **cy** member.

Note that : *Both a length and height are font dependent.*

Hence work similar with ***tmHeight*** *and* ***tmAveCharWidth****.*

¾ How to use SIZE structure

? : Code :

```
HDC hdc ;
SIZE sz ;
int X, Y, i ;
PAINTSTRUCT ps ;
char * str [ ] = { “AAAAA”, “BBBBB”, “CCCCC”, “DDDDD”, “EEEEE” } ;
char * headings [ ] = { “AAAAA”, “BBBBB”, “CCCCC”, “DDDDD”, “EEEEE” } ;

case WM_PAINT :
hdc = BeginPaint ( hwnd, & ps ) ;
GetTextExtentPoint32 ( hdc, headings [0], strlen ( headings [0] ), &sz ) ;
// Maximum length string in headings array is
Item_Name X=Y=5;

for ( i = 0 ; i <= 3 ; ++i )
{
        TextOut ( hdc, X, Y, headings [i], strlen ( headings [i] ) ) ;
        X = X + sz.cx ;
}

// Multiple Lines
X=5;
Y = Y + sz.cy ;

for ( i = 0 ; i <= 4 ; ++i )
{
        TextOut ( hdc, X, Y, str [i], strlen ( str [i] ) ) ;
        Y = Y + sz.cy ;
}
EndPaint ( hwnd, &ps ) ;
break ;
```

1) How many characters can fit in window vertically & horizontally ?

:- To get this knowledge, we have to process **WM_SIZE** message, whose **lParam** gives this information.

HIWORD of **lParam** gives height of the window and **LOWORD** of **lParam** gives width of the window. If we know height and width of the window and at the same time and if we know height and width of character, then dividing respective terms, we can get the number of characters to fit, vertically & horizontally.

One another benefit of processing **WM_SIZE** is that, whenever window size is changed, new height & new width will be sent in **lParam**. Hence new values of number of characters to fit, horizontally & vertically, can be gained.

In scrolling logic, it is supposed to be sophisticated way of programming, to display scroll bars only when necessary.

For this purpose, we should be continuously aware of how many characters can fit in window, horizontally & vertically. Obviously, we need **WM_SIZE**.

Code :

```
HDC hdc ;
SIZE sz ;
PAINTSTRUCT ps ;
int xMaxChar, yMaxChar, cxClient, cyClient ;
int cxChar, cyChar, X, Y, i ;
char * str [ ] = { “AAAAA”, “BBBBB”, “CCCCC”, “DDDDD”, “EEEEE” } ;
char * headings [ ] = { “AAAAA”, “BBBBB”, “CCCCC”, “DDDDD”, “EEEEE” } ;
…… (same as above programs)
….
….
….
….

case WM_SIZE :

cxClient = LOWORD ( lParam ) ;
cyClient = HIWORD ( lParam ) ;
xMaxChar = cxClient / cxChar ;
yMaxChar = cyClient / cyChar ;
break ;
```

In above message handler, **xMaxChar** gives maximum number of characters can fit, horizontally and **yMaxChar** gives maximum number of characters can fit vertically.

Figure of Scroll bars and its various parts. (Figure 2)

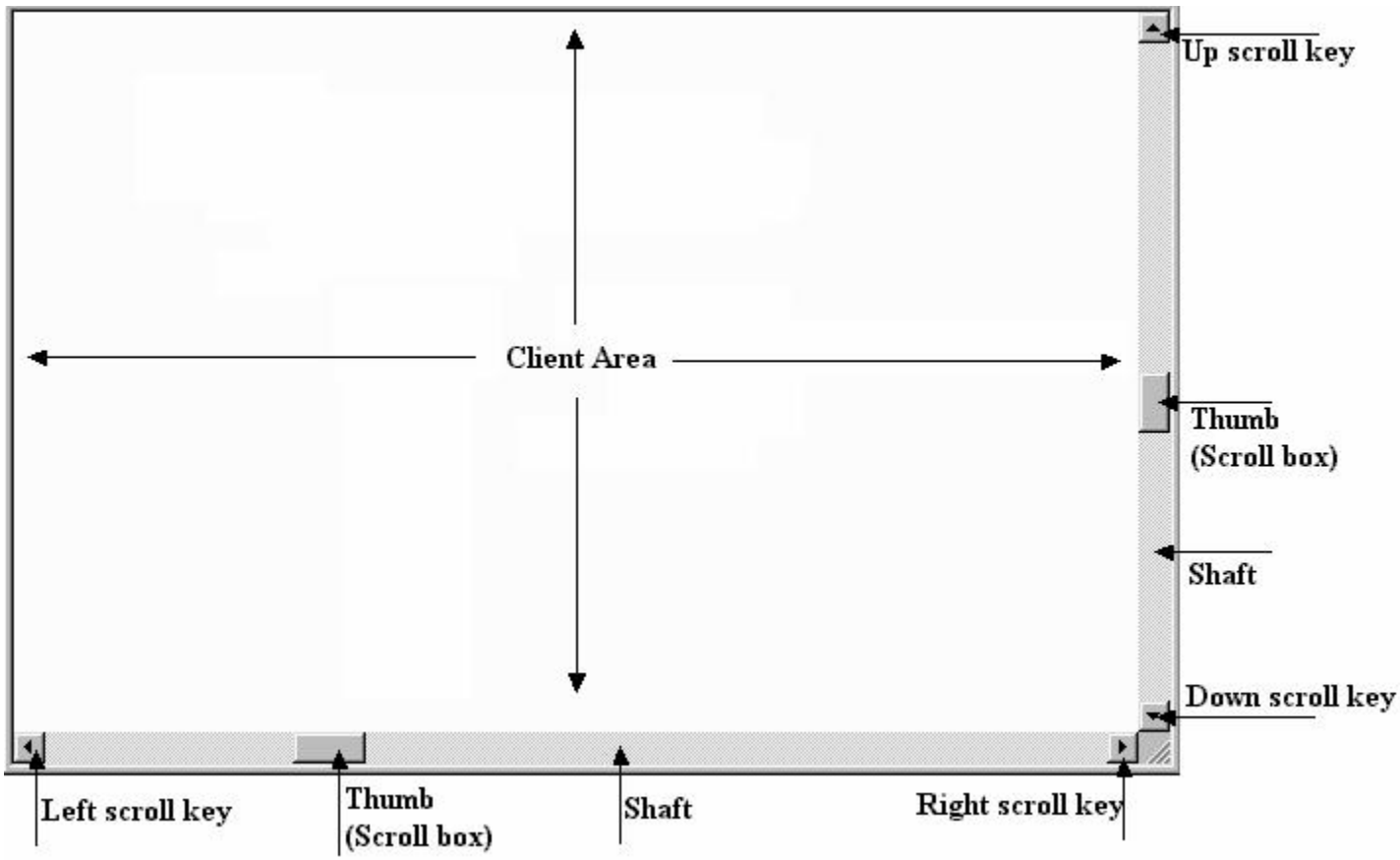
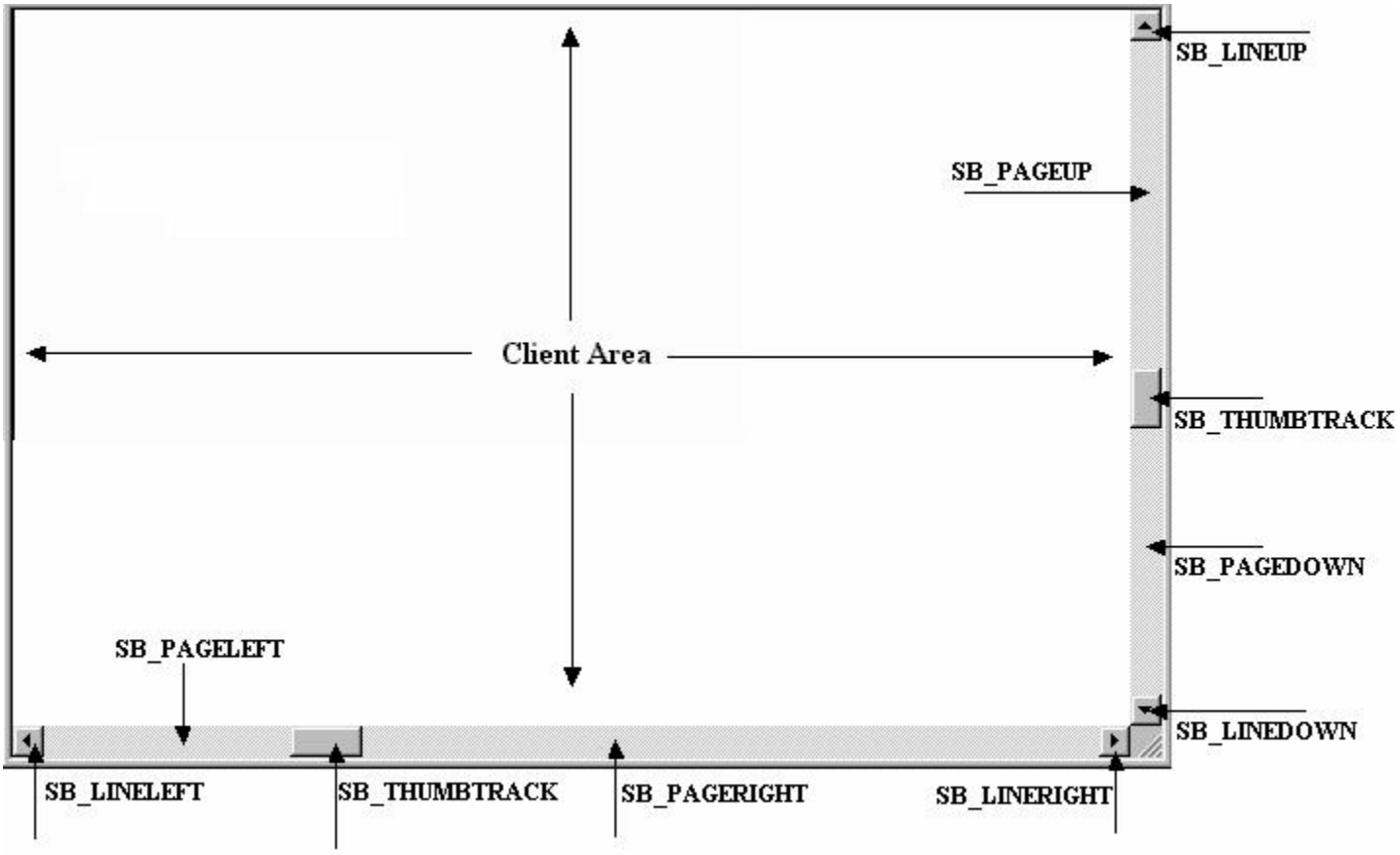


Figure of Scroll bars and messages regarding them. (Figure 2)



(Other messages, which can not be shown in figure are : **SB_TOP**, **SB_BOTTOM**, **SB_THUMBPOSITION**, **SB_ENDSCROLL**)

Note : *All above messages are given by mouse application on scroll bar. These messages are given, when mouse button is pressed on the shown areas. All above messages also have* ***SB_ENDSCROLL*** *message when mouse button is released from respective areas, except* ***SB_THUMBTRACK****.*

When mouse button is pressed on thumb, then ***SB_THUMBTRACK*** *message is sent as shown in figure. When mouse button is released instead of* ***SB_ENDSCROLL****, it sends* ***SB_THUMBPOSITION****.*

All mouse logic for scroll bar is handled by Windows O.S. itself. You don't need any ***WM_LBUTTONDOWN*** *or* ***WM_LBUTTONUP*** *handler. But keyboard logic for scrolling is not handled by Windows O.S. It must be given explicitly by programmer under* ***WM_KEYDOWN*** *message handler.*

¾ Scroll bars are one of the fascinating features of GUI. They are easy to use, but slightly difficult to program. These are two types of scroll bars.

a) Horizontal scroll bar.
b) Vertical scroll bar.

Its parts and respective program messages are shown in figure 1 & 2.

They are slightly difficult to program, because of different user's perspective & programmer's perspective for scroll bars.

Programmers sometimes have problems with scrolling terminology because their perspective is different from the user's. A user who scrolls down wants to bring a lower part of the document into view; however, the program actually moves the document up in relation to the display window. The Window documentation and the header file identifiers are based on the user's perspective: scroll up means moving toward the beginning of the document; scroll down means moving toward the end.

For vertical scroll bar, when any of its portion is used by mouse, Window sends **WM_VSCROLL** message to the **WndProc ( )** of the application. Its **wParam** has all above **SB_** MACROS, means during **WM_VSCROLL** message handler, we must again write a **switch-case** statement for all above "**SB**" values.

Similarly, when any of the portion of horizontal scroll bar is used by mouse, Window sends **WM_HSCROLL** message to the **WndProc ( )** of the application. Its **wParam** will have "**SB**" values of horizontal scroll bar.

¾ How to display the scroll bar ?

:- Displaying scroll bar for our window may require two situations :

1) Display the scroll bar, right from the beginning of the program, to the end of the program, whether scrolling is necessary or not.
For such a type of scroll bar, just add **WS_VSCROLL** for vertical scroll bar or **WS_HSCROLL** for horizontal scroll bar or **WS_VSCROLL | WS_HSCROLL** for both, as Window Style (third parameter) in **CreateWindow ( )** using | (bar/pipe sign) with **WS_OVERLAPPEDWINDOW**. Though, this is easy, it is bad programming practice, to keep scroll bars all the times displayed, even if no necessary. By using this style, you can do sophisticated program by :
   a) Display the scroll bar all the time, but making disabled, when not necessary and when scrolling requires make them enabled.
   b) Display the scroll bars only when necessary, otherwise hide them, when not necessary.

2) Showing of scroll bar also can be done by using one of the following *API*s.
   a) ShowScrollBar ( )
   b) SetScrollRange ( )
   c) SetScrollPos ( )
   d) SetScrollInfo ( )

Out of above functions, only first function is purely used for showing of scroll bars. Remaining functions, though can display scroll bars, they have additional effects. Hence in practice, **ShowScrollBar ( )** is commonly used for showing of scroll bars, when displaying is only task. When displaying of scroll bar is combined with another task, like setting the scrolling range or setting the thumb position, then other functions are used.

¾ **Prototype of ShowScrollBar ( ) :**
ShowScrollBar (HWND, int, BOOL) ;

1st parameter is the handle of the window to which the scroll bar is going to be attached.
2nd parameter is an integer value, either

SB_VERT for vertical scroll bar **or**
SB_HORZ for horizontal scroll bar **or**
SB_BOTH for both scroll bars
SB_CTL when scroll bar is going to be used as control for Dialog Box, etc.

Note : *Besides scrolling main window, scroll bar is also can be used as child window control for dialog boxes. e.g. Spin Control.*

Displaying a scroll bar by **ShowScrollBar ( )** *API* is usually done when an application has several displays. Not all requires scrolling but few may require. In such cases this *API* is used only when scrolling require displays.

3rd parameter is to display or hide a scroll bar. To display the scroll bar, this parameter is set to TRUE (i.e. 1) and to hide, this parameter is set to FALSE (i.e. 0).

¾ Scroll Bar Messages.

For Vertical Scroll Bar	Key-Codes	For Horizontal Scroll Bar	Key-Codes
1) SB_TOP	VK_HOME		
2) SB_BOTTOM	VK_END		
3) SB_LINEUP	VK_UP	1) SB_LINERIGHT	VK_RIGHT
4) SB_LINEDOWN	VK_DOWN	2) SB_LINELEFT	VK_LEFT
5) SB_PAGEUP	VK_PRIOR	3) SB_PAGERIGHT	--------
6) SB_PAGEDOWN	VK_NEXT	4) SB_PAGELEFT	--------
7) SB_THUMBTRACK	--------	5) SB_THUMBTRACK	--------
8) SB_THUMBPOSITION	--------	6) SB_THUMBPOSTION	--------
9) SB_ENDSCROLL	--------	7) SB_ENDSCROLL	--------

From Above Table, SB_TOP & SB_BOTTOM are messages, which can be handled by only keyboard .

Out of these nine vertical scroll bar messages, seven vertical scroll bar messages (one of the **THUMBTRACK** or **THUMBPOSITION**) are processed for mouse actions with vertical scroll bar.

And out of seven horizontal scroll bar messages, six horizontal scroll bar messages (one of **THUMBTRACK** and **THUMBPOSITION**) are processed for mouse actions with horizontal scroll bar.

Then for keyboard action, six virtual key-codes are processed for vertical scroll bar and only two are processed for horizontal scroll bar.

- Keyboard cannot process **SB_THUMBTRACK** & **SB_THUMBPOSITION** scroll bar messages in both, vertical & horizontal scroll bars.
- Keyboard cannot process **SB_PAGERIGHT**, **SB_PAGELEFT** messages for horizontal scroll bar.
- **SB_ENDSCROLL** is handled directly by O.S.

Code

Assumptions :

- Here we assume that, we have multi-line, multi-columnar text out, which can not fit in normal window size, hence it requires both, vertical & horizontal scrolling.
- We also assume that, you have already declared **cxChar**, **cyChar**, **cxCaps** with **TEXTMETRIC tm ;**.
- We also assume that, we already have one dimensional or two dimensional array with multiple lines or multiple columns, say **char text [ ] [ ] ;**
- We also assume that, we have a MACRO **LINES**, #defined as 100. e.g. **# define LINES 100**
- We also assume that, we have already set **WM_CREATE** message handler and **TEXTMETRIC** logic as previous program.

Date : 02 – 07 – 2002
Day : Tuesday.

Steps

1) **What to do in WM_CREATE ?**
2) **What to do in WM_SIZE ?**
3) **What to do in WM_VSCROLL ?**
4) **What to do in WM_HSCROLL ?**
5) **What to do in WM_PAINT ?**
6) **What to do in Keyboard Scrolling ?**

Before starting the steps, we have to study one structure and two functions.
The structure is **SCROLLINFO** and the functions are **GetScrollInfo ( )** & **SetScrollInfo ( )**.

¾ Structure **SCROLLINFO** :

Declaration :

```
typedef struct tag SCROLLINFO
{
        UINT cbSize ; // Size of SCROLLINFO structure.
        UINT fMask ; // Attributes to get or to set.
        int nMin ;          // Minimum value of the scrolling range.
        int nMax ;          // Maximum value of the scrolling range.
        UINT nPage ;        // Page size.
        int nPos ;          // Scrolling thumb position.
        int nTrackPos ;     // Scrolling thumb's tracking postion.
}
SCROLLINFO, * PSCROLLINFO ;
```

i) When much of the members are kept empty and address of the **SCROLLINFO** structure variable is used in **GetScrollInfo ( )**, then system fills the empty member values of this structure, to get attributes of current scrolling mechanism.

ii) When much of the members are given values by programmer and address of such structure variable is passed to **SetScrollInfo ( )**, then system uses programmer given values of this structure to set the current scrolling mechanism.
Besides given meanings of structure members, fMask member requires additional information, because it is the member which is going to retrieved or set the scrolling mechanism.
There are six defined values of this member. Those are :-

1) **SIF_RANGE**
2) **SIF_POS**
3) **SIF_PAGE**
4) **SIF_TRACKPOS**
5) **SIF_DISABLENOSCROLL**
6) **SIF_ALL**

1) **SIF_RANGE :**
When we want to get or set the range, this value is used. When this member is used by **SetScrollInfo ( )**, it means that you want to set the range, means obviously, we must specify **nMin** & **nMax** members of the **SCROLLINFO** structure. When we use this value with **GetScrollInfo ( )**, then system, itself fills **nMin** & **nMax** value for us.

2) **SIF_POS :**
When we want to get or set the position, this value is used. When this member is used by **SetScrollInfo ( )**, it means that you want to set the position, means obviously, we must specify **nPos** member of the **SCROLLINFO** structure. When we use this value with **GetScrollInfo ( )**,then OPERATING SYSTEM, itself fills **nPos** value for us.

3) **SIF_PAGE :**
When we want to get or set the page size, this value is used. When this member is used by **SetScrollInfo ( )**, it means that you want to set the page size, means obviously, we must specify **nPage** member of the **SCROLLINFO** structure. When we use this value with **GetScrollInfo ( )**, then OPERATING SYSTEM, itself fills **nPage** for us.
Important Note : *In WIN 16 API, the size of the thumb was static (i.e. constant), but Win 32 API advanced this feature, so that size of the thumb will change according to the size of document.*
This doesn't mean that you can't use fixed size thumb. If you wish fixed size thumb, just don't use this value with ***GetScrollInfo ( )*** *&* ***SetScrollInfo ( )*** *API.*

4) **SIF_TRACKPOS :**
This value can be used only with **GetScrollInfo ( )** *API* and this return value is used to set the **SB_THUMBTRACK** or **SB_THUMBPOSITION** scroll bar message in **WM_VSCROLL** & **WM_HSCROLL**.
When used with **GetScrollInfo ( )**, system fills **si.nTrackPos** member with current tracked thumb position.

5) **SIF_DISABLENOSCROLL :**
This value can be used only with **SetScrollInfo ( )** *API*. This flag is important when the scroll bar is not required, means when the number of lines (vertically) or number of words (horizontally) are capable of fitting inside client area.
Here we have two choices, either keep the scroll bar visible but disabled then or hide the scroll bar. Out of these two choices, when you want to keep the scroll bar visible but wanted to disable it, then this value is used.

6) **SIF_ALL :**
SIF_ALL is the combination of **SIF_RANGE + SIF_POS + SIF_PAGE + SIF_TRACKPOS** and used to retrieve or to set these members with **GetScrollInfo( )** and **SetScrollInfo ( )**.

¾ Functions : GetScrollInfo ( ) & SetScrollInfo ( ).

GetScrollInfo ( ) :

Parameters of GetScrollInfo ( ) are :

1) **Handle of the window.**
2) **Type of scroll bar.**
3) **Address of SCROLLINFO variable.**

Prototype of GetScrollInfo ( ) :

GetScrollInfo (HWND, int, PSCROLLINFO) ;

OR **GetScrollInfo (HWND, int , SCROLLINFO *) ;**

SetScrollInfo ( ) :

Parameters of SetScrollInfo ( ) are :

1) **Handle of the window.**
2) **Type of scroll bar.**
3) **Address of SCROLLINFO variable.**
4) **Whether scroll bar should be displayed (TRUE) or not to be displayed (FALSE).**

Prototype of SetScrollInfo ( ) :

SetScrollInfo (HWND, int, PSCROLLINFO, BOOL) ;

OR **SetScrollInfo (HWND, int , SCROLLINFO *, BOOL) ;**

Date : 03 – 07 – 2002
Day : Wednesday.

Step 1 : What to do in WM_CREATE ?

i) As usual steps of our previous programs, write the code up to **ReleaseDC ( )** as it is, means Device Context, TEXTMETRIC , cxChar, cyChar, cxCaps, etc.

ii) Declare another static integer variable **iMaxWidth** and assign your last column's X co-ordinate to it. This is quite OK, for multi-columnar output. But if your output is not columnar, but having long sentences, then **iMaxWidth** variable should assigned to the X co-ordinate of the longest sentence. (For these X co-ordinates, see **WM_PAINT's** X,Y)

Step 2 : What to do in WM_SIZE ?

i) Get width and height of the page in **cxClient** & **cyClient** as explained before.

ii) Set the vertical scrolling accordingly (depending on **WM_SIZE**), means according on current size.
Declare **SCROLLINFO** variable, **SCROLLINFO si ;** and right the following code :

```
// Vertical scrolling settings
si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_RANGE | SIF_PAGE ;
si.nMin = 0;
si.nMax = LINES – 1 ;
si.nPage = cyClient / cyChar ;
SetScrollInfo ( hwnd, SB_VERT, &si, TRUE );
```

iii) Set horizontal scrolling accordingly, by adding following code :

```
// Horizontal scrolling settings
si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_RANGE | SIF_PAGE ;
si.nMin = 0 ;
si.nMax = iMaxWidth – 1 ;
si.nPage = iMaxWidth / cxChar + 2 ; // 2 is added in equation for neat
output. SetScrollInfo ( hwnd, SB_HORZ, &si, TRUE ) ;
```

Step 3 : What to do in WM_VSCROLL ?

i) You are already in **switch(iMsg)** construct, but when Windows OPERATING SYSTEM sends **WM_VSCROLL** or **WM_HSCROLL**, it sends a value as **LOWORD** of **wParam**, so it has to be checked again with another **switch-case** construct.

```
case WM_VSCROLL :

// Get current scrolling information
si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_ALL ; // to get scroll info of SIF_RANGE, SIF_POS, SIF_PAGE,
                        SIF_TRACKPOS
GetScrollInfo ( hwnd, SB_VERT, &si ) ;
nVertPos = si.nPos ; // Declare nVertPos variable before getting current position
                        of thumb
switch ( LOWORD ( wParam) )
{
        case SB_TOP :
        si.nPos = si.nMin ;
        break ;

        case SB_BOTTOM :
        si.nPos = si.nMax ;
        break ;

        case SB_LINEUP :
        si.nPos = si.nPos –1 ;
        break ;

        case SB_LINEDOWN :
        si.nPos = si.nPos + 1 ;
        break ;

        case SB_PAGEUP :
        si.nPos = si.nPos – si.nPage ;
        break ;

        case SB_PAGEDOWN :
        si.nPos = si.nPos + si.nPage ;
        break ;

        case SB_THUMBTRACK :
        si.nPos = si.nTrackPos ;
        break ;

        default :
        break ;

}
// According to value of si.nPos in case…break, we have to set the
thumb. si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_POS ;
SetScrollInfo ( hwnd, SB_VERT, &si, TRUE ) ;
// Windows O.S. sets the position of the thumb, according to above call, but if
sets the value according to the current size of the window. Hence, retrieve the
Windows's adjusted thumb position again
GetScrollInfo ( hwnd, SB_VERT, &si ) ;
```

```
// Now compare saved variables value nVertPos with retrieved nPos and do painting
only if these values are different.
if ( nVertPos != si.nPos )
{
        ScrollWindow ( hwnd, 0, cyChar × ( nVertPos – si.nPos ), NULL, NULL
        ); UpdateWindow ( hwnd );
}
break ;
```

Step 3 : What to do in WM_HSCROLL ?

```
case WM_HSCROLL :

// Get current scrolling information
si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_ALL ; // to get scroll info of SIF_RANGE, SIF_POS, SIF_PAGE,
                        SIF_TRACKPOS
GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
nHorzPos = si.nPos ; // Declare nHorzPos variable before getting current position of
                        thumb
switch ( LOWORD ( wParam) )
{
        case SB_LINELEFT :
        si.nPos = si.nPos – 1 ;
        break ;

        case SB_LINERIGHT :
        si.nPos = si.nPos + 1;
        break ;

        case SB_PAGELEFT :
        si.nPos = si.nPage –1 ;
        break ;

        case SB_PAGERIGHT :
        si.nPos = si.nPage + 1 ;
        break ;

        case SB_THUMBTRACK :
        si.nPos = si.nTrackPos ;
        break ;

        default :
        break ;

}
// According to value of si.nPos in case…break, we have to set the
thumb. si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_POS ;
```

```
SetScrollInfo ( hwnd, SB_HORZ, &si, TRUE ) ;
// Windows O.S. sets the position of the thumb, according to above call, but if
sets the value according to the current size of the window. Hence, retrieve the
Windows's adjusted thumb position again
GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
//  Now compare saved variables value nHorzPos with retrieved nPos and do
painting only if these values are different.
if ( nHorzPos != si.nPos )
{
        ScrollWindow ( hwnd, cxChar × ( nHorzPos – si.nPos ), 0, NULL, NULL );
        UpdateWindow ( hwnd );
}
break ;
```

Date : 08 – 07 – 2002.
Day : Monday.

Step 3 : What to do in WM_PAINT ?

The **WM_PAINT** is sent to the application's window procedure, when we touch any of the scroll bar. This is not at all shocking, because you force OPERATING SYSTEM to do this by writing **UpdateWindow ( )** *API* in **WM_VSCROLL** & **WM_HSCROLL** after the call to **ScrollWindow ( )** *API*.

if (nVertPos != si.nPos) OR **if (nHorzPos != si.nPos)** is the most important part of the code of both scrolling, because this code tells to do painting whenever the two values differ by user's interaction with the scroll bar.

Within these **if-blocks** before **UpdateWindow ( )**, we wrote **ScrollWindow ( )** *API*. This is also very important *API*, because through **ScrollWindow ( )** *API*, we tell the system, dimension of the area of the painting and then we sent **WM_PAINT** through **UpdateWindow( ).**

Now in **WM_PAINT**, above mentioned repainting area is sent by the OPERATING SYSTEM to you in **rcPaint** member of the **PAINTSTRUCT** variable **ps**. We will use dimensions of this rectangle as **iPaintBeg** & **iPaintEnd** variables. Code :

```
case WM_PAINT :
hdc = BeginPaint ( hwnd, &ps ) ;

// Getting vertical scroll thumb position
si.cbSize = sizeof ( SCROLLINFO ) ;
si.fMask = SIF_POS ;
GetScrollInfo ( hwnd, SB_VERT, &si )
; nVertPos = si.nPos ;

// Getting Horizontal scroll thumb
position si.cbSize = sizeof ( SCROLLINFO
) ; si.fMask = SIF_POS ;
GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
nHorzPos = si.nPos ;
```

```
// Setting the repainting limits
iPaintBeg = max ( 0, nVertPos + ps.rcPaint.top / cyChar ) ; // max & min are MACROS,
                                                            used as functions.
iPaintEnd = min ( LINES – 1, nVertPos + ps.rcPaint.bottom / cyChar ) ;

for ( i = iPaintBeg ; i <= iPaintEnd ; i++ )
{
        X = cxChar × ( 1 – nHorzPos ) ;
        Y = cyChar × ( i – nVertPos ) ;
.
.
.
// other code is same as previous Text outputs…

}
EndPaint ( hwnd, &ps ) ;
break ;
```

Note : *Don't forget to declare variables* ***iPaintBeg, iPaintEnd, i ;***

¾ Keyboard Logic for Scrolling :

Just add following case code to enable keyboard scrolling.

```
case WM_KEYDOWN :
    {
        switch (wParam)
        {
            case VK_HOME :
            SendMessage ( hwnd, WM_VSCROLL, SB_TOP, 0 );
            break ;

            case VK_END :
            SendMessage ( hwnd, WM_VSCROLL, SB_BOTTOM, 0 );
            break ;

            case VK_UP :
            SendMessage ( hwnd, WM_VSCROLL, SB_LINEUP, 0 );
            break ;

            case VK_DOWN :
            SendMessage ( hwnd, WM_VSCROLL, SB_LINEDOWN, 0 );
            break ;

            case VK_PRIOR :
            SendMessage ( hwnd, WM_VSCROLL, SB_PAGEUP, 0 )
            ; break ;
```

```
                case VK_NEXT :
                SendMessage ( hwnd, WM_VSCROLL, SB_PAGEDOWN, 0 ) ;
                break ;

                case VK_LEFT :
                SendMessage ( hwnd, WM_HSCROLL, SB_LINELEFT, 0
                ); break ;

                case VK_RIGHT :
                SendMessage ( hwnd, WM_HSCROLL, SB_LINERIGHT, 0
                ); break ;

                default :
                break ;
        }
```

A Complete Program for Scroll bars.

```
#include < windows.h >
#include <stdio.h>
#define LINES 26

LRESULT CALLBACK WndProc (  HWND,
                            UINT,
                            WPARAM,
                            LPARAM ) ;

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

{

        // Local Variables

        WNDCLASSEX wndclass ;
        HWND hwnd ;
        MSG msg ;
        char AppName[ ] = "Windows" ;

        // Code

        wndclass.cbSize = sizeof ( WNDCLASSEX ) ;
        wndclass.style = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc = WndProc ;
        wndclass.cbClsExtra = 0 ;
        wndclass.cbWndExtra = 0 ;
        wndclass.hInstance = hInstance ;
```

```
        wndclass.hIcon = LoadIcon ( NULL, IDI_APPLICATION ) ;
        wndclass.hCursor = LoadCursor ( NULL, IDC_ARROW ) ;
        wndclass.hbrBackground = ( HBRUSH ) GetStockObject ( WHITE_BRUSH ) ;
        wndclass.lpszClassName = AppName ;
        wndclass.lpszMenuName = NULL ;
        wndclass.hIconSm = LoadIcon ( NULL, IDI_APPLICATION ) ;

        // Registration
        RegisterClassEx ( &wndclass ) ;

        // Creation of Window
        hwnd = CreateWindow ( AppName,
                              "SaGaR's First Window",
                              WS_OVERLAPPEDWINDOW,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                              NULL,
                              NULL,
                              hInstance,
                              NULL ) ;

        // Displaying of Window

        ShowWindow ( hwnd, SW_MAXIMIZE ) ;
        UpdateWindow ( hwnd ) ;

        // Message of Loop

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

        return (msg.wParam);


        // End of WinMain ( )
}

        // Window Procedure (Window Function) :

        LRESULT CALLBACK WndProc ( HWND hwnd,
                                   UINT iMsg,
                                   WPARAM wParam,
                                   LPARAM lParam )
        {
```

```
        // Declaration of local
        variables HDC hdc ;
        TEXTMETRIC tm ;
        static int cxChar, cyChar, cxCaps, iMaxWidth, nVertPos, nHorzPos
        ; static int iPaintBeg, iPaintEnd, cxClient, cyClient ;
        int i, X, Y ;
        PAINTSTRUCT ps
        ; SCROLLINFO si ;

        char                *                                   str[26]                 =
{"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ",
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
"CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC",
“DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD”,
“EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EEE EE”,
“FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF”,
“GGGG GGGG GGGG GGGG GGGG GGGG GGGG GGGG GGGG GGGG GGGG”,
“HHH HHH HHH HHH HHH HHH HHH HHH HHH HHH HHH HHH HHH HHH”,
“IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII IIII”, “JJJJ
JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ JJJJ”, “KKKK
KKKK KKKK KKKK KKKK KKKK KKKK KKKK KKKK KKKK KKKK KKKK”, “LLLL
LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL”,
“MMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM”,
“NNNN NNNN NNNN NNNN NNNN NNNN NNNN NNNN NNNN NNNN NNNN”,
“OOO OOO OOO OOO OOO OOO OOO OOO OOO OOO OOO OOO OOO OOO”,
“PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP PPP”,
“QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ QQQ”,
“RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR RRR”,
“SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS SSS”,
“TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT”,
“UUU UUU UUU UUU UUU UUU UUU UUU UUU UUU UUU UUU UUU UUU”,
“VVV VVV VVV VVV VVV VVV VVV VVV VVV VVV VVV VVV VVV VV”,
“WWW WWW WWW WWW WWW WWW WWW WWW WWW WWW WWW”,
“XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX,
“ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ ZZZZ” } ;

        // Code switch
        ( iMsg )
                {
                case WM_CREATE :
                        hdc = GetDC ( hwnd ) ;
                        GetTextMetrics ( hdc, &tm ) ;
                        ReleaseDC ( hwnd, hdc ) ;
                        cxChar = tm.tmAveCharWidth ;
                        cyChar = tm.tmHeight + tm.tmExternalLeading ;
                        iMaxWidth = 52 ;
                        if ( ( tm.tmPitchAndFamily & 1 ) != 0
                        ) cxCaps = 3 * cxChar / 2 ;
                        else
                        cxCaps = cxChar
                        ; break ;
```

```
case WM_SIZE :
        cxClient = LOWORD ( lParam ) ;
        cyClient = HIWORD ( lParam ) ;

        // Vertical Scrolling Settings
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_RANGE | SIF_PAGE ;
        si.nMin = 0 ;
        si.nMax = LINES - 1 ;
        si.nPage = cyClient / cyChar ;
        SetScrollInfo ( hwnd, SB_VERT, &si, TRUE ) ;

        // Horizontal Scrolling Settings
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_RANGE | SIF_PAGE ;
        si.nMin = 0 ;
        si.nMax = iMaxWidth - 1 ;
        si.nPage = iMaxWidth / cxChar + 2 ;
        SetScrollInfo ( hwnd, SB_HORZ, &si, TRUE ) ;
        break ;

case WM_VSCROLL :
        // Get current scrolling information
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_ALL ; /* to get scroll info of SIF_RANGE, SIF_POS,
        SIF_PAGE, SIF_TRACKPOS */
        GetScrollInfo ( hwnd, SB_VERT, &si ) ;
        nVertPos = si.nPos ; /* Declare nVertPos variable before getting
        current position of thumb */

        switch ( LOWORD ( wParam) )
        {
                case SB_TOP :
                        si.nPos = si.nMin ;
                        break ;

                case SB_BOTTOM :
                        si.nPos = si.nMax ;
                        break ;

                case SB_LINEUP :
                        si.nPos = si.nPos -1 ;
                        break ;

                case SB_LINEDOWN :
                        si.nPos = si.nPos + 1 ;
                        break ;

                case SB_PAGEUP :
                        si.nPos = si.nPos - si.nPage ;
                        break ;
```

```
                case SB_PAGEDOWN :
                        si.nPos = si.nPos + si.nPage ;
                        break ;

                case SB_THUMBTRACK :
                        si.nPos = si.nTrackPos ;
                        break ;

                        default :
                        break ;
        }
        /* According to value of si.nPos in case…break, we have to set the
        thumb. */
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_POS ;
        SetScrollInfo ( hwnd, SB_VERT, &si, TRUE ) ;
        /* Windows O.S. sets the position of the thumb, according to above
        call, but if sets the value according to the current size of the window.
        Hence,retrieve the Windows's adjusted thumb position again */
        GetScrollInfo ( hwnd, SB_VERT, &si ) ;
        /* Now compare saved variables value nVertPos with retrieved nPos
        and do painting only if these values are different. */
        if ( nVertPos != si.nPos )
        {
                ScrollWindow ( hwnd, 0, cyChar * ( nVertPos - si.nPos ), NULL,
                                                        NULL );
                UpdateWindow ( hwnd );
        }
        break ;

case WM _HSCROLL :
        // Get current scrolling information
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_ALL ; /* to get scroll info of SIF_RANGE, SIF_POS,
        SIF_PAGE, SIF_TRACKPOS */
        GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
        nHorzPos = si.nPos ; /* Declare nHorzPos variable before getting
        current position of thumb */

        switch ( LOWORD ( wParam) )
                {
                        case SB_LINELEFT :
                                si.nPos = si.nPos - 1 ;
                                break ;


                case SB_LINERIGHT :
                                si.nPos = si.nPos + 1;
                                break ;
```

```
                        case SB_PAGELEFT :
                                si.nPos = si.nPos - si.nPage ;
                                break ;

                        case SB_PAGERIGHT :
                                si.nPos = si.nPos + si.nPage ;
                                break ;

                        case SB_THUMBTRACK :
                                si.nPos = si.nTrackPos ;
                                break ;

                        default :
                                break ;
                        }

                case WM_KEYDOWN :

                        switch (wParam)
                                {
                                case VK_HOME :
                                        SendMessage ( hwnd, WM_VSCROLL, SB_TOP, 0 );
                                        break ;

                                case VK_END :
                                        SendMessage ( hwnd, WM_VSCROLL, SB_BOTTOM, 0
);                                      break ;

                                case VK_UP :
                                        SendMessage ( hwnd, WM_VSCROLL, SB_LINEUP, 0 );
                                        break;

                                case VK_DOWN :
                                        SendMessage ( hwnd, WM_VSCROLL, SB_LINEDOWN,
                                                0 );
                                        break ;

                                case VK_LEFT :
                                        SendMessage ( hwnd, WM_HSCROLL, SB_LINELEFT,
                                                0);
                                        break ;

                                case VK_RIGHT :
                                        SendMessage ( hwnd, WM_HSCROLL, SB_LINERIGHT,
                                                0 );
                                        break ;

                                case VK_PRIOR :
                                        SendMessage ( hwnd, WM_VSCROLL, SB_PAGEUP, 0
                                        ); break ;
```

```
                case VK_NEXT :
                        SendMessage ( hwnd, WM_VSCROLL,
                                        SB_PAGEDOWN, 0 ) ;
                        break ;

                default :
                break ;
                }

        /* According to value of si.nPos in case…break, we have to set the
        thumb. */
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_POS ;
        SetScrollInfo ( hwnd, SB_HORZ, &si, TRUE ) ;
        /* Windows O.S. sets the position of the thumb, according to above
        call, but if sets the value according to the current size of the window.
        Hence, retrieve the Windows's adjusted thumb position again */
        GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
        /* Now compare saved variables value nHorzPos with retrieved nPos
        and do painting only if these values are different. */
        if ( nHorzPos != si.nPos )
        {
                ScrollWindow ( hwnd, cxChar * ( nHorzPos - si.nPos ), 0,
                                NULL, NULL );
                UpdateWindow ( hwnd );
        }
        break ;

case WM_PAINT :
        hdc = BeginPaint ( hwnd, &ps ) ;
        // Getting vertical scroll thumb position
        si.cbSize = sizeof ( SCROLLINFO ) ;
        si.fMask = SIF_POS ;
        GetScrollInfo ( hwnd, SB_VERT, &si )
        ; nVertPos = si.nPos ;
        // Getting Horizontal scroll thumb
        position si.cbSize = sizeof ( SCROLLINFO
        ) ; si.fMask = SIF_POS ;
        GetScrollInfo ( hwnd, SB_HORZ, &si ) ;
        nHorzPos = si.nPos ;
        // Setting painting limits
        iPaintBeg = max ( 0, nVertPos + ps.rcPaint.top / cyChar ) ;
        iPaintEnd = min ( LINES - 1, nVertPos + ps.rcPaint.bottom / cyChar ) ;


        for ( i = iPaintBeg ; i <= iPaintEnd ; i++ )
                {
                        X = cxChar * ( 1 - nHorzPos ) ;
                        Y = cyChar * ( i - nVertPos ) ;
                        TextOut ( hdc, X, Y, str [i], strlen ( str [i] ) ) ;
                }
```

```
                EndPaint ( hwnd, &ps ) ;
                break ;
        case WM_DESTROY :
                PostQuitMessage ( 0 ) ;
                break ;
        }

    return ( DefWindowProc (hwnd, iMsg, wParam, lParam ) );
} // End of Complete Program
```

Date : 09 - 07 – 2002
Day : Tuesday.

Above Program did practically on a computer in a class.

Date : 10 - 07 – 2002
Day : Tuesday.

¾ Colouring the Text.

Before colouring the text, we should know, the data structure of the colour in Win 32. This Data structure is called **COLORREF**. It is a 32 -bit number having 0 to 7 bits for red, 8 to 15 bits are for green, 16 to 23 for blue and 24 to 31 are reserved and are zero.

Each colour ranges its intensity from 0 to 255. Win 32 *API* provides a MACRO for colours known as **RGB ( )**, where R stands for RED, G stands for GREEN and B stands for BLUE.

This MACRO function has three parameters. :

1st integer is for RED, 0 to 255.
2nd integer is for GREEN, 0 to 255.
3rd integer is for BLUE, 0 to 255.

e.g.

RGB (255, 0 , 0)	**:**	**Red**	**(Pure).**
RGB (0, 255, 0)	**:**	**Green**	**(Pure).**
RGB (0, 0, 255)	**:**	**Blue**	**(Pure).**
RGB (0, 0, 0)	**:**	**Black**	**(Pure).**
RGB (255, 255, 255)	**:**	**White**	**(Pure).**
RGB (128, 128, 128)	**:**	**Grey**	**(Moderate).**
RGB (64, 64, 64)	**:**	**Grey**	**(Dark).**
RGB (192, 192, 192)	**:**	**Grey**	**(Light).**
RGB (255, 255, 0)	**:**	**Yellow.**	
RGB (0, 255, 255)	**:**	**Cyan.**	
RGB (255, 0, 255)	**:**	**Magenta.**	
RGB (255, 200, 0)	**:**	**Orange.**	
RGB (255, 175, 175)	**:**	**Pink.**	

For colour of the text , there are two *API*s.

1) **GetTextColor ( )**
2) **SetTextColor ( )**

1) **GetTextColor ( )** :

This function retrieves current text colour.

Prototype :
COLORREF GetTextColor (HDC) ;

Parameter is the handle to the Device Context and the return value is RGB constant.

2) **SetTextColor ( )** :

This function sets the colour of the text to desired colour. This function not only sets the current text colour but also return the previous text colour. The default text colour is black.

Prototype :
COLORREF SetTextColor (HDC, COLORREF) ;

Parameters are the handle to the Device Context & RGB constant and the return value is also a RGB constant which returns previous text colour.

¾ Colouring the Background :

Colouring the text also involves colouring the background too. For colouring the background there two *API*s.

1) **GetBkColor ( )**
2) **SetBkColor ( )**

1) **GetBkColor ( ) :**

This function retrieves current background colour.

Prototype :
COLORREF GetBkColor (HDC) ;

Parameter is the handle of Device Context and return value is RGB constant..

2) **SetBkColor ( )**

This function sets the colour of the background to desired colour. This function not only sets the current background colour but also return the previous background colour. The default background colour is white.

Prototype :
COLORREF SetBkColor (HDC, COLORREF) ;

Parameters are the handle to the Device Context & RGB constant and the return value is also a RGB constant, which returns previous text colour.

‰ Background colour also depends on background mode. Win 32 *API* defines two background modes. :

1) **OPAQUE.**
2) **TRANSPARENT**.

OPAQUE is default mode. There are two *API*s concerned with background modes :

1) GetBkMode ( )
2) SetBkMode ( )

1) GetBkMode ( ) :]

This *API* retrieves current background mode as integer.

Prototype :
int GetBkMode (HDC) ;

Returns either **OPAQUE** or **TRANSPARENT**.

2) SetBkMode ( ) :

This function sets background mode to the desired value, which either **OPAQUE** or **TRANSPARENT** and it also returns previous background mode.

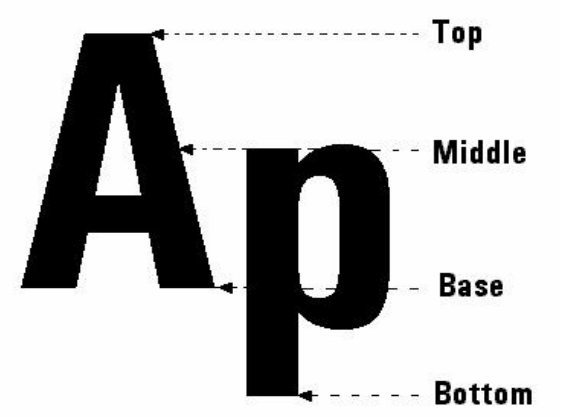
Prototype :
int SetBkMode (HDC, int) ;

Note : *There are many "**Set**" functions in Win 32 API, which not only set the current value but also return the previous value which may be used in the program, somewhere else. e.g.* ***SetTextColor ( ), SetBkColor ( ), SetBkMode ( )****.*

¾ Text alignment & justification. :

Many times, especially in multi-columnar output, there may be some columns with text & columns alternatives letters & numbers in a font having different sizes. Hence output may look unjustified. Thus to align the numbers or to justify the text, Win 32 has two main *API*s :

Diagram : Text



Important Note : *When we give Y co-ordinate, means next line's co-ordinate, painting of the text by default starts from the top, not from the baseline. Means letter A in the next line will be positioned by its tip at Y co-ordinate. (Above diagram explains the top, middle, baseline & bottom of a font).*

1) **SetTextAlign ( )**
2) **SetTextJustification ( )**

1) SetTextAlign ( ) :

Prototype :
UINT SetTextAlign (HDC, UINT) ;

The alignment value is of **UINT** type and this function sets the alignment passed as parameter and returns the previous alignment value.

There are many values :
TA_BASELINE
TA_TOP
TA_BOTTOM
TA_CENTER
TA_RIGHT
TA_LEFT
TA_UPDATECP
TA_NOUPDATECP
TA_RTLREADING (For right to left reading)

At a time two or more values can be used by |.
e.g.
For left justification use :
SetTextAlign (HDC, TA_LEFT | TA_TOP) ;
For right jusification use :
SetTextAlign (HDC, TA_RIGHT | TA_TOP) ;

The default alignments are **TA_TOP** & **TA_LEFT**.
In pictorial languages like kanji, the common alignment styles are **VTA_BASELINE**, **VTA_CENTER**.
You should use this function when you have multi-columnar textual & numerical output.

2) SetTextJustification ( ) :

This *API* sets the break character (e.g. space) for desired number of times to give uniform look to the output.
Prototype :
BOOL SetTextJustification (HDC, int, int) ;
Where the second parameter is number of required break characters and third parameter is the actual break character,
This function is usually used for single output contains multiple fonts in a single line.

3rd Chapter - Graphics

The *API* for graphics user interface are located in **GDI32.DLL**. The graphics output devices are mainly of two types :

1) **Raster Device.**
2) **Vector Device.**

Most PCs use **Raster Device** which includes display adapter (i.e. Display card).
Raster devices represent everything in terms of dots (pixels), while vector devices represent everything in terms of shapes like line.
The e.g. of Raster device are display adapter, dot matrix printer, laser printer, etc.
The e.g. of Vector device is plotter.

¾ Co-ordinate System

Windows 9x uses co-ordinate system by two ways :

1) **Device co-ordinate system.**
2) **Virtual co-ordinate system.**

1) Device co-ordinate system :

This system actually deals with hardware and *API* of this system directly communicates with the VGA card or printer.

2) Virtual co-ordinate system :

The *API* of this system, do not directly interact with the hardware, but the co-ordinates of this system can be mapped into actual device co-ordinate.
(**Note :** Mapping is nothing but scaled conversion.)

Windows 9x support co-ordinates in 16-bit number (0 to 32767), but Windows NT support 32-bit co-ordinates (0 to 4 GB number).

Microsoft Windows has vast capacity of drawing different objects on both, screen as well as printer. But its memory model can not support fast animation. For this purpose, you have to take some external graphics library like **Silicon Graphics** invented **OpenGL**.

¾ Types of functions concerned with GDI

1) Those functions which get or release Device Context. e.g. **GetDC ( )** – **ReleaseDC ( )**, **BeginPaint ( )** – **EndPaint ( )**.
2) Those functions which give information about the Device Context as information context. e.g. **CreateIC ( )**.
3) Those functions which get or set the attributes (properties) of the Device Context. e.g. **SetTextColor ( )**, **SetTextAlign ( )**, **SetBkColor ( )**.
4) Those functions which actually paint the different shapes or text, like **LineTo ( )**, **Rectangle ( )**, **Ellipse ( )** and for text, **TextOut ( )**, **TabbedTextOut ( )**, **DrawText ( )**.

5) Those functions which are not actual **GDI** functions (because they do not have **HDC** as parameter), but they give different objects to draw shape or text. These objects are pen, brush & font.
   e.g. **CreatePen ( )**, **CreateSolidBrush ( )**, **CreateFont ( )**.
   Different shapes that we can draw are
   1) Lines
   2) Curves
   3) Geometrical objects
   4) Bazler
   5) B-Spline
   6) Filled areas
   7) Pie
   Some additional graphic objects like bitmaps, Meta files, palette, region, path, clipping, mapping modes, etc.

™ Device Context (Revisited)

As explained in "**TEXT**" topic, now we know how to Get & Release Device Context by using **GetDC ( )** – **ReleaseDC ( )** pair or by using **BeginPaint ( )** – **EndPaint ( )** pair. But the DC created by above functions is limited only for the client area.

In other words, we can say that **GetDC ( )** & **BeginPaint ( )** return handle to the Device Context of client area only.

- Sometimes, we may want handle to the Device Context of entire window (not just the client area), then we should use **GetWindowDC ( )** *API*, whose prototype is :
  HDC GetWindowDC (HWND) ;
  The handle created by this function should be released by **ReleaseDC ( )**.

- Sometimes we may want handle to the Device Context for entire screen (like in full screen game or full screen movies), then there are two ways :

1) Use **GetDC ( )** , but with NULL parameter, which will return to the desktop window i.e. fullscreen.
2) Or use **CreateDC ( )** *API* function, which has four parameters and the prototype is :
   HDC CreateDC (LPCTSTR, LPCTSTR, LPCTSTR, CONST DEVMODE *) ;
   1st parameter is Driver name.
   2nd parameter is Device name.
   3rd parameter is reserved, always NULL,
   4th parameter is pointer to **DEVMODE** structure variable.
   If you want to use **CreateDC ( )** to create handle to Device Context of entire screen, then use **CreateDC ( )** *API* as follows :
   HDC hdc ;
   hdc = **CreateDC ("DISPLAY", NULL, NULL, NULL) ;**

¾ Saving the Device Context handle

When you create DC, either by **GetDC ( )** or by **BeginPaint ( )**, window creates all DC related attributes in memory block and returns its reference to you in the form of handle **HDC**.

When you release or delete the Device Context, Windows free the memory block and attributes get lost.

Sometimes in spite of release or deleting DC, you may want to save the attributes of current DC to be used in future. In other words, you want to retain the memory of DC. This can be done by two ways.

1) Go to the window class declaration in **WinMain ( )** , go to the **wndclass.style** and add **CS_OWNDC** by |.
   When you specify this class style, only those DCs, which are retrieved from **GetDC ( )** or **BeginPaint ( )** will affect (means, this style won't affect the DCs created by **CreateWindowDC ( )** & **CreateDC ( )** *API*.)
   This style allows the created DC to be private to each window created with this window class and remains active until the window gets destroyed or you release it explicitly.
2) Go to the window class declaration in **WinMain ( )**, go the **wndclass.style** member and add **CS_CLASSDC** style by |.
   The DC retrieved by this style by using **GetDC ( )** or **BeginPaint ( )** are very similar to **CS_OWNDC** style. The major difference is that, when you create multiple windows of same window class and try to change one of the attribute of one of the Device Context of one of the window, then the change in attributes reflects for all windows of that class, because this style allows the created DC to be public to all windows of the same class.
   Sometimes after doing some work with the current Device Context, you may want to change one of the attributes of the DC then want to do some painting, want to turn back to the previous state of DC. For such type of situations, *API* provides two functions :
   1) SaveDC ( ) whose prototype is
   : **int SaveDC (HDC) ;**

   2) RestoreDC ( ) whose prototype is :
   BOOL RestoreDC (HDC, int) ;
   e.g.

```
// Program……..
….
….
…
/ * Here you want to change a text colour, but before doing that save the current status of the DC, then use :- */
iStatus = SaveDC ( hdc ) ; // declare int iStatus ; before
// Now do some painting with the changed text colour.
…
…
…
/* Now we want to turn back to our previous text, means previous status, then use: - */
RestoreDC ( hdc, iStatus ) ;
```

   You can call **SaveDC ( )** & **RestoreDC ( )** as much time you want, but if you want to restore latest changed status, then no special variable like **iStatus** is required, just call **RestoreDC ( )** with second parameter as **–1**.

¾ How to check device capability ?

When you draw on screen or on printer, you first have to check the default capabilities of two devices. This is necessary to get optimum performance from these two devices.

Windows *API* provides one very important function to accomplish this task and i.e. **GetDeviceCaps ( )**, whose prototype is :

int GetDeviceCaps (HDC, int) ;

HDC is handle to Device Context.

2nd parameter **int** is the index of information you want.

There are about 38 indices of the information. Many are concerned with the printer, but the following 8 are most important and concerned with both devices (screen & printer). These are :

1) **HORZSIZE** **:** Physical width of the screen in terms of millimeters.

2) **VERTSIZE** **:** Physical height of the screen in terms of millimeters.

3) **HORZRES** **:** Horizontal resolution in terms of pixels.

4) **VERTRES** **:** Vertical resolution in terms of pixels.

5) **LOGPIXELSX** **:** These are no. of pixels per logical inch of the screen width.

6) **LOGPIXELSY** **:** These are no. of pixels per logical inch of the screen height.

7) **BITPIXEL** **:** Number of the colour bits per pixel, means this value will be 16-bit, 24-bit, 32-bit, etc.

8) **PLANES** **:** Number of colour planes. These no. of colour planes is multiplied with **BITPIXEL**.

e.g. Suppose your graphic can be more readable in 800 × 600 resolution with 16-bit ore more colours.

Then anticipating that use may work in lower resolution or lower colour, you should use these functions 3 times.

i) To get **HORZRES**.
ii) To get **VERTRES**.
iii) To get colour bits i.e. **BITPIXEL** and display the user, his/her current settings and allow to change to your required settings.

- As getting device capability is the most initial duty of the programmer, these things must be done in **WM_CREATE** message handler.

PEN / BRUSH / FONT

Windows uses some non-**GDI** objects to draw **GDI**. Such non-**GDI** objects are :

A) **Pen** **:** To draw geometrical shapes.
B) **Brush** **:** To fill the geometrical shapes by colour.
C) **Font** **:** To display the text to various format of alphabets.

We will first consider the pen. To create these 3 non-**GDI** objects, there is a common way. :

1. Either use stock object i.e. available in system, e.g. Stock pens, stock brushes, stock fonts.
2. Or Create custom objects (your own). Creating custom objects can be achieved in 2 ways :
    a. Directly by using **Create<object> ( )** *API*.
    b. Indirectly by using **Create<object>Indirect ( )** *API*.

™ **P E N**

The default pen is black coloured given by **BLACK_PEN** macro.

¾ **Using Stock Pens :**

The function is **GetStockObject ( )** which returns handle to **GDI** objects, hence if we want pen, we have to **typecast** the value to **HPEN** type. The code will be as follows :

```
HPEN hMyPen ;
…
…
// code
hMyPen = ( HPEN ) GetStockObject ( WHITE_PEN )
; SelectObject ( hdc, hMyPen ) ;
….
// Do necessary painting
```

[Note about deleting the object : Stock objects need not to be deleted, but custom objects must be deleted. This deletion must be done after releasing or deleting current Device Context (not before). The *API* is **DeleteObject ( )]**

- **List of Stock Pens :**

1. **BLACK_PEN** (by default)
2. **WHITE_PEN**
3. **NULL_PEN** (does nothing)
4. **DC_PEN** (Used mainly by Windows NT & it is supported by Windows 98, but not by Windows 95. This pen is specific for specific Device Context. By default it is white, if you want to change it, you have to use **SetDCPenColor( )** *API*.

¾ **Custom Pen :**

A) <u>Directly by using CreatePen ( ) *API* :</u>

Prototype is :

HPEN CreatePen (int, int, COLORREF) ;

i) 1st parameter is **int** : Pen's style, there are 7 such styles.

PS_SOLID _____

PS_DASH	**----------**
PS_DOT	**……**
PS_DASHDOT	[redacted]
PS_DASHDOTDOT	[redacted]
PS_NULL	Pen is invisible.
PS_INSIDEFRAME	Even if you draw the pen larger than frame width, this option automatically makes it equal to width of the frame.

ii) 2nd parameter is **int :** Width of the pen (thickness of the pen).
By default this parameter is zero i.e. pen is one pixel thick. If you want to use size greater than one, you must specify the pen style as **PS_SOLID**. If you use larger values for other styles, you won't get the desired result, but pen will remain one pixel wide only.

iii) 3rd parameter is **RGB** constant. This function returns handle to the pen of **HPEN** type.
For using it to draw different shapes, we have to select it in current Device Context by using **SelectObject ( )** *API*, whose prototype is :
HGDIOBJ SelectObject (HDC, HGDIOBJ) ;

This function has its 1st parameter as handle to current Device Context and 2nd parameter as handle of the new object, that you want to select in current Device Context . On return, this function gives handle of the old object of the same type.
It means, you can use this function by 2 ways :

1. Using its return value :-
e.g. Suppose you have 2 pens **hRedPen** & **hBluePen**. You created them with **CreatePen ( )** function. First you want to draw with Red Pen & then with Blue Pen & then again with Red Pen. This can be done by following code :

```
// code
SelectObject ( hdc, hRedPen ) ; ……..……(1)
…
…
…
// Do necessary painting
…
…
hRedPen = SelectObject ( hdc, hBluePen ) ; …….. (2)
…
…
// Painting with Blue Pen
…
…
// Switch back to original pen
SelectObject ( hdc, hRedPen ) ;……(3)
```

In the 2nd call to **SelectObject ( )**, we select **hBluePen** as current pen and at the same time, we catch the old pen's handle (**hRedPen**) with return value of it. So that in the 3rd call, we can use the Red Pen again.

2. Without using its return value :-
1st & 3rd call of function in above code are e.g. of this type, in which we ignored the return value.

¾ **DeleteObject ( ) :**

After creating, selecting & using the custom created pen, you have to delete it by **DeleteObject ( )** whose prototype is :
BOOL DeleteObject (HGDIOBJ) ;
If there are many **GDI** objects then, it is better to delete them by using **DeleteObject ( )** in **WM_DESTROY** message handler.

B) Indirectly by using CreatePenIndirect ( ) *API* :

Prototype is :

HPEN CreatePenIndirect (CONST LOGPEN *) ;

This function uses only one parameter i.e. address of variable of **LOGPEN** structure type.

LOGPEN structure declaration :

```
typedef struct tagLOGPEN
{
        UINT            lopnStyle ;     // Style of the pen
        POINT           lopnWidth ;     // Width of the pen
        COLORREF        lopnColor ;     // RGB Colour constant
} LOGPEN ;
```

e.g.

```
HPEN hMyPen ;
LOGPEN lp ;

// filling the structure members
lp.lopnStyle = PS_SOLID ;
lp.lopnWidth = 2 ;
lp.lopnColor = RGB ( 255, 0, 0 ) ;
hMyPen = CreatePenIndirect ( &lp ) ;
SelectObject ( hdc, hMyPen ) ;
…
…
// Do necessary painting
…
…
DeleteObject ( hMyPen ) ;
```

** **Imp. Note :** About the **POINT** structure variable **PenWidth** : We saw **CreatePen ( )** uses pen width as integer type parameter, but the same entity when becomes the member of **LOGPEN** structure, it is of **POINT** type. This gives you one advantage that, x & y members of **POINT** structure. You can give more dimensions to the pen, useful in 3D drawing in **OpenGL** & **DirectX**.

™ BRUSH

It is the non-**GDI** object used to do a **GDI** function **'filling the shapes'** by a specific colour. Note that brushes are nothing but 8 pixels × 8 pixels bitmaps. The method of creating, using & deleting the pens.

¾ Using Stock Brushes :

The same **GetStockObject ( )** *API* is used to create brush. Available stock brushes are :

i)	**BLACK_BRUSH**	
ii)	**WHITE_BRUSH**	**(Default brush)**
iii)	**DECAYGRAY_BRUSH**	**(Dark)**
iv)	**LTGRAY_BRUSH**	**(Light)**
v)	**HOLLOW_BRUSH**	**(Transparent)**
vi)	**DC_BRUSH**	**(Mainly used in Win NT)**

The method of creating & using stock brush is same as pen :

```
HBRUSH hMyBrush ;
…
…
hMyBrush = ( HBRUSH ) GetStockObject ( WHITE_BRUSH )
; SelectObject ( hdc, hMyPen ) ;
…
…
// do necessary painting
```

¾ Using Custom Brush :

There are about 9 custom brush creating *API* in MSDN, out of them 4 are important. For others, see MSDN. The important 4 are :

1. **CreateSolidBrush ( )**
2. **CreateHatchBrush ( )**
3. **CreatePatternBrush ( )**
4. **CreateBrushIndirect ( )**

Out of above 4, first 3 are for Direct method and 4th is for Indirect method.

1. CreateSolidBrush ( ) :

Prototype is :
HBRUSH CreateSolidBrush (COLORREF) ;

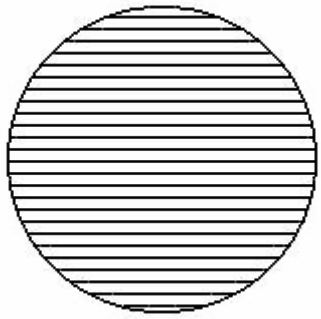
The only parameter is **RGB** constant. The return value is handle to the brush.

2. CreateHatchBrush ( ) :

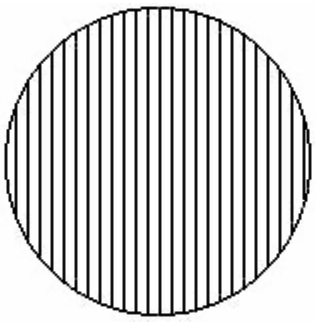
Creates a hatched brush, means the brush strokes are having gaps in-between.
Prototype is :
HBRUSH CreateHatchBrush (int, COLORREF) ;

The first parameter is integer constant of hatch style. There are six known hatch styles.

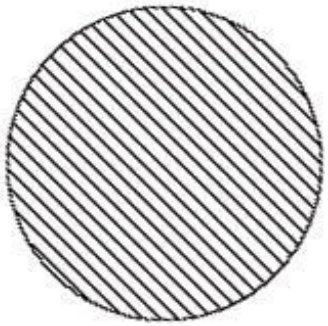
1. HS_HORIZONTAL :- (HS stands for Hatch Style)



2. HS_ VERTICAL :

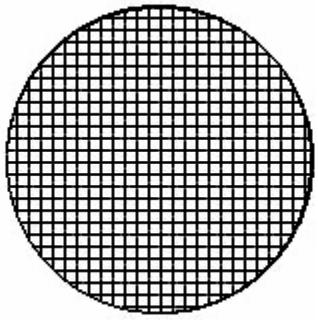


3. HS_FDIAGONAL :

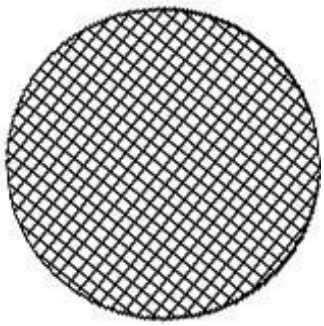


4. HS_BDIAGONAL :

5. HS_CROSS (Combination of HS_VERTICAL & HS_HORIZONTAL) :



6. HS_DIAGCROSS (Combination of HS_EDIAGONAL & HS_BDIAGONAL)



The 2nd parameter is as usual **RGB ( )** constant.

Return value is handle to the brush.

3) CreatePatternBrush ( )

This function creates brush with a specified bitmap pattern.

Note that : Before creating such type of brush, you must have handle to the bitmap which you are going to use as pattern to this brush.

The prototype is :

HBRUSH CreatePatternBrush (HBITMAP) ;

The only parameter is the handle to the bitmap which you are going to use as pattern of this brush. The return value is handle to the brush.

[**Note:** The **BMP** used here is **D**evice **D**ependent **B**itmap (**DDB**), and not **D**evice **I**ndependent **B**itmap (**DIB**). You may surprise that, why such a type of brush is needed, means making a bitmap of colour, then using **CreatePatternBrush ( )** is useless, because **CreateSolidBrush ( )** using this function is to create any user-desired 8 × 8 bitmap (not just restricted to the colour) and then use it with **CreatePatternBrush ( )** to point the whole desktop as **Windows Desktop Properties** sheet makes tiles of a bitmap.]

4) CreateBrushIndirect ( )

As pen's indirect function, this function also uses only one parameter as address of variable of **LOGBRUSH** structure. The surprising thing is that, using this single function you can create any of the above three brushes. The structure is defined as follows :

```
typedef struct tagLOGBRUSH
{
        UINT            lbStyle ;// Brush style, explained later.
        COLORREF        lbColor ;// Colour of the brush ( Beginning with "BS" )
        LONG            lbHatch ;// Hatch style as seen before ( Beginning with
                                    "HS" )
} LOGBRUSH ;
```

As the first member **lbStyle**, there are nine brush styles defined :

1) **BS_DIBPATTERN (BS stands for Brush Style): Device Independent Bitmap** pattern brush.
2) **BS_DIBPATTERN8*8 : Device Independent Bitmap** pattern brush, only the bitmap size is 8 × 8.
3) **BS_DIBPATTERNPT : Device Independent Bitmap** pattern brush, using point instead of pixels.
4) **BS_HOLLOW :** Just like **HOLLOW** stock brush.
5) **BS_NULL :** Just like **NULL** stock brush.
6) **BS_DDBPATTERN :** Using **Device Dependent Bitmap** pattern brush.
7) **BS_DDBPATTERN8*8 : Device Dependent Bitmap** pattern brush, only the bitmap size is 8 × 8.
8) **BS_SOLID :** Solid colour brush.
9) **BS_HATCHED :** Hatched colour brush.

Out of above 9 styles, only 3 are important to us, which resemble 3 direct brush creating functions : **1) BS_SOLID 2) BS_HATCHED 3) BS_PATTERN.**

1) **BS_SOLID :** When use this value with **lbStyle** member of **LOGBRUSH** structure, the created brush is solid brush like **CreateSolidBrush ( )** *API*. Obviously, in this case **lbHatch** member is ignored. Hence it is better to assign zero to it.
2) **BS_HATCHED :** When this value is used with **lbStyle** of **LOGBRUSH** structure, then the created brush is the hatched brush, similar to **CreateHatchBrush ( )** *API*. In this case, **lbHatch** member is important and you have to specify one of the six **"HS"** value.

Code :

```
// Creating Yellow coloured cross diagonal hatched brush, using Indirect Way.

        HBRUSH hMyBrush ;
        LOGBRUSH lb;

        lb.lbStyle = BS_HATCH ;
        lb.lbColor = RGB ( 255, 255, 0 ) ;
        lb.lbHatch = HS_DIAGCROSS ;

        hMyBrush = CreateBrushIndirect ( &lb) ;
        SelectObject ( hdc, hMyBrush ) ;
        …
        …
        // Painting
        …
        …
        EndPaint ( hdc ) ;
        DeleteObject ( hMyBrush ) ;

// Creating Yellow coloured cross diagonal hatched brush, using Direct Way.
        HBRUSH hMyBrush ;
        HMyBrush = CreateHatchBrush ( HS_DIACROSS, RGB ( 255, 255, 0 ) ) ;
        SelectObject ( hdc, hMyBrush ) ;
        …
        …
        // Painting
        …
        …
        EndPaint ( hdc ) ;
        DeleteObject ( hMyBrush ) ;

// Above program creates hatched yellow & diacross type brush.
```

™ FONT

There are another non-**GDI** objects used to display the characters. As like other two objects, they can also be used by two ways :

1) **Stock Fonts**
2) **Custom Fonts : i)** By direct method **CreateFont ( )** *API*.
   ii) By indirect method **CreateFontIndirect ( )** *API*, which used **LOGFONT** structure.

¾ Information about fonts :

Though fonts are similar to other two non -**GDI** objects, Pen & Brush, they are present, virtually in all programs, hence system has rich resources about them. Thus fonts deserve some more explanation.

Logically speaking, fonts are nothing but bitmap images of alphabets which are the most commonly used **TrueType Fonts** (**TTF**). Windows O.S. supports many type of fonts in English as well as many other international languages, to which we call multilingual support. Fonts have particular name (also called as TypeFace) and it also have particular size in width and in height.

Broadly, fonts are classified into two types :

1) **GDI Fonts.**
2) **Device Fonts.**

1) **GDI Fonts : GDI** fonts are available as files, e.g. **.FON**, **.TTF**, **.LPT**, **.TIF** which are stored on disk, mainly in **Windows\System** & **Windows\Fonts** directories.

2) **Device Fonts :** Located on output devices (hardwired as ROM) such as Printers, Plotters.

Further **GDI** fonts are sub-classified into three types :

1) **Raster Fonts**
2) **Stroke Fonts (Vector Fonts)**
3) **TrueType Fonts**

1) **Raster Fonts :**
   Raster fonts are bitmap picture fonts; hence if you try to change their size, shape & orientation, then they get distorted. Thus they have poor scalability. Thus called as non-scalable fonts, but they have great advantage of faster performance of faster display. As they are hand-made bitmap picture, they are more readable.

2) **Stroke Fonts :**
   These fonts are created by connecting the dots. They are completely scalable, but have poor performance of display and much of the readability gets lost for small sizes. They are mainly made for plotters, hence also called as **Plotter Fonts** or **Vector Fonts**.

3) **TrueType Fonts :**
   These are fonts mainly of programmer's interest. They are created by merging the best properties of above two, while removing the poor properties from them. They are first created by **Apple Computers**, then idea is more widened by the collaboration of **Apple** & **Microsoft**. They are also termed as "**Output Fonts**", means instead of joining the

dots, they are created by joining the pixels by lines and curves and thus they are much scalable & can be used for both, Video Display & Printer.
When you demand these fonts to O.S., O.S. does rasterisation of them, means converts them to bitmap and hence performance is also satisfactory.
Windows 95 supports 13 types of **TrueType** fonts. Their **TypeFaces** are as follows :

01) Courier New
2) Courier New Bold
3) Courier New Italic
4) Courier New Bold Italic
5) Times New Roman
6) Times New Roman Bold
7) Times New Roman Italic
8) Times New Roman Bold Italic
9) Arial
10) Arial Bold
11) Arial Italic
12) Arial Bold Italic
13) Symbol

Out of above 13 TypeFaces, first 12 TypeFaces are for Alphabetical Fonts and 13th is for Symbolic Font.

Besides this Windows 95 system support, six font families :

1) **FF_DONTCARE**
2) **FF_SWISS**
3) **FF_MODERN**
4) **FF_ROMAN**
5) **FF_SCRIPT**
6) **FF_DECORATIVE**

These families are important because they determine the pitch (width) of the character.

¾ Using Stock Fonts :

Method is same as using stock pens & stock brushes. System supports six types of stock fonts. :

1) **SYSTEM_FONT (is default)**
2) **SYSTEM_FIXED_FONT**
3) **OEM_FIXED_FONT**
4) **DEFAULT_GUI_FONT**
5) **ANSI_FIXED_FONT**
6) **ANSI_VAR_FONT**

(OEM : Original Equipment Manufacturer, VAR : Variable)

Code :

```
HFONT hMyFont ;
…
…
hMyFont = ( HFONT ) GetStockObject ( DEFAULT_GUI_FONT ) ;
SelectObject ( hdc, hMyFont ) ;
…
…
// Do necessary painting
…
…
EndPaint ( hdc ) ;
```

As like pen & brush, you also can use **SelectObject ( )** for dual purpose.

¾ Using Custom Fonts :

1) By Using CreateFont ( ) *API* – The Direct Way :

Prototype is :

HFONT CreateFont (int, int, int, int, int,
DWORD, DWORD, DWORD, DWORD,
DWORD, DWORD, DWORD, DWORD,
LPCTSTR) ;

1) 1st parameter is **int nHeight :** Height of the font. If zero, system uses default height i.e. 8 × 8 pixels.

2) 2nd parameter is **int nWidth :** Width of the font. If zero, Windows calculates itself, according to current aspect ratio of screen resolution.

3) 3rd parameter is **int nEscapement :** This parameter is used, when you want to change the default horizontal behavior of font, means when you want to rotate the font. Thus this parameter is nothing but angle of rotation, positive for clockwise & negative for anti-clockwise. The angle is supplied by multiplying your desired angle with 10. e.g. If you want to display a text in vertical form, you have to rotate it in ± 90° with aspect to the default horizontal behavior, then instead of supplying ± 90, pass ± 90 × 10 = ± 900. For default horizontal position, it will be zero.

4) 4th parameter is **int nOrientation :** It is quite similar with the **nEscapment**, but the **nEscapement** applies to whole string, while using **nOrienation**, you can rotate a single character, remaining logic is same as **nEscapment.**

5) 5th parameter is **int nWeight :** Weight of the character. The default is zero. Value can ranged from 0 to 1000. For normal, used weight is 400. For bold, it is 700. But to make this task more easier, Windows defines 10 macros for this parameter. :

i) FW_DONTCARE (FW stands for Font Weight)
ii) FW_THIN
iii) FW_EXTRALIGHT
iv) FW_LIGHT
v)FW_NORMAL (400)
vi) FW_MEDIUM
vii) FW_SEMIBOLD

viii) FW_BOLD (700)
ix) FW_EXTRABOLD
x)FW_HEAVY (1000)

6) 6th is **DWORD fdwItal (fdw stands for font double word) :**
This specifies *Italic Style*. The default is zero, means non-italic style. If you want italic style, then give non-zero value.

7) 7th is **DWORD fdwUnderline :** This specifies Underline. The default is zero, means no underline. If you want underline, then give non-zero value.

8) 8th is **DWORD fdwStrikeOut :** This specifies Strikeout. (e.g. ~~NAME~~) The default is zero, means no strikeout. If you want strikeout, then give non-zero value.

9) 9th is **DWORD fdwCharSet :** Which character set you want to use, there are many character sets. Some of them are :
- **i) ANSI_CHARSET**
- **ii) DEFAULT_CHARSET**
- **iii) OEM_CHARSET**
- **iv) SHIFTJIS_CHARSET** , etc.

10) 10th is **DWORD fdwOutputPrecision :** This parameter determines, how closely matching of output and the font characteristics should be done. There are many values, the default is **OUT_DEFAULT_PRECISION**. For others, see **MSDN**.

11) 11th is **DWORD fdwClipPrecision :** This parameter specifies, the clipping precision during painting, there are many options. The default is **CLIP_DEFAULT_PRECISION**.

12) 12th parameter is **DWORD fdwQuality :** This parameter specifies the output quality. The default is **DEFAULT_QUALITY**. Remaining two are:
- i) **DRAFT_QUALITY**
- ii) **PROOF_QUALITY**.

13) 13th parameter is **DWORD fdwPitchAndFamily :** This parameter speficies the Pitch and the family of the font. There are six pitch values :
- **i) FF_DONTCARE**
- **ii) FF_SWISS**
- **iii) FF_MODERN**
- **iv) FF_ROMAN**
- **v) FF_SCRIPT**
- **vi) FF_DECORATIVE**

You can use these values, one from pitch & one from family using | (bar), but while comparing this parameter with another value, you have to do it with single "**&**" as explained in "**TEXT**" program

14) 14th parameter is **LPCTSTR lpszFace :** This is the parameter where you specify, the **TypeFace** names, length of which must not be greater than 32 characters including a NULL character.

Code :

```
HFONT hMyFont ;
…
…
…
hMyFont = CreateFont ( int, int, int, int, int, DWORD, DWORD, DWORD, DWORD,
                       DWORD, DWORD, DWORD, DWORD, LPCTSTR ) ;
…
…
…
SelectObject ( hdc, hMyFont ) ;
…
…
// Do necessary painting
…
…
EndPaint ( hdc ) ;
DeleteObject ( hMyFont ) ;
```

2) By Using CreateFontIndirect ( ) *API* – The Indirect Way :

This function uses **LOGFONT** structure. The prototype is :

HFONT CreateFontIndirect (LOGFONT *) ;

As a parameter, this function uses address of variable of **LOGFONT** structure. On return, it gives handle to font object. **LOGFONT** structure is defined as follows :

```
typedef struct tagLOGFONT
{
        LONG        lfHeight ;
        LONG        lfWidth ;
        LONG        lfEscapement ;
        LONG        lfOrientation ;
        LONG        lfWeight ;
        BYTE        lfItalic ;
        BYTE        lfUnderline ;
        BYTE        lfStrikeOut ;
        BYTE        lfCharset ;
        BYTE        lfOutputPrecision ;
        BYTE        lfClipPrecision ;
        BYTE        lfQuality ;
        BYTE        lfPitchAndFamily ;
        CHAR        lfFaceName [ LF_FACESIZE ] ;
// LF_FACESIZE is constant defined in windows.h, size = 32-bit.
} LOGFONT ;
```

Code :

```
HFONT hMyFont ;
LOGFONT lf ;
…
…
…
lf.lfHeight = 3 ;
…
…
…
/* All members, should be assigned to required values, only last member
should be declared as follows : */
strcpy ( lf.lfFaceName, "Times New Roman" ) ;
hMyFont = CreateFontIndirect ( &lf ) ;
SelectObject ( hdc, hMyFont ) ;
…
…
…
EndPaint ( hdc ) ;
DeleteObject ( hMyFont ) ;
```

- **GetObject ( )**

Prototype is :

int GetObject (HDIOBJ, int, LPVOID) ;

1st parameter is handle of the graphic object about which you need information.
2nd parameter is obviously, before getting object, you already know what type of object you want to get, hence you have to first declare an empty variable of **LOG____** structure of that object & use **sizeof** operator as this parameter.
3rd parameter is address of the variable of the **LOG____** structure type of the object with casted to **(LPVOID)**. This function mainly used to get information about stock objects, because neither Windows nor **MSDN** give properties of stock objects. Only this function can help you to get those properties.

e.g. Suppose you want properties of **SYSTEM_FONT** font's TypeFace name, code is as follows:

```
HFONT hMyFont ;
LOGFONT lf ;
…
…
…
hMyFont = ( HFONT ) GetStockObject ( SYSTEM_FONT ) ;
GetObject ( hMyFont, sizeof ( LOGFONT ), ( LPVOID ) &lf ) ;
…
…
…
// Now O.S. will fill 14 properties of SYSTEM_FONT in &lf ( LPVOID )
```

- There is yet another way of creating font. The way is **"Font Common Dialog Box"** (Dialog boxes given by system to us. We will see how to use common Dialog Boxes in future.

MAPPING MODES

Explained in Details….

The study of mapping modes, includes a study of five entities :

1) **Mapping Modes**
2) **Window Origin**
3) **Viewport Origin**
4) **Viewport Extent**

All above entities are actual attributes of Device Context.

¾ What is Mapping Mode ?

Mapping mode is the method by which Windows O.S. transforms **logical co-ordinates** into **device co-ordinates** in some definite scalable manner. In other words, we can say that **Mapping Mode** is scalable transformation of logical co-ordinates into device co-ordinates & vice versa.

¾ What is Mapping Mode ?

We use **TextOut ( )**, by passing X, Y co-ordinates to it, where X stands for distance of a character from X axis, while Y stands for distance of a character from Y axis. These co-ordinates that we passed are "**our understandable**" co-ordinates. Actual display drives (or say Video Card) cannot understand these units in the same way. So it is the duty of Windows O.S. to convert human understandable co-ordinates into actual device co-ordinates. This duty of Windows O.S. is possible due to **Mapping Modes**. O.S. gives you the freedom of choosing any available **Mapping Mode** to change the look and feel of your display.

Windows define following types of **Mapping Modes.** There are in all 8 types :

1) **MM_TEXT**
2) **MM_LOMETRIC**
3) **MM_HIMETRIC**
4) **MM_LOENGLISH**
5) **MM_HIENGLISH**
6) **MM_TWIPS**
7) **MM_ISOTROPIC**
8) **MM_ANISOTROPIC**

¾ How to get & set Mapping Mode ?

The default **Mapping Mode** is **MM_TEXT**. You can get or set **Mapping Mode** as per your desire, by using **GetMapMode ( )** and **SetMapMode ( )**, respectively.

¾ **GetMapMode ( ) :**

Prototype is :

int GetMapMode (HDC) ;

This function receives handle to the current Device Context as its only parameter and returns current **Mapping Mode** as integer value.

¾ **SetMapMode ( )**

: Prototype is :

int SetMapMode (HDC, int) ;

1st parameter is the handle to the Device Context.

2nd parameter is the **Mapping Mode** that you want to set, while returning this function gives previous **Mapping Mode**. So you can use this single for both, getting & setting of **Mapping Mode**.

¾ **What is Window Origin & Window Extent ? How to get & set them ?**

This term "**Window**" is must confusing term in the terminology of **Mapping Modes.** Here the term "**Window**" not all concerned with our usual frame window which we used to see on screen. Instead, the term "**Window**" means **Logical Co-ordinate System**. Obviously, **Window Origin** is "**Logical Co-ordinate System Origin**" and **Window Extent** is "**Logical Co-ordinate System Extent**". The unit can be anything that O.S. define, means in Windows, this unit can be either pixels or millimeters or inches.

- To get the **Window Origin**, we use the function **GetWindowOrgEx ( )** whose **prototype** is :

  BOOL GetWindowOrgEx (HDC, LPPOINT) ;

1st parameter **HDC** is handle to the current Device Context.

2nd parameter is the address of variable of **POINT** structure type. On return, function gives successful or failure statues as TRUE or FALSE, respectively.

While using this function you should pass address of an empty **POINT** structure variable. So on return, system will fill it with proper logical co-ordinate system origin (Window) co-ordinate.

- To set **Window Origin**, we have to use the function **SetWindowOrgEx ( )** whose **prototype** is :

  BOOL SetWindowOrgEx (HDC, int, int, LPPOINT) ;

1st parameter **HDC** is handle to the current Device Context.

2nd parameter is X co-ordinate of the new **Window Origin** that you want to set.

3rd parameter is Y co-ordinate of the new **Window Origin** that you want to set.

4th parameter is **POINT** structure type variable, which is initially empty when passed & System fills it with X & Y co-ordinates of the old **Window Origin**.

- To get the **Window Extent**, we should use **GetWindowExtEx ( )** whose **prototype** is :

  BOOL GetWindowExtEx (HDC, LPSIZE) ;

1st parameter **HDC** is handle to the current Device Context.

2nd parameter is address of **SIZE** structure type variable, when passed it should be empty, System will fill it with current **Logical Co- ordinate System's** (**Window**) X **Extent** & Y **Extent** i.e. distance between the right border & left border and the distance between the bottom border & top border.

- To set the **Window Extent**, we should use **SetWindowExtEx ( )** whose **prototype** is :
  BOOL SetWindowExtEx (HDC, int, int, LPSIZE) ;

1st parameter is **HDC** is handle to the current Device Context.
2nd parameter is X co-ordinate of the new **Logical Co-ordinate System's** (**Window**) Extent that you want to set.
3rd parameter is Y co-ordinate of the new **Logical Co-ordinate System's** (**Window**) Extent that you want to set.
4th parameter is **SIZE** structure type variable, which is initially empty, when passed & System fills it with X & Y co-ordinates of the old **Window Extent**.

¾ What is Viewport Origin & Viewport Extent ? How to get & set them ?

The term "**Viewport**" is another confusing term in the terminology of **Mapping Modes**. Here the term "**Viewport**" means the "**Actual Device Co-ordinate System**". So obviously, **Viewport Origin** is the origin of **Device Co-ordinate System** and **Viewport Extent** is Extent of **Device Co-ordinate System**. Here unit is fixed and it is fixed.

- To get **Viewport Extent**, we use the *API* function **GetViewportOrgEx ( )** whos **prototype** is :
  BOOL GetViewportOrgEx (HDC, LPPOINT) ;

1st parameter **HDC** is handle to the current Device Context.
2nd parameter is address of variable of **POINT** structure type.
On return, function gives successful or failure status as TRUE or FALSE, respectively.
While using this function, you should pass an empty **POINT** structure variable, so that on return, System will fill it with proper **Device Co-ordinates** of **Viewport Origin**.

- To set **Viewport Origin**, we use the *API* function **SetViewportOrgEx ( )** whos **prototype** is :
  BOOL SetViewportOrgEx (HDC, int, int, LPPOINT) ;

1st parameter **HDC** is handle to the current Device Context.
2nd parameter is X co-ordinate of new desired origin.
3rd parameter is Y co-ordinate of new desired origin.
4th parameter is address of variable of **POINT** structure type which is then filled by O.S. with X & Y co-ordinates of previous **Viewport Origin**..
Note that if you already have some values in this structure variable, then O.S. overwrites them.

- To get **Viewport Extent**, we use function **GetViewportExtEx ( )** whose **prototype** is :
  BOOL GetViewportExtEx (HDC, LPSIZE) ;

1st parameter is handle to the current Device Context,
2nd parameter is address of **SIZE** structure type variable, when passed, it should be empty.

On return, System will fill it with current X & Y extents of **Device Co-ordinate System**.

- To set **Viewport Extent**, we should use **SetViewportExtEx ( )** whose **prototype** is :
  BOOL SetViewportExtEx (HDC, int, int, LPSIZE) ;

1st parameter is handle to the current Device Context.
2nd parameter is the new **X Extent**.
3rd parameter is the new **Y Extent**.
4th parameter is variable of **SIZE** structure type, which is initially empty/filled.
On return, System fills it with **X** & **Y** Extents of previous **Viewport Extent**.

Device Co-ordinate System comes in three flavors :

1. **Screen's Device Co-ordinate System**
2. **Whole Window's Device Co-ordinate System**
3. **Client Area's Device Co-ordinate System**

1. Screen's Device Co-ordinate System :

This applies to whole screen (in other words DESKTOP). Its default origin is located at left top corner of the screen. You can get handle to Device Context for entire screen by using one of the following two methods :

a) **hdc = CreateDC ("DISPLAY", NULL, NULL, NULL) ;**
b) **hdc = GetDC (NULL) ;**

Window itself, uses this co-ordinate system for setting origin and setting extent of newly created window (but not the child window) in **CreateWindow ( )**.

2. Whole Window's Device Co-ordinate System :

This applies to whole window. Its default origin is at the junction of left and top window borders. To get the handle to Device Context of whole window, we use **GetWindowDC ( )**.

3. Client Area's Device Co-ordinate System :

This applies to client area only, means its default origin is located at the junction of left window border & bottom border of caption bar (when menu is absent) or bottom border of menu bar (when menu is present) . To get the handle to Device Context of client area, we use either **GetDC ( )** or **BeginPaint ( )**.

¾ Inter-conversion functions between these **Device Co-ordinate System** :

a) To get the **Window**'s dimension (frame window), we can use **GetWindowRect ( )** whose **prototype** is :
BOOL GetWindowRect (HWND, LPRECT
) 1st parameter is handle to window.
2nd parameter is address of variable of **RECT** structure type, when passed it is empty. On return, it is filled with **Window**'s current dimensions.

b) To get **Client Area**'s dimension, we can use **GetClientRect ( )** whose **prototype** is :
BOOL GetClientRect (HWND, LPRECT)
; 1st parameter is handle to **Client Area**.
2nd parameter is address of variable of **RECT** structure type, when passed it is empty.
On return, it is filled with **Client Area**'s current dimensions.

c) To convert **Client Area**'s co-ordinates into **Screen** co-ordinates and to convert **Screen** co-ordinates into **Client Area**'s co-ordinates, **ClientToScreen ( )** & **ScreenToClient ( )** functions are used respectively..

- **ClientToScreen ( )**

Prototype is :

BOOL ClientToScreen (HWND, LPPOINT) ;

Its 1st parameter is handle of the window, whose **Screen** co-ordinates you want.

2nd parameter is address of **POINT** type variable. When passed, it has **Client Area**'s co-ordinates. On return, system wipes off these values and fills it with converted **Screen** co-ordinates.

- **ScreenToClient ( )**

Prototype is :

BOOL ScreenToClient (HWND, LPPOINT) ;

Its 1st parameter is handle of the window, whose **Client Area**'s co-ordinates you want.

2nd parameter is address of **POINT** type variable. When passed, it has **Screen** co-ordinates. On return, system wipes off these values and fills it with converted **Client Area**'s co-ordinates.

¾ How Window Does Window To Viewport & Viewport To Window Mapping ?

As explained before, we said that a program used to pass **Logical** co-ordinates (called as **Window**) to which O.S. maps converts/translates/transforms into **Actual Device** co-ordinates (called as **Viewport**).

Windows O.S. does this thing by using following mathematical formulae

:-1) For **Window** to **Viewport** Conversion :

xViewport = (xWindow – xWindowOrg) * xViewportExt / xWindowExt + xViewportOrg
yViewport = (yWindow – yWindowOrg) * yViewportExt / yWindowExt + yViewportOrg

2) For **Viewport** to **Window** Conversion :

xWindow = (xViewport – xViewportOrg) * xWindowExt / xViewportExt + xWindowOrg
yWindow = (yViewport – yViewportOrg) * yWindowExt / yViewportExt + yWindowOrg

¾ Can we do above things on our own ?

For Case 1 means, for **Window** to **Viewport** Conversion, the *API* is **LPtoDP ( )**.

Prototype is :

BOOL LPtoDP (HDC, LPPOINT, int) ;

1st parameter is handle to the current Device Context.

2nd parameter is address of the array of POINT type, values in which you want to translate.

3rd parameter is number of array elements.

As this function is array based, you can pass any number of Logical co-ordinates (Window) to get converted into Device co-ordinates (Viewport).

For Case 2 means, for **Viewport** to **Window** conversion, the *API* is **DPtoLP ( )**.

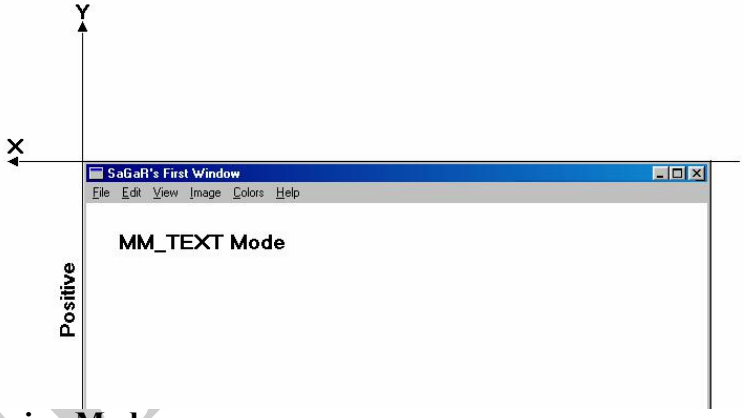
Prototype is :

BOOL DPtoLP (HDC, LPPOINT, int) ;

1st parameter is handle to the current Device Context.
2nd parameter is address of the array of POINT type, values in which you want to translate.
3rd parameter is number of array elements. As this function is array based, you can pass any number of Device co-ordinates (Window) to get converted into Logical co-ordinates (Window).

¾ Brief description of every Mapping Mode :

1) **MM_TEXT :** This is the the default **Mapping Mode** & most commonly used. The word TEXT indicates that this **Mapping Mode** is mainly used for display of text, because this mode follows our usual text reading pattern. As we read text from left to right & top to bottom, its X axis is located at top border of client area and its Y axis is located at left border of client area. As like reading pattern, its values of X are increasing from left to right & values of Y are increasing from top to bottom.
Above discription also shows that, its origin (0,0) will be left top corner of client area.



‰ **Metric Mapping Modes**

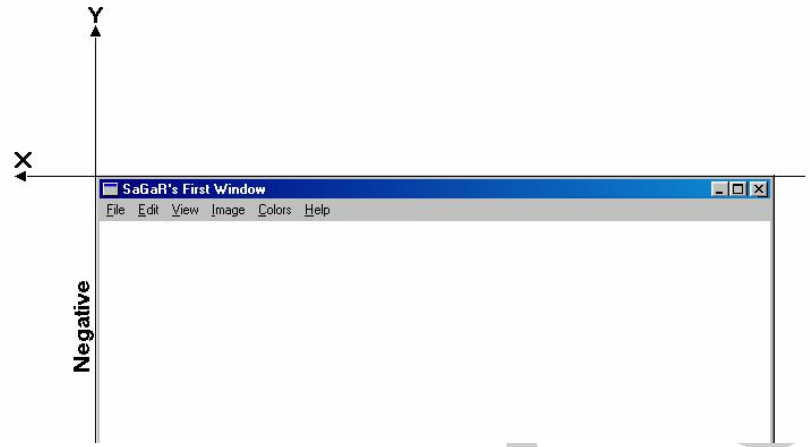
The five modes are called as Metric **Mapping Modes** :

1. MM_LOMETRIC : Logical unit is 0.1 mm (1/10th of millimeters)
2. MM_HIMETRIC : Logical unit is 0.01 mm (1/100th of millimeters)
3. MM_LOENGLISH : Logical unit is 0.01 inches (1/100th of inches)
4. MM_HIENGLISH : Logical unit is 0.001 inches (1/1000th of inches)
5. MM_TWIPS : It is acronym (short form) for 20th of point. Hence TWIPS. A unit of point is 1/72th of inch.
(1/72th means in terms of inches : 1/20 * 72 th = 1/1440th of inch)

‰ While using these **Mapping Modes** in your program, you must think in the unit that you specify, means suppose you specify 0.1 mm as **LOMETRIC** while drawing a

figure, you use 5000 as X co-ordinate, then converting it into inches, you can see that it is beyond the width of physical screen dimensions and thus can not be mapped.

- About axes of these **Mapping Modes** : X axis and its value is same as **MM_TEXT**, but though position of Y axis is same as **MM_TEXT**, the co-ordinates are negative.



- **MM_ISOTROPIC :**

ISOTROPIC means equal in all direction, so this **Mapping Mode** has equally scale axis in all directions (unlike **MM_TEXT** & **Metric Mapping Mode**). This mode gives you much freedom, because you can change **Window Origin**, **Window Extent**, **Viewport Origin**, **Viewport Extent**.
The common way of using this **Mapping Mode** is as follows :

1. Set this **Mapping Mode** by using **SetMapMode ( )**.
2. Set **Window Extent** to maximum of the Logical co-ordinate unit (32,767).
3. Set **Viewport Extent** to **cxClient** & **cyClient**.
4. Set **Viewport Origin** at you desired position. e.g. If you want your **Viewport Origin** at the center of the screen, say…..
   SetViewportOrgEx (hdc, cxClient/2, cyClient/2, NULL) ;

- **MM_ANISOTROPIC :**

ANISOTROPIC means not all equally in any direction, thus this mode gives you full freedom for your drawing. You can set or change **Window Origin**, **Window Extent**, **Viewport Origin**, **Viewport Extent**. But the drawback is that in **MM_ISOTROPIC,** if you enlarge or shrink a drawing, its get enlarged or shrink by keeping aspect ratio constant, hence no image distortion takes place. But in **MM_ANISOTROPIC** as there is no equal scaling, the enlarging or shrinking of image may give distorted results. e.g. Suppose, we want out text to be displayed DOS like, do following:

```
SetMapMode ( hdc, MM_ANISOTROPIC ) ;
SetWindowExtEx ( hdc, 1, 1, NULL ) ;
SetViewportExtEx ( hdc, cxChar, cyChar, NULL ) ;
TextOut ( hdc, 3, 2, "Hello, Windows 98 !!" ) ;
```

This **TextOut ( )** will display string 3 spaces away from the left & 2 spaces away from the top, which is just like using **printf ( )** in C.

There are 3 *API*s.

GetWorldTransform ( ) , ModifyWorldTransform ( ), SetWorldTransform ( )
These are mainly used while using fonts and some graphics.

MSDN : World transform a specification of an international method of transforming world space to page space. These are required for internationalization of yours application with UINICODE, mainly used by NT.

SHAPES

To use of **GDI** function is mainly targeted to two tasks :

1. **Fonts**
2. **Graphical Shapes**

The most common shapes which can be drawn using **GDI** functions are :

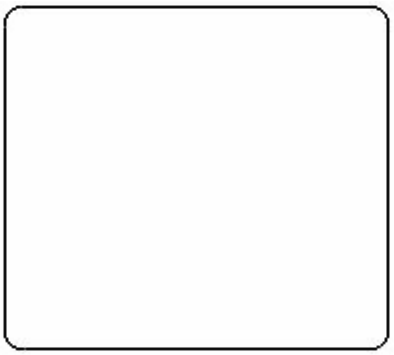
1. **Lines : a) Linear b) Polyline c) Spline & Bezier**
2. **Rectangles : a) Plane b) Rounded c) Square**
3. **Ellipse : a) Plane b) Circle**
4. **Curves : a) Arc b) Chord c) Pie**
5. **Polygon :** Arc closed linear shape with more than 2 edges is a polygon. (Rectangle is also a polygon) : **a) Triangle b) Hexagon c) Octagon**, etc.
6.
   - **Some Common Basics :**

1) As Video Device always works in terms of pixels, exact drawing of curves, rounds, diagonals cannot be drawn. Though they look drawn correctly, they are not. We can see this by magnifying the drawn shape by magnifying tool in Paint Brush (MSPAINT) program. The magnified shape that curves and diagonals are actually made of small rectangles.

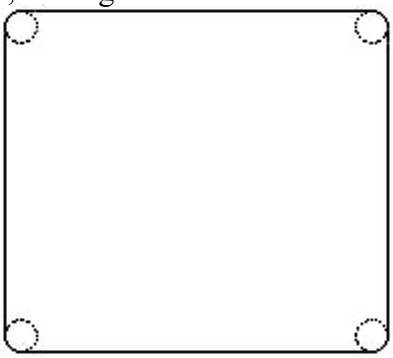
2) On other hands, the linear shape i.e. plane rectangles, plane lines are drawn very correctly & their magnification also looks very smooth.

3) So we can have conclusion that video works in terms of rectangles, hence any curved shapes like ellipse, circle, chord, pie, arc are also drawn inside the bounded rectangle, means when we draw ellipse, we actually pass co-ordinates of four corners of rectangle inside which ellipse is drawn though we cannot see our rectangle. Similarly circle is drawn inside a square, thus almost every *API* (except lines) uses left, top, right, bottom co-ordinates as their parameters.

4) Now think about a rounded rectangle. In terms of Windows Graphics, we see a rounded rectangle with four borders & curved corners, like below figure :



But in Windows Graphics, this is combination of 4 borders & 4 ellipses like corners, like figure below :



5) Another very important point about Windows Graphics is that, whenever you give an ending point of shape, O.S. draws that shape up to co-ordinates of that ending point, but not including those co-ordinates. This is because of the default minimum width of graphic pen which is one pixel. Thus when you specify draw a line from (10,5) to (100,5), the actual line is drawn from (10,5) to (99,5), because width of the pen is one pixel and hence 100 is gained by external edge of that pixel. If this logic was not provided, there might be problems in accuracy because if the end point co-ordinates are included the thickness of pixel may advance by one pixel, means line drawn from (10,5) to (100,5) will get mistakenly drawn from (10,5) to (101,5) which is not a user specification.

¾ **Drawing Lines :**

Line drawing requires use of two functions :

1. **MoveToEx ( ) :** Which moves current location to the desired locations from you want to draw the line (means it is origin of the line).
   Prototype is :
   BOOL MoveToEx (HDC, int, int, LPPOINT) ;

   1st parameter is handle to the current Device Context.
   2nd & 3rd parameters are the co-ordinates of point, which you want to use as origin of line. 4th parameter returns the previous position, from which you are moving now.

2. **LineTo ( ) :** This function actually draws line from the origin (given by **MoveToEx ( )**) to the destination point.
   Prototype is :

BOOL LineTo (HDC, int, int) ;

1st parameter is handle to the current Device Context.
2nd & 3rd parameters are co-ordinates of the destination point.

- When you want to get current pen position, use **GetCurrentPenPositionEx ( )**.

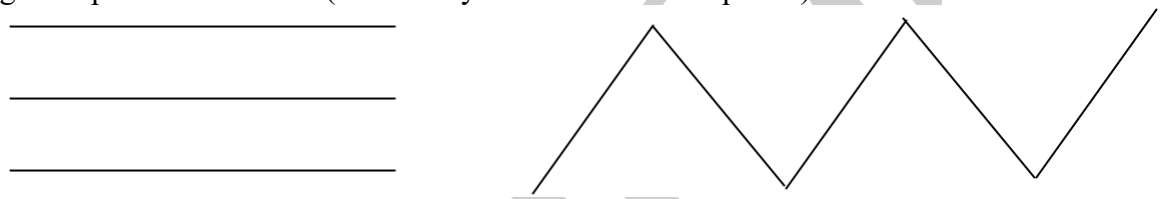
Prototype is :

BOOL GetCurrentPenPositionEx (HDC, LPPOINT) ;

Where 1st parameter is the handle to the current Device Context.
2nd parameter is the co-ordinates of the current position.

- **Polyline :**

Drawing multiple lines at a time (which may be connected or separate)



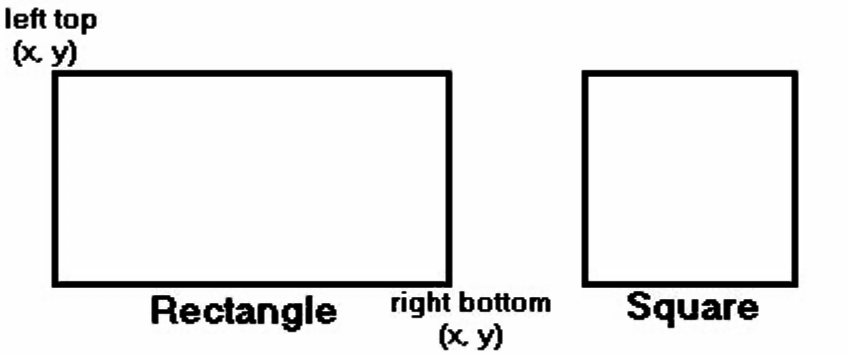
The function is used for this purpose is **Polyline ( ) :**

BOOL Polyline (HDC, LPPOINT, int) ;

1st parameter is handle to the current Device Context.
2nd parameter is address of **POINT** type array, in which you specify the co-ordinates of starting point and ending point of each line.
3rd parameter is number of elements in that array (or number of how many lines you want to draw).

- **Rectangle :**

This is the most basic shapes of many other shapes (ellipse, circle, rounded rectangle, arc, pie, chord) because many of these shapes use rectangle as their bounding shape. The function used for this purpose is **Rectangle ( )** which can draw both, linear rectangle & linear square.



Prototype is :

BOOL Rectangle (HDC, int, int, int, int) ;

1st parameter is handle to the current Device Context.
2nd parameter is X co-ordinate of left top or simply 'left'.
3rd parameter is Y co-ordinate of left top or simply 'top'.
4th parameter is X co-ordinate of right bottom or simply 'right'.
5th parameter is Y co-ordinate of right bottom or simply 'bottom'.

The same function can be used to draw square if you maintain equal distance between above two X co-ordinates & above two Y co-ordinates.

¾ Rounded Rectangle (Rectangle with rounded corners) :

The O.S. draws the shape by using four edges of rectangle + four ellipses at corner. Obviously, the function has all the parameters for **Rectangle ( )** + two new parameters. The function **RoundRect ( )** is used.

Prototype is :

BOOL RoundRect (HDC, int, int, int, int, int, int) ;

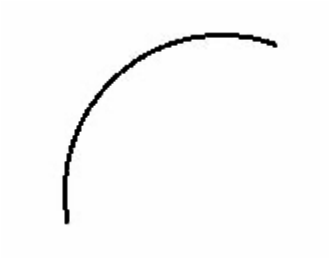
The 6th & 7th parameters are width & height of the ellipse, used to draw 4 ellipses (having same dimensions) at corners.

Usually the formula used for width is **(Right – Left) / 4**, for height is **(Bottom – Top) / 4**. But for large sized rectangles, these formulae don't give satisfactory results. Hence it is better to draw this shape with trial & error basis.

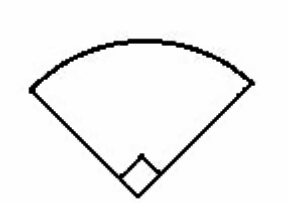
¾ Arc, Chord, Pie :

These three are rounded shapes and all have same number of parameters. The difference is only in their name.

Arc (…..) ;



Pie (…..) ;



Chord (…..) ;

These all use 9 parameters :

1st parameter is handle to Device Context.
2nd, 3rd, 4th & 5th are of bounding rectangle.
6th parameter is X co-ordinate of starting point of shape (**xStart**).
7th parameter is Y co-ordinate of starting point of shape (**yStart**).
8th parameter is X co-ordinate of ending point of shape (**xEnd**).
9th parameter is Y co-ordinate of ending point of shape (**yEnd**).

Actually Windows draws a bounding rectangle inside that bounding ellipse (both are not visible), then it draws an imaginary line from centre of ellipse to **xStart** & **yStart** and the point where the imaginary line intersects the bounding rectangle, Windows starts drawing the shape from this point. Windows also draws an imaginary line from centre of the ellipse to **xEnd**, **yEnd** & the point where the line intersects the bounding rectangle is used as stopping point of the shape.

These 3 shapes are drawn anticlockwise (Windows NT can draw clockwise).

¾ **Ellipse :**

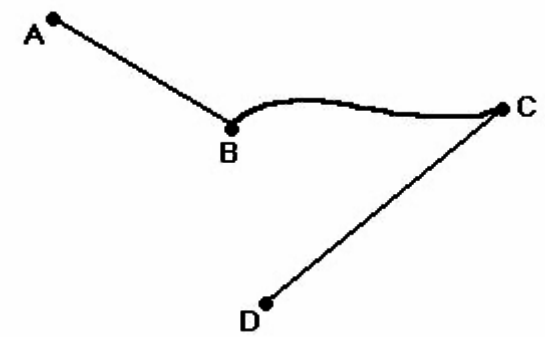
This shape has same parameters that of **Rectangle ( )** and their meaning is also same. This shows very curve shape uses bounding rectangle.

Prototype is :

BOOL Ellipse (HDC, int, int, int, int) ;

If the bounding rectangle is present, ellipse will be drawn & in case of square you get a circle.

¾ Spline Bezier :



A – Starting point.
D – Ending point.
B & **C** are 1st and 2nd control points.

In past Bezier were drawn using **Polyline ( )** function by giving 4 points in an array (refer **Polyline ( )** prototype). Out of which, 1st is considered as starting point & 4th is ending point. 2nd & 3rd are called as 1st & 2nd control point respectively, which are then used in 'Parametric Equations' to draw curved line between 2nd & 3rd point (see **Petzold** for equation). But now onwards (Win9x onwards), we don't have to worry about parametric equation. O.S. takes care of it. You must use once of the two functions.

Prototype is :

BOOL PolyBezier (HDC, LPPOINT, int) ;
BOOL PolyBezierTo (HDC, LPPOINT, int) ;

The meanings are same as that of Polyline ( ) function. These functions take first 4 points from array (i.e. Bezier at least requires 4 points) where 1st point becomes starting point, 2nd point becomes first control point, 3rd point becomes second control point & 4th point becomes end point. Using these 4 points, it draws 1st Bezier. Then it uses end point of 1st Bezier as starting point of 2nd. Hence Bezier are always connected. This also shows that only 1st Bezier requires 4 points. The consequent will require 3 only. Hence the last parameter is usually **(3 * number of desired number of curves + 1)**.

Note : In Bezier lines, connection of 2 lines or curves will be smooth only & only if the 2nd control point & end points of 1st Bezier & 1st control point of 2nd Bezier lie on single imaginary line.

¾ Polygon ( ) :

Parameters and their meaning is same as that of **Polyline ( )**. Just note that in **Polygon ( )** function, last integer is used as number of vertices of single shape & the points in **POINT** array get connected.

¾ **PolyPolygon ( ) :**

BOOL PolyPolygon (HDC, POINT *, int *, int) ;

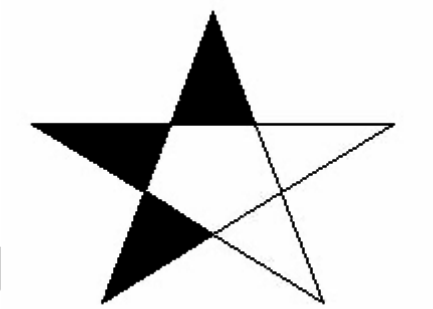
1st parameter is handle to the Device Context.
2nd parameter is co-ordinate of vertices specified in **POINT** type array.
3rd parameter is pointer to an integer array, which will specify the number of vertices of each shape.
4th parameter is number of shapes you want.

The process of filling the shapes has been covered in topic of "**Brush**".

1. Create the new brush either from stock object or custom.
2. Select it in current Device Context.
3. If the brush is stock, don't do anything. But if it custom, delete it.

Especially, for polygons, there are 2 filling modes, one is **ALTERNATE** and 2nd is **WINDING**.

‰ **ALTERNATE** : The polygonal shape is filled. Alternatively with odd counting (1, 3, 5, …) and it will fill only that part of polygon which is accessible from outside.



In the figure, the interior polygon is not accessible from outside, hence remain unfilled.

‰ **WINDING** : In this mode, usually all the polygonal areas are filled, means in above figure the interior polygon will also get filled.

Function used to set polygon filling mode is **SetPolyFillMode ( )**, **Prototype is :**
int SetPolyFillMode (HDC, int) ;

1st parameter is handle to the Device Context.
2nd parameter is one of the above value i.e. either **ALTERNATE** or **WINDING**.

Return value is previous polyfill mode. This function is used just one step before the polygon.

¾ **Some rectangle related function :-** (not commonly used)

i) **int FillRect (HDC, RECT *, HBRUSH) ; :** This function fills the give rectangle.

ii) **int FrameRect (HDC, RECT *, HBRUSH) ; :** Draws a frame (border) around the rectangle with HBRUSH colour.

iii) **BOOL InvertRect (HDC, RECT*) ; :** Inverts the colour of background & colour of rectangle logical `NOT' operation is used here.

iv) **BOOL SetRect (LPRECT, int, int, int, int) ; :** If you draw a rectangle with rectangle function & you want to draw 2nd rectangle with same co-ordinate instead of piece meal copying you pass the 4 co-ordinates into address of an empty RECT type structure variable.

v) **BOOL OffSetRect (LPRECT, int, int) ; :** This function moves given rectangle horizontally with distance specified in 2nd parameter & vertically with the distance specified in 3rd parameter.

vi) **BOOL InflateRect (LPRECT, int, int) ; :** This function can dynamically increase or decrease size of rect by specified value for width (2nd parameter) for height (3rd parameter).

vii) **BOOL SetRectEmpty (LPRECT) ; :** The values of structure **RECT** get set to zero (it gets empty) & there remains no area for this rectangle.

viii) **BOOL CopyRect (LPRECT, LPRECT) ; :**

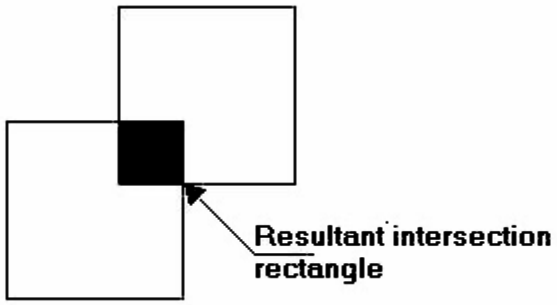
Destination Source

Copies source rectangle to destination rectangle by co-ordinate to co-ordinate copying.

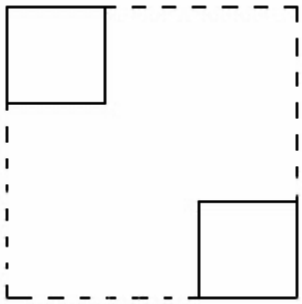
ix) **BOOL IntersectRect (LPRECT, LPRECT, LPRECT) ; :**

Destination Source Source

The function calculates the smallest intersecting rectangle of 2 rectangles.



x) **BOOL UnionRect (LPRECT, LPRECT, LPRECT) ; :** Calculates smallest rectangle capable of encompasing the 2 rectangles.



xi) **BOOL IsRectEmpty (LPRECT) ; :** Determines whether the supplied rectangle is empty or not.

xii) **BOOL PtInRect (LPRECT, POINT) ; :** Checks whether specified point is within all the borders or on the left and top (the points on right or bottom of rectangle are cosidered outside the rectangle. This is due to 'Upto' logic of O.S. explained earlier).

¾ **Metafile**

It is collection of binary data of **GDI** calls.

f To get the handle to the metafile, **CreateEnhMetaFile ( )** Win32 *API* is used.

Prototype is :

HEMFMETAFILE CreateEnhMetaFile (HDC, LPCTSTR, RECT *, LPCTSTR) ;

1st parameter is handle to the current device context created by **GetDC ( )** or **BeginPaint ( )**
2nd parameter is name of the Enhanced Metafile with its path. If you are going to display metafile directly which is created in memory & not on disk, then this parameter is NULL.
3rd parameter is dimension rectangle in which you are going to play the metafile.
4th parameter: Title to your created picture.

This function returns the handle to the device context of metafile. Once you created handle to metafile in memory, now you can use is just like **hdc** to draw anything on it. Just keep it in mind that when you call **GDI** function, instead of **hdc** use this metafile handle.

- After creating the metafile & after drawing on if you can play your newly created metafile by **PlayEnhMetaFile ( )**.

Prototype is :

BOOL PlayEnhMetaFile (HDC, HEMFMETAFILE, RECT *) ;

1st parameter is the handle to the device context (not the handle of metafile) created by **GetDC( )** or **BeginPaint ( )**. *Note that this handle must be the same that you have used as 1st parameter for **CreateEnhMetaFile ( )**.*
2nd parameter handle of metafile on which you have drawn various painting
3rd parameter address of the **RECT** structure variable which is going to be used as bounding rectangle in which you want to play metafile. It is not necessary that this rectangle must be the same as that of specified in **CreateEnhMetaFile ( )**.
It may be the same or different, means if these 2 are different, note that image will neither get shrink not will get expanded.,

- After completing use of metafile you should first close the handle of metafile, by using **CloseEnhMetaFile ( )**.

Prototype is :

HEMFMETAFILE CloseEnhMetaFile (HDC) ;

This function takes handle **hdc** as its parameter & returns handle of just closed metafile which further you can use in **DeleteEnhMetaFile ( )** to delete it from memory. Obviously, close function closes playing metafile, but doesn't delete it from memory. To remove metafile completely from memory you should use **DeleteEnhMetaFile ( )**.

Prototype is :

BOOL DeleteEnhMetaFile (HEMFMETAFILE) ;

The parameter is usually, the returned handle by **CloseEnhMetaFile ( )**. But if you wish you can use directly **DeleteEnhMetaFile ( )**. In such case this parameter will be handle returned by **CreateEnhMetaFile ( )**.

General Programming Scheme:

```
case WM_PAINT :
hdc = BeginPaint ( …. ) ;
hEnh = CreateEnhMetaFile ( …. ) ;
....
....
// Do necessaruy painting by using hEnh just liek hdc in GDI function
.....
….
PlayEnhMetaFile ( …. ) ;
hEnh = CloseEnhMetaFile ( .... ) ;
DeleteEnhMetaFile ( hEnh ) ;
```

Concerned with metafile, still there are 2 functions. When you create metafile **hdc** keeps the detailed properties of your newly created metafile in **ENHMETAHEADER** structure. You can get details of your metafile by function **GetEnhMetaFileHeader ( )**.

Prototype is :

UINT GetEnhMetaFileHeader (HEMFMETAFILE, UINT, LPENHMETAHEADER) ;

1st parameter is handle of your metafile about which you want details.
2nd parameter is **sizeof (ENHMETAHEADER)**.
3rd address is of **ENHMETAHEADER** structure variable. While sending, it is empty and on return it gets filled by system. The return value is actually filled **size of ENHMETAHEADER** structure (See MSDN for structure details).
If you have metafile saved on disk (then the 2nd parameter of **CreateEnhMetaFile ( )** won't be NULL. But it will contain desired path & desired metafile name) then to get its handle don't use **CreateEnhMetaFile ( )**, instead you use **GetEnhMetaFile ( )**.

Prototype is :

HEMFMETAFILE GetEnhMetaFIle (LPCTSTR) ;

There are 3 topics remaining which required for Advanced Study of Graphics. :
1) Drawing Modes. : **GetROP2 ( )** , **SetROP2 ( )**
ROP2 stands for Raster Operation 2
    i) Region
    ii) Path

4th Chapter

Dialog Box & Child Window Control

Dialog Box

Dialog Box is major **UI** (User Interface) used in almost all **GUI** applications. Conceptually, Dialog Box is nothing but a window usually a software program's Main Window creates Dialog Box to give facility of input to the user. Obviously, the Dialog Box becomes Child Window of the program's Main Window.

Child Windows are of two types :

1. Pre-defined i.e. Given by O.S.
2. Custom-created.

You can use Child Windows as "Controls" either on your Main Window or on the Dialog Box. In programmers using Child Window Controls on Dialog Box is more common than using Child Window Controls on Main Window. In this sense a Child Window Control has Dialog Box as its parent window and Main Window as its grand-parent window.

Consider O.S,. itself as a window (say Desktop Window), then your program's window is O.S.'s Child Window. Your window & O.S.'s window interact with each other by getting and sending message to each other. We used this logic in all programs up till now (in **WndProc**, where all **WM_** are received messages and all used *API* functions like **PostQuitMessage ( )** is sent message.).

Similarly, if Dialog Box is parent window of Child Window Control (like Radio Button, Push Button), then Dialog Box & Child Window Control also can talk with each other. This is called as "***Child is talking to its parent***" and "***Parent is talking to its child***".

¾ Creating & Using Dialog Box.

Before studying this topic, first of all note that **SDK** given us two ways of creating many **UI** items including Dialog Box. :-

1. Creating in our ".C" source file using *API*.
2. Creating as **resource**.

The first way though give full freedom, there is a major drawback that many items many remain in memory, even if not needed, e.g. Suppose, there are three menu items and also suppose that, clicking on second menu item displays a Dialog Box. User opened your program and finishes it by using only first and third menu items. He never chooses the second menu item. In such cases, though the second menu item is not used, Dialog Box code remains in the memory, because you wrote it in your ".C" source file.

So to avoid, this unnecessary 'memory-eating' task, there should be such a way in which such items, say Dialog Box code will get loaded in memory when & only when needed, means in our above example, Dialog Box code will be loaded in memory only when second menu item will be clicked. This can be done by second way, in which we will create such a item by "resource".

¾ Resource File & Resource Compiler

Here & in all future programs, we are going to write Dialog Boxes and all Child Window Controls in a resource file.

1. Creating the resource file : You can create resource file in any editor by giving your desired filename with '.rc' extension. In Visual C++ IDE, you can create it by :

File⟶ New ⟶Files ⟶Text Files⟶ Specify name & extension ⟶ Save.

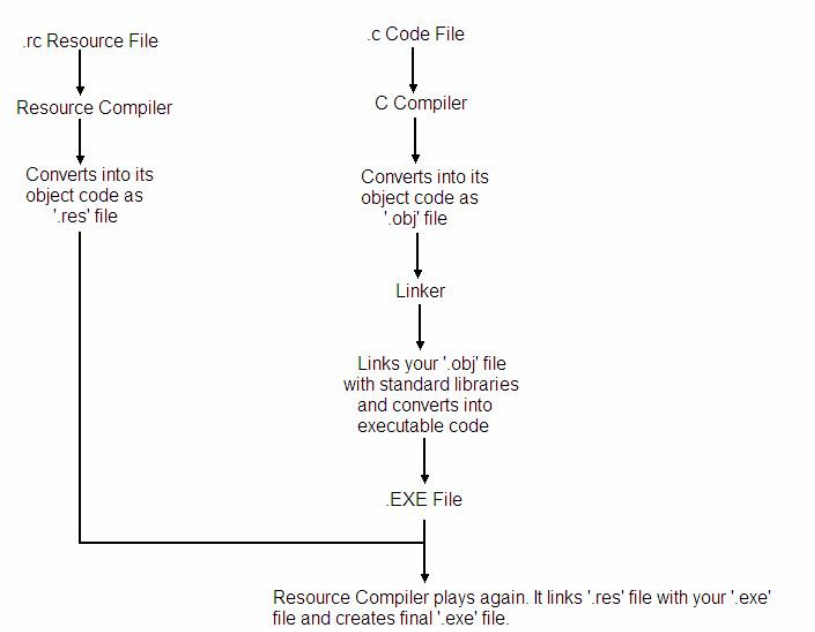
This will open an empty window with your '.rc' file name as its title. Now you can write your resource specific code in this file.

2. Adding newly created resource file into the project : If you do above task while your project is open, then your newly created resource file will get automatically added into your project. But if your write your resource file in different editor & now you want to add it in your project, then :

First of all copy this resource file to the project's directory ⟶ open your project's workspace ⟶ go to project menu ⟶ select add files⟶ then click on your resource file name⟶ finally click on add.

3. Resource Compiler : When you compile your '.c' source code, if you look at the output window, you will found that, only your '.c' file gets compiled and not the resource file. But if you click on Build or Re-Build, then output window shows the messages of resource compilation. This indicates that resource file has different compiler than the usual **C** compiler. This compiler for the resource file called as 'Resource Compiler'.

¾ How Resource Compiler works ?



Above flow chat shows that, Resource Compiler works twice.

¾ Dialog Box Template

Now its clear that, we are going to use resource file to create Dialog Box. The syntax of resource file script is different than the usual **C** syntax. The code for Dialog Box called as Dialog Box Template.

I. DIALOG Resource Statement :

Syntax :

<desired dialog's> **DIALOG** <load option> <memory option> **X, Y, Width, Height** resource name

Explanation :

- **Load option :** There are two load options –

1. **ALWAYS :** means, Dialog Box will always remain in memory, not used commonly.
2. **LOADONCALL :** means, Dialog Box will be loaded in memory whenever required, otherwise not. This is by default and most commonly used, need not to be mentioned.

- **Memory option :** There are three memory options.

1. **FIXED :** The code will remain at fixed memory location. O.S. will not do any adjustments.
2. **MOVEABLE :** O.S. will move memory block of code, whenever necessary. This is by default.
3. **DISCARDABLE :** Whenever not required, O.S. will remove from memory block of the memory.

Note : *Dialog Box Units : While specifying X, Y, Width & Height, you must know that position and extent of Dialog Box does not depend upon pixel unit. Instead, it uses a special type of unit known as Dialog based unit in which X & Width is expressed in terms of 1/4th of average character width of current system font. While Y & Height is expressed in terms of 1/8th of average character height of current system font.*

These units are not applicable to X, Y, Width & Height of Dialog Box, but also of all Child Window Controls, that you want to place on the Dialog Box.

II. STYLE Statement :

The common and the default window styles for Dialog Box are **WS_POPUP | WS_VISIBLE | WS_CAPTION**. Besides this, there are two common styles in practice, which are **BS_MODALFRAME** which creates Modal Dialog Box with a window frame.

DS_CENTER : which keeps the Dialog Box in the center of the client area independent of the resolution and the font.

DS_SYSMODAL : To create System Modal Dialog Box.

III. CAPTION Statement :

Syntax :

CAPTION "<desired heading of the Dialog Box which will appear in its title bar>"

IV. FONT Statement :

By default system uses current system font for drawing Dialog Box & all its Child Window Controls. If you wish, means if matter is much more than the desired size of Dialog Box, you can change the font for Dialog Box by using FONT statement.

Syntax :

FONT <size>, "<Font's name>"

e.g. FONT 8, "MS SANS SERIF"

V. BEGIN Statement :

This statement indicates that all following statement will be the Child Window Control statement, that we want to put on our Dialog Box.

…….
…….
…….
Child Window Control Resource Statement
…….
…….
…….

VI. END Statement :

This statement indicates that all Child Window Control statements are finished and thus dialog statements are also finished.

NOTE :- As Dialog Box is a window. How Windows O.S. uses resource script to create the Dialog Box ?

: *Windows O.S. simply calls* ***CreateWindow ( )*** *uses "dialog" as class name (1st parameter), a string at caption is used as 2nd parameter, your given styles in STYLE statement are used as 3rd parameter with addition of* ***WS_CHILD****, your X, Y, Width, & Height as 4th, 5th, 6th, 7th parameters, handle of the Main Window as parent window handle as 8th parameter (taken from* ***DialogBox ( )*** *API). Then it uses your dialog resource name as 9th parameter casted to* ***HMENU*** *type. Then it uses handle at the instance (taken from* ***DialogBox ( )*** *) and keeps last parameter as* ***NULL****.*

Above method of Windows O.S. is not only applicable to **DialogBox ( )**, but also to all window controls.

In other words, though you use simpler resource statement to create Dialog and Child Window Controls, O.S. internally uses the same old **CreateWindow ( )** for this purpose.

VII. LANGUAGE Statement :

As like FONT statement, you can specify language option too. For the supported languages see **MSDN**. If this statement is not used, then default language is **US_ENGLISH**.

¾ Types of Dialog Boxes :

There are two types of Dialog Boxes :-

1) Modal
2) Modeless (non-modal)

1) Modal Dialog Box :-

This type of Dialog Box does not allow user to enter next part of your program, means you have to close Dialog Box first before doing something else with the program. But you can run another program from **Start** Menu or from somewhere else. This type of Dialog Box is called **Appmodal**.

These is yet another sub-type of Modal Dialog Box known as **Sysmodal,** which even does not allow user, to start any program, neither from **Start** Menu, nor from anywhere else, means user have to close this Dialog Box before doing anything with Dialog Box .

Modal Dialog Boxes are more common in programming practice.

2) Modeless (non-modal) Dialog Boxes:

Unlike Modal Dialog Box, this type of Dialog Box can allow user to do anything with the program or with another program. But for this sake Dialog Box should be minimized.

Many Dialog Boxes on Internet Applets are of this type. Thus Modeless Dialog Boxes are not strict as Modal Dialog Boxes.

Child Window Controls

As stated before, when you want Child Windows on Main Window (which is unusual), then creating Child Window is done by using **CreateWindow ( )** or **CreateWindowEx ( )** directly.

But when we want to create Child Windows on Dialog Box, then there is yet another way by specifying desired Child Window's resource statement. By using resource statement, you can create Child Window on Dialog Box by two ways :-

a) By using Child Window's specific resource statement. e.g. Push Button.

PUSHBUTTON "OK", IDOK, 140, 115, 25, 10

Above statement will create a Push Button. Here **PUSHBUTTON** is the Push Button specific resource statement.

b) By using general CONTROL statement :

Syntax :

CONTROL "<Text>", ID, "<class>", STYLES, X, Y, Width, Height

Note that : *In first way, styles are taken by default, but in second way, you have to give class name & styles explicitly. Every Child Window must have three styles :* ***WS_CHILD****,* ***WS_VISIBLE*** *& control specific styles. e.g. In above example, the control specific style is BS_PUSHBUTTON.*

Now suppose, if you want to do same task by **CreateWindow ( )**, then call would be :

```
CreateWindow ( "button", "OK", WS_CHILD | WS_VISIBLE |
                               WS_TABSTOP | BS_PUSHBUTTON,
                               140 * cxChar / 4, 115 * cyChar / 8,
                               25 * cxChar / 4, 10 * cyChar / 8, hDlg,
                               (HMENU) IDOK, hInst, NULL ) ;
```

No difference between the calls. The class name "button" goes as first parameter, then "OK" text on Push Button goes as 2nd parameter.

All styles go as 3rd parameter.

Then X and Width are converted into dialog based unit by multiplying their required size with the current system font's average character width and divided by 4.

While Y and Height are converted into dialog based unit by multiplying their required size with the current system font's average character height and divided by 8.

So care of 4, 5, 6, 7 parameter has been taken now the 8th parameter is parent window handle. If you want to put Child Window on Main Window, it will be **HWND** and if you want to put Child Window on Dialog Box, it will be **HDLG**.

9th parameter is actually for handle of the menu, but as Child Window Control is not going to have menu, we can use its place for specifying our Child Window Control's ID. But ID is an integer and this parameter is **HMENU** type. *Hence ID must be casted to* ***HMENU*** *type*.

The 10th parameter is the instance handle. We already has seen the two ways of getting instance handle in our Second Chapter (Text) in **WM_CREATE**, there is yet

another third way of getting instance locally. This is done by using **GetWindowLong ( )** *API*. The call will be like follows :

hInst = (HINSTANCE) GetWindowLong (hwnd, GWL_INSTANCE) ;

(Note : GWL stands for **G**et **W**indow **L**ong)

By using this same function, you can get all the eleven parameters value of **CreateWindow ( )** by using **GetWindowLong ( )**.

The 11th parameter is as usual NULL, but if you wish, you can send user-defined window creation parameter.

Above discussion is just for knowledge. Most programmers use first way of Child Window Control createion i.e. by using resource specific statement.

In resource specific statement, first of all we should know that how many Child Window, we can add to the Dialog Box.

Child Window Control	Resource Specific Statement	Class
1. Push Button	PUSHBUTTON	button
2. Default Push Button	DEFPUSHBUTTON	button
3. Radio Button	RADIOBUTTON	button
4. Automatic Radio Button	AUTORADIOBUTTON	button
5. Group Box	GROUPBOX	button
6. Check Box	CHECKBOX	button
7. Centered Static Text	CTEXT	static
8. Left Aligned Static Text	LTEXT	static
9. Right Aligned Static Text	RTEXT	static
10. Edit Text Box	EDITTEXT	edit
11. List Box	LISTBOX	listbox
12. Combo Box	COMBOBOX	combobox
13. Scroll Bar	SCROLLBAR	scrollbar

The resource statements for above Child Window Controls have two forms :-

1) For "Push Button, Default Push Button, Radio Button, Auto Radio Button, Group Box, Check Box, Static Text (CTEXT, RTEXT, LTEXT), the Syntax is :

<Resource Statement> "<Text>", ID, <X>, <Y>, <Width>, <Height>, <STYLES>
e.g. PUSHBUTTON

2) But for List Box, Edit Text Box, Combo Box, Scroll Bar, the Syntax is :

<Resource Statement> ID, <X>, <Y>, <Width>, <Height>, <STYLES>

Dialog Box DIAGRAM ……

The resource script for above Dialog Box will be something like follows (say **MyDialog.rc**)

```
#include <windows.h>
#include "mydialog.h"

DATAENTRY DIALOG DISCARDABLE 0, 0, 300, 150
STYLE DS_MODALFRAME | DS_CENTER | WS_CAPTION
CAPTION "DATAENTRY"

BEGIN
        LTEXT "Name         :", -1, 35, 10, 30, 10
        EDITTEXT ID_ETNAME, 70, 10, 150, 10, WS_TABSTOP
        LTEXT "Address      :", -1, 27, 30, 50, 10
        EDITTEXT ID_ETADDRESS, 70, 30, 150, 10, WS_TABSTOP
        LTEXT "Age          :", -1, 41, 50, 20, 10
        EDITTEXT ID_ETAGE, 70, 50, 11, 10, WS_TABSTOP
        LTEXT "Salary       :", -1, 32, 70, 30, 10



        LTEXT "Marital Status       :", -1, 5, 90, 60, 10
         AUTORADIOBUTTON "Married", ID_RBMARRIED, 70, 90, 50, 10,
                                                 WS_TABSTOP | WS_GROUP
        AUTORADIOBUTTON "Unmarried", ID_RBUNMARRIED, 120, 90, 65, 10
        DEFPUSHBUTTON "Continue", ID_PBCONTINUE, 33, 115, 35, 10,
                                                        WS_TABSTOP
        PUSHBUTTON "Ok", IDOK, 140, 115, 25, 10
        PUSHBUTTON "Cancel", IDCANCEL, 230, 115, 30, 10
END
```

Usually LTEXT, RTEXT, CTEXT like Child Windows are given IDs as -1, when we don't want any messages to be received from them, nor we want to send any messages to them. But if your intension is to change static text during displaying of Dialog Box, you can give them IDs.

The next step is to create header file, say **MyDialog.h** and include this file in both '.c' file & '.rc' file by **#include "….."** directive. The header file for above Dialog Box will be some think like follows :

```
#define ID_ETNAME               101
#define ID_ETADDRESS            102
#define ID_ETAGE                103
#define ID_ETSALRS              104
#define ID_ETSALPS              105
#define ID_RBMARRIED            106
#define ID_RBUNMARRIED          107
#define ID_PBCONTINUE           108   <ENTER>
```

(Don't forget to press ENTER)

This means that all programmer defined IDs must be given unique integer values in this header file.

¾ How to display the Dialog Box ?

We created the Dialog Box template in '.rc' file, but it is not going to display Dialog Box on window. Displaying the Modal Dialog Box requires **DialogBox ( )** *API*, whose **Prototype** is :

int DialogBox (HINSTANCE, LPCTSTR, HWND, DLGPROC) ;

1st parameter is instance handle of the application.

2nd parameter is the Dialog Box template name, which you have written in '.rc' file before the word "DIALOG".

3rd parameter is the handle of the parent window.

4th parameter is the name of the Dialog Box's CALLBACK Procedure.

Note that : *As Window Procedure's (**WndProc**) prototype, Dialog Box's procedure (DlgProc) must be declared globally, because it is of CALLBACK type.*

• Where to call this function ?

1) If you want to display the Dialog Box, before your Main Window, then you can do it at two places : a) in **WinMain ( )**, after the call to **CreateWindow ( )**, but before the call to **ShowWindow ( )**. b) in **WM_CREATE**.
2) In **WndProc ( )**, under your desired message handler case-construct.

The **DialogBox ( )** displays the Dialog Box on screen, gives control to Window O.S., but this does not return until Dialog Box destroyed by **EndDialog ( )**. This is usually done in the Dialog Box's CALLBACK Procedure.

• Writing DialogBox ( ) CALLBACK Procedure ?

The general prototype of **DialogBox ( )** is :

BOOL CALLBACK <desired name, e.g. DlgProc> (HWND, UINT, WPARAM, LPARAM) ;

As a convention, the function name is post-fixed by DlgProc. e.g. The Dialog Box Procedure for above Dialog Box may have name **MyDlgProc ( )**.

Note that : *This is the name that you must use as the fourth parameter of **DialogBox ( )**. The prototype of **DialogBox ( )**, must be declared globally as it is **CALLBACK** type.*

If you see the parameters and the naming convention of this **CALLBACK** function, it look very much similar to **WndProc ( )**. This is because Dialog Box itself is a window and as a rule every window must have its own Window Procedure.

You may surprise to see that **WndProc ( )** is having return type **LRESULT**, but this **DlgProc ( )** is having return value **BOOL**.

In **WndProc ( )**, the return value was **LRESULT** because we use **DefWindowProc ( )** function as return value of **WndProc ( )**. **DefWindowProc ( )**'s return value is of **LRESULT** type. Hence **WndProc ( )** has return value as **LRESULT**.

This obviously shows that our **DlgProc ( )** is not going to have **DefWindowProc ( )** because our **DlgProc ( )** is not true Dialog Box Procedure. The true Dialog Box Procedure (having **DefWindowProc ( )**) is hidden in O.S. Our **DlgProc ( )** is just representative of that true Dialog Box Procedure.

The message flow for our Main Window and **WndProc ( )** is …….

User ⟶ O.S.⟶ **WndProc ( )**⟶ **WinMain ( )** ⟶**WndProc ( )**⟶ O.S.

The message flow for **DlgProc ( )** is ……

User ⟶ Dialog Box ⟶ O.S. ⟶ True Dialog Box Procedure hidden in Windows ⟶ Application's **DlgProc ( )** ⟶ Dialog Box ⟶ O.S.

The body of Dialog Box is very much similar to the body of **WndProc ( )** . It is also going to have switch loop and multiple case statements of received messages from True Dialog Box Procedure.

Whenever user interacts with Dialog Box (enters a name in Edit Box, checks a Radio Button or presses a Push Button), the message is sent from the Dialog Box to the True Dialog Box Procedure, which is hidden in O.S. Then this Dialog Box Procedure sends the same or alters appropriately and then sends it to the application's **DlgProc ( )**, where we handle them by different case statement with message handlers.

Obviously, some messages we will not handle (in **WndProc ( )**, there was **DefWindowProc ( )** to handle this situation) . For such messages we will just return FALSE and for processed (handle) messages, we will return TRUE. That is why this function has return type **BOOL**.

The **DialogBox ( )** Procedure can handle many messages like **WndProc ( )**, but two are more important :

1) WM_INITDIALOG 2) WM_COMMAND

1) WM_INITDIALOG :

This is just like **WM_CREATE** of Main Window's **WndProc ( )**. This message is received by **DlgProc ( )** before Dialog Box is displayed. Hence the case-construct of this message is used for initialization of Dialog Box. e.g. Setting focus in specific Edit Box.

Already checking a Radio Button or Check Box. In data-entry programs this message can be used to show the user his previous record where he was stopped.

2) WM_COMMAND :

This is heart of **DlgProc ( )**. It receives all the messages sent by Child Window Controls on the Dialog Box, means

i) if a string is entered in Edit Text, Edit Box will notify the **DlgProc ( )** by **WM_COMMAND**
ii) When Radio Button checked, Check Box will notify the **DlgProc ( )** by **WM_COMMAND**.
iii) When Push Button is clicked, Push Button will notify the **DlgProc ( )** by **WM_COMMAND**.

Note That :
Then the question arises "What things Child Window Controls give to ***WM_COMMAND****, and then it gives it back to us ?*
: These Child Window Controls give three things for ***WM_COMMAND*** *:*

1. *Their ID.*
2. *Their notification code (i.e. what exactly user done with them)*
3. *Their window handle.*

Now ***WM_COMMAND*** *gives these three things to us as* ***LOWORD (wParam)****,* ***HIWORD (lParam)*** *and* ***lParam*** *respectively.*

In general, we can say that "receive whatever Dialog Box is saying in **WM_COMMAND** and do whatever you want to do with Dialog Box in **WM_INITDIALOG**.

¾ The code of our **DlgProc ( )** will be something like follows :

```
BOOL CALLBACK MyDlgProc ( HWND, UINT, WPARAM, LPARAM ) ;
// Globally declared
structure struct INPUT
{
        char name [50], address [50] ;
        int age, mstatus ;
        float sal ;
} ;

int WINAPI WinMain ( …… )
{
        …
        …
        …
        // as it is
        …
        …
        …
}

LRESULT CALLBACK WndProc ( HWND hwnd, UINT iMsg,
                                        WPARAM wParam,
                                        LPARAM lParam )
{
        HINSTANCE hInst ;
        switch (iMsg)
        {
                case WM_CREATE :
                hInst = ( HINSTANCE ) GetWindowLong ( hwnd, GWL_INSTANCE ) ;
                // call to Dialog Box
```

```
                DialogBox ( hInst, "Data Entry", hwnd, MyDlgProc ) ;
                ....
                ....
                ....
                // any dialog related
                work ....
                ....
                ....
                break ;
                .... ....
                ....
                // other case structures
                ....
                ....
        }

return ( DefWindowProc ( hwnd, iMsg, wParam, lParam ) ) ;
}

// Dialog Box Procedure

BOOL CALLBACK MyDlgProc ( HWND hDlg, UINT iMsg, WPARAM wParam,
                                        LPARAM lParam )
{
        char salrs [6], salps [3] ;
        switch ( iMsg )
        {
                case WM_INITDIALOG :
                // set focus in name Edit Box
                SetFocus ( GetDlgItem ( hDlg, ID_ETNAME ) )
                ; // keep married Radio Button checked
                SendDlgItemMessage ( hDlg, ID_RBMARRIED, BM_SETCHECK,1,0 ) ;
                return ( TRUE ) ;

                case WM_COMMAND :
                switch ( LOWORD ( wParam ) )
                {
                        case ID_PBCONTINUE :
                        ....
                        ....
                        ....
                        // whatever task you
                        want ....
                        ....
                        EndDialog ( hDlg, 0 )
                        ; break ;
```

```
                    /* The 2nd parameter of EndDialog ( ) is used as return value
                    of DialogBox ( ) */

                    case IDOK :
                    // Get the user-entered name
                    GetDlgItemText ( hDlg, ID_ETNAME, in.name, 50 ) ;
                    // Get the user-entered address
                    GetDlgItemText ( hDlg, ID_ETADDRESS, in.address, 50 ) ;
                    // Get the user-entered age
                    GetDlgItemInt ( hDlg, ID_ETAGE, in.age, NULL, TRUE )
                    ; // Get the user-entered salary
                    GetDlgItemText ( hDlg, ID_ETSALRS, salrs, 6 ) ;
                    GetDlgItemText ( hDlg, ID_ ETSALPS, salps, 3 )
                    ; in.sal = atoi ( salrs ) + ( float ) atoi ( salps / 100 ) ;
                    // Get user-entered marital status
                    in.mstatus = SendDlgItemMessage ( hDlg, ID_RBMARRIED,
                                                        BM_GETCHECK, 0, 0 ) ;
                    EndDialog ( hDlg, 0 ) ;
                    break ;

                    case IDCANCEL :
                    EndDialog ( hDlg, 0 ) ;
                    break ;
                }
                return ( TRUE ) ;
        }
        return ( FALSE ) ;
} // End of Dialog Procedure
```

¾ Tasks those we can do in WM_INITDIALOG

There are many advantages of **WM_INITDIALOG**. Some important are :

1) Setting focus to the desired Child Window Control on the Dialog Box.
   Steps :
   I. First get the handle of the Child Window Control by using **GetDlgItem ( )**
      Prototype is :
      HWND GetDlgItem (HWND, int) ;
      1st parameter is handle of the Dialog Box on which the control is present. 2nd parameter is ID of Child Window Control.
      On return this function gives handle of the Child Window Control that you queried for.
      e.g. **hwndedit = GetDlgItem (hDlg, ID_ETNAME) ;**
   II. Now set the focus on desired Child Window Control. The function is used for this purpose is **SetFocus ( )**, whose **prototype** is :
      HWND SetFocus (HWND) ;

The only parameter is handle of the window on which you want to set the focus.
The return value is the handle of the window which loses focus.
e.g. **SetFocus (hwndedit) ;**
In our code, to avoid extra variable declaration of **hwndedit**, we combined these function as :
SetFocus (GetDlgItem (hDlg, ID_ETNAME)) ;

(*Note :* *The return value of the **SetFocus ( ) gives you the handle of the window which loses the focus, but if you want the handle of the window which presently has focus, then you may use GetFocus ( ), the **prototype** is :***
HWND GetFocus (void) ;
This function returns the required window handle having the focus.
*The same task can be done by using **SendMessage ( )** with its 2nd parameter as **WM_SETFOCUS** or **WM_KILLFOCUS**. For **wParam**, **lParam** of these two messages, see **MSDN**.*
*Not only by **SendMessage ( )**, but even using 'case' of these two messages, you can do the same task. **)***

2) Changing the text 'of the control' or 'in the control'
: This can be done by using two ways :
   I. First get the handle of Child Window whose text you want to change by **GetDlgItem ( )**, as shown above and then by using **SetWindowText ( )**, you can set the text. This function is general purpose function, means it can change the text associated with Child Window Control or it can change the window title of any window. The **prototype** is :
      int **SetWindowText (HWND, LPCTSTR) ;**
   II. You can directly use **SetDlgItemText ( )** to do the same. The **prototype** :
      BOOL SetDlgItemText (HWND, int, LPCTSTR, int) ;
      2nd parameter is the ID of the Child Window whose text you want to change.
      3rd parameter is actual text.
      4th parameter is string length.

3) Checking of desired Radio Button before display of Dialog Box. This can be done by using 2 methods :
   I. First get the Child Window handle by using **GetDlgItem ( )** and then use **SendMessage ( )**.
      e.g. **SendMessage (GetDlgItem (hDlg, ID_RBMARRIED), BM_SETCHECK, 1, 0) ;**
   II. By directly using **SendDlgItemMessage ( )**, whose **prototype** is : **int SendDlgItemMessage (HWND, int, UINT, wParam, lParam) ;**
      This function is just similar to **SendMessage ( )**, except 2nd parameter which here stands for the Child Window Control's ID.
      e.g.
      SendDlgItemMessage (hDlg, ID_RBMARRIED, BM_SETCHECK, 1,0);

4) In the cases like Data-Entry, if Dialog Boxes is getting displayed continuously by pushing 'Continue' button, then we can use **WM_INITDIALOG** to allow the user to see all the contents of previous records. In such cases, we will get the data

after pressing 'Ok' button, means in case **IDOK** of **WM_COMMAND** of **DlgProc ( )**, then we will store these received values somewhere globally and then we will use them in **WM_INITDIALOG** with the functions **SetDlgItemText ( )**, **SetDlgItemInt ( )**, **SendDlgItemMessage ( )** to display previously entered records.

¾ Tasks which can be done in WM_COMMAND

1) By checking **LOWORD (wParam)** with various defined IDs, you can take appropriate actions on the user's interaction with the Child Window Control. e.g.

i) In our program, the case of 'Continue' button will have the statements of redisplaying of Dialog Box with empty and default fields as explained in **WM_INITDIALOG**.

ii) In case of **IDOK**, we will write statements to receive user input information like user input string, integer, selection of Radio Button, etc.

iii) In case of **IDCANCEL**, we write the statements of exiting the Dialog Box without receiving any information.

2) Dynamic enabling or disabling of the Child Window Control. e.g. If Dialog Box has two radio buttons for "Father's Name" and "Husband's Name", then for male sex "Husband's Name" Radio Button should be disabled.

Above tasks can be done as :

1. How to get the user-entered string ?

: For this purpose, you should have a string variable with appropriate size of maximize of the Edit Box. Then use **GetDlgItemText ( )** at appropriate location. e.g. In our example, it should be in case **IDOK**. The call will be like follows :

GetDlgItemText (hDlg, ID_ETNAME, in.name, 255) ;

Where **in.name** is the string variable, when passed it is empty, but when this function will return, it will get filled with user-entered in name edit field.
The last parameter is size of the string.

Note that : *String of your specified size will return String of your specified size will return, means if you specify maximum size, the whole string will return. But if you specify last parameter as 6 and user-entered string of about 50 characters, then only 5 of them will be put in the string variable and* 6^{th} *will be kept for NULL character.*

2. How to get the user-entered integer ?

: To get the user-entered integer from the edit field, you should use **GetDlgItemInt ( )** like :

in.age = GetDlgItemInt (hDlg, ID_ETAGE, NULL, TRUE) ;

Obviously, to receive the value returned by this function, you should have same integer, like here we have **in.age**.

By same long way, you may also do same above things by using **GetDlgItemText ( ),** in which integer will be received in int & then by using **C**

library function like **atoi ( )** or **strtoi ( )**, you can convert integer string to integer number.

3. How to receive user-entered floating values ?

: There is no separate function like **GetDlgItemInt ( )** to receive floating point values. Hence you have to do three tasks :

1. Receive the user-entered floating point number as string by **GetDlgItemText ( )**.
2. a) Then check this string for presence of multiple decimal points
   b) If multiple show Message Box of error
   c) If not convert the string to floating point number by **atof ( )** or **strtof ( )** library function. These **C** functions are declared in **stdlib.h**. Obviously, you should have floating point variable to receive the floating point number returned by **atof ( )** or **strtof ( )**.

4. Dynamic Enabling & Disabling of Child Window.

: This is very simple job, can be done in **WM_COMMAND**, just by using single function **EnableWindow ( )**. **EnableWindow ( )** takes only two parameters, handle of the window which you want to enable or disable. 2nd parameter is **TRUE** for enable, **FALSE** for disable.
So get the Child Window's handle by **GetDlgItem ( )**, if you are in **WM_INITDIALOG** and by **lParam** if you are in **WM_COMMAND**.

After doing desired tasks in **WM_COMMAND** switch give the closing break. ***Note that :*** *The case statement in **WM_COMMAND** switch can break, because **WM_COMMAND** finally returns TRUE.*

After completing switch, now write return statement for **WM_COMMAND**.

About return statement of Dialog Procedure, as explained before. The actual Dialog Box for our Dialog Box is hidden in Windows O.S. Hence this Dialog Procedure which we had written now is representative of that actual Dialog Box Procedure.

We have also that, the messages that we handle should returned TRUE. Hence **WM_INITDIALOG** & **WM_COMMAND** return TRUE. Obviously, we do not write any message handlers for any other message, except this two. Hence all other non-processed messages should return FALSE. That is why, our Dialog Box's last return statement is FALSE, which represent all those non-processed messages.

Modeless Dialog Box (Non-Modal Dialog Box)

Modeless Dialog Box requires a different method of creation and some extra work, because Modeless Dialog Box gets its messages from your program's message queue.

¾ Important differences between Modal & Modeless Dialog Boxes & in their handling are :

1) Modeless Dialog Boxes usually have System Menu and no Caption Bar. Thus System Menu is added by adding **WS_SYSMENU** style in resource script. Though Modal Dialog Boxes can have System Menu, its of no use, because no effect occurs by clicking on it.
2) For Modeless Dialog Box, you must specify **WS_VISIBLE** style in its template or you must use **ShowWindow ( )** with **SW_SHOW** or **SW_SHOWNORMAL** as its 2nd parameter. Without one of these two, Modeless Dialog Box will not be displayed. On contrarily, Modal Dialog Box does not require explicit mentioning of this style, neither requires **ShowWindow ( )**, because **WS_VISIBLE** is default style of Modal Dialog Box.
3) The messages for Modal Dialog Box come directly through the main **DlgProc ( )** hidden in Windows O.S. Hence they do not interfere with the application's message loop. But messages of Modeless Dialog Box come through the application's message loop. Hence we must modify the message loop.
4) Modeless Dialog Box does not terminate with **EndDialog ( )**. Instead, it requires use of **DestroyWindow ( )** with handle of Modeless Dialog Box as its parameter.
5) To create the Modal Dialog Box, we already used **DialogBox ( )**. While to create the Modeless Dialog Box, we need **CreateDialog ( )**, which takes same parameter with same meanings as that of Dialog Box. But the difference is the Dialog Box function was returning while **CreateDialog ( )** returns handle to the modeless Dialog Box, which should be caught in the globally declared **HWND** type variable, because we usually use **CreateDialog ( )** in **WndProc ( )** and we use its returned handle in **WinMain ( )** too.

*Note : Besides creating Modeless Dialog Box by **CreateDialog ( )**, there are five other functions which can be used to create the same Modeless Dialog Box. These functions are :*

1. **DialogBoxParam ( ) - (** Macro version of **CreateDialog ( )**.)
2. **CreateDialogIndirect ( )**
3. **DialogBoxIndirect ( ) –**
   (Macro version of **CreateDialogIndirect ( )**)
4. **CreateDialogIndirectParam ( )**
5. **DialogBoxIndirectParam ( ) –**
   (Macro version of **CreateDialogIndirectParam ( )**)

For details of these functions, see **MSDN**.

¾ Creating & Using Modeless Dialog Box ?

1. Changes in resource script : Add **WS_VISIBLE** style, if not already there. Also you can add **WS_SYSMENU**, if you want to add System Menu.
2. Creating the Modeless Dialog Box : At the place where you used **DialogBox ( )**, now use **CreateDialog ( )**. First declare global **HWND** variable, like **HWND hwndmodeless ;** (*Note : Some use to initialize this handle by NULL as a good habit.)*
   Now at the place of **DialogBox ( )**, give following call
   : **hwndmodeless = CreateDialog (hInst,**
   "DialogBoxTemplateName",
   hwnd,
   Name of DialogBox Procedure i.e. DlgProc) ;

   We are going to use this handle at three places :

i) In **WinMain ( )**'s message loop.
ii) If **WS_VISIBLE** style is not mentioned in resource script, we will call **ShowWindow ( )** with this handle after above call.
iii) In **DialogBox ( )** Procedure's **WM_CLOSE** message handler.

3. Changes in **DialogBox ( )** Procedure : You can keep **DialogBox ( )** Procedure as it is in Modal Dialog Box. For Modeless Dialog Box, just add new message handler after **WM_INITDIALOG** & **WM_COMMAND** like follows :
   Code :

```
case WM_CLOSE :
DestroyWindow ( hDlg ) ;
hwndmodeless = 0 ;
break ;
```

Note that : *In **DestroyWindow ( )**, we used the local Dialog Box handle i.e. **hDlg** and on next statement, we made the global Dialog Box handle NULL. This method ensures that both the handles are destroyed. The reason why we used **WM_CLOSE** is that, the user may close Dialog Box by selecting 'Close' from the System Menu of the Dialog Box. In the cases of **IDOK**, **IDCANCEL**, etc., wherever we used **EndDialog ( )**, now replace it by : **i) DestroyWindow (hDlg) &***
ii) hwndmodeless = 0 ;

4. Changes in the message loop : The message loop in **WinMain ( )** should be modified as follows :

```
while ( GetMessage ( & msg, NULL, 0, 0 ) )
{
        if ( hwndmodeless = = NULL | |
        ( IsDialogMessage ( hwndmodeless, & msg ) = = 0 ) )
        {
                TranslateMessage ( & msg ) ;
                DispatchMessage ( & msg ) ;
        }
}
```

The **TranslateMessage ( )** & **DispatchMessage ( )** are part of the loop is for our main **WndProc ( )**. The message loop should go to **WndProc ( )**, only when **DialogBox ( )** is zero or there is no message for Dialog Box. If any of these condition fails, means if either handle of the Dialog Box is valid (non-zero) or if there is some message to Dialog Box, then the **TranslateMessage ( )** & **DispatchMessage ( )** will not execute. But now control will go first to O.S., then to the main **DlgProc ( )** hidden in O.S., then to **DlgProc ( )** of your application.

Message Box

¾ MessageBox ( )

This function displays the Dialog Box style Message Box, which is of Modal type and user must respond to it before proceeding further. The **prototype** of this function is :

int MessageBox (HWND, LPCTSTR, LPCTSTR, UINT) ;

Note : See Page No. 30 of 1st Chapter (Introduction & History) in topic of Message Box. There we had written two groups of 4th parameter. Actually there are 5 groups :

1. Number of Push Buttons. (Already seen on Page 30 & 31 in 1st Chapter.)
2. About icon of message box (Already seen on Page 30 & 31 in 1st Chapter.)
3. Which type of Push Button, you want to make default :
   a) MB_DEFBUTTON1 :- This is by default.
   b) MB_DEFBUTTON2 :- 2nd Push Button is made by default.
   c) MB_DEFBUTTON3 :- 3rd Push Button is made by default.
   d) MB_DEFBUTTON4 :- 4th Push Button is made by default.
4. Modality of the Message Box. There are 3 options :

i) MB_APPMODAL : You must response the Message Box before proceeding to your program.

ii) MB_SYSMODAL : You must response the Message Box before doing anything with the system.

iii) MB_TASKMODAL : This style will require, if you specify NULL for the 1st parameter and you want effect of

APPMODAL 5. Some extra styles :

i) MB_DEFAULT_DESKTOP_ONLY : Desktop is made by default owner of the Message Box.

ii) MB_HELP : Help button is also displayed with other Push Button, which respond to mouse-click, to key-press and also to key **F1**.

iii) MB_RIGHT : The text in Message Box gets right aligned (By default is left aligned).

iv) MB_RTLREADING : For right to left reading as in Hebrew or Urdu.

v) MB_SETFOREGROUND : The window becomes the fore ground window.

vi) MB_TOPMOST : The window becomes top most window.

vii) MB_SERVICE_NOTIFICATION : For NT only.
viii) MB_SERVICE_NOTIFICATION_NT3X : For NT only.
ix) MB_NOFOCUS : By giving this style, the Message Box has no focus.

If you text is longer than the width of the window, then system breaks the text into multi -line inside the Message Box. But if you wish you can do it on your own, by using '**\n**' in the string of 2nd parameter of **MessageBox ( )**.

¾ Colouring the Dialog Box :

The default system colour of Dialog Box (which is mostly grey) can be changed by using **WM_CTLCOLORDLG** message handler in your **DialogBox ( )** Procedure :

1. First declare a static variable of **HBRUSH** type. Say **hDlgBrush**.
2. Now write a case of **WM_CTLCOLORDLG** as follows :

```
case WM_CTLCOLORDLG :
hDlgBrush = CreateSolidBrush ( RGB ( 255, 0, 0 ) ) ;
return ( ( BOOL ) hDlgBrush ) ;
```

The colouring of Dialog Box is done by system's currently selected brush. By processing **WM_CTLCOLORDLG**, you tell the system, you don't want system's default brush, but you will specify yours. Thus you create brush, in this case by using **CreateSolidBrush ( )** and you return the handle of the newly created brush to the O.S. So that O.S. can use it to point your Dialog Box. As our Dialog Box Procedure **BOOL** type value, you must typecast this handle to **BOOL** type.

3. Don't forget to delete this handle before using **EndDialog ( )** in any further case statement like **DeleteObject (hDlgBrush) ;**

Edit Control

This is a rectangular window which allows user to write information in string form.

Note that : *Edit Control has built-in capacity of an Edit Menu commands like Undo, Cut, Copy, Paste, Delete, Select All.*

Actually, Edit Control is a small window, usually larger in length but one character in height. So multi-line text in an Edit Control is not a common practice. But if you wish you can create Edit Control as large as your client area, even you can apply horizontal & vertical scrolling with multi-line input. The resource statement of Edit Control is something like follows :

EDITTEXT ID_ETNAME, X, Y, Width, Height, Styles

The default styles for Edit Box are : **WS_TABSTOP**, **WS_CHILD**, **WS_VISIBLE**, **WS_BORDER**.

¾ Common Edit Styles (all are prefixed by ES_)

1) ES_AUTOHSCROLL : This enables auto horizontal scrolling, means although limit of Edit Window ends, horizontal scrolling goes on towards right.
2) ES_AUTOVSCROLL : This enables auto vertical scrolling, means although limit of Edit Window ends, vertical scrolling goes on towards bottom.
3) ES_CENTER : This enables caret to appear in the center of the Edit Window.
4) ES_LEFT : This enables caret to appear in the left of the Edit Window.
5) ES_LOWERCASE : Input text will be entered in lower case.
6) ES_MULTILINE : To allow multi-line Edit Control.
7) ES_NOHIDECELL :
8) ES_OEMCONVERT :
9) ES_PASSWORD : Password like input (asterisk).
10) ES_RIGHT : This put caret in the right of the Edit Window, so text appears from right.
11) ES_UPPERCASE : Input text will be entered in upper case.
12) ES_READONLY : User cannot type here, but Edit Box appears with some pre-determined text.
13) ES_WANTRETURN : To enable to move tab to next box after pressing Enter.

¾ Edit Control Notifications (prefixed by EN_)

1) EN_CHANGE :
2) EN_ERRSPACE : this notification message will be sent from the control to parent window's procedure when the limit of Edit Control Window ends.
3) EN_HSCROLL :
4) EN_KILLFOCUS : This notification message sent when the Edit Control loses focus.
5) EN_MAXTEXT :
6) EN_SETFOCUS :
7) EN_UPDATE :
8) EN_VSCROLL :

¾ Edit Control Messages (prefixed by EM_)

1) EM_CANUNDO :
2) EM_CHARFROMPOS :
3) EM_EMPTYUNDOBUFFER :
4) EM_FMTLINE :
5) EM_GETFIRSTVISIBLELINE :
6) EM_GETHANDLE :
7) EM_GETLINE : Allows programmer to take input string of concern line.
8) EM_GETLINECOUNT : Allows programmer to retrieve number of line from multi-line Edit Control.
9) EM_GETTHUMB : When control has scroll bar.

10) EM_GETWORDBREAK :
11) EM_LIMITTEXT :
12) EM_LINEFROMCHAR :
13) EM_LINEINDEX :
14) EM_LINELENGTH :
15) EM_LINESCROLL :
16) EM_POSFROMCHAR :
17) EM_REPLACESEL :
18) EM_SCROLL :
19) EM_SCROLLCARET :
20) EM_SETHANDLE :
21) EM_SETLIMITTEXT :
22) EM_SETMARGINS :
23) EM_SETMODIFY :
24) EM_SETPASSWORDCHAR :
25) EM_SETREADONLY :
26) EM_SETRECT :
27) EM_SETRECTNP :
28) EM_SETSEL :
29) EM_SETWORDBREAKPROC :
30) EM_UNDO :

If you have multiple Edit Controls in your Dialog Box or parent window and the first control (out of all other controls) is the Edit Control, then it gets focused by default.

List Box

This is a rectangular box (window) which contain list of strings. The user is allowed to select any of the string by clicking on it. If the number of strings are much more than given vertical limit of the box, then vertical scroll bar is automatically displayed at right side of the List Box. But you can not type in the List Box. List Box is pre-defined control and its resource statement is : **LISTBOX**

The resource statement is something like follows :

LISTBOX ID_LB X, Y, Width, Height, Styles

Diagram of List Box.

The default styles are **WS_CHILD**, **WS_VISIBLE**, **WS_TABSTOP**, **LBS_NOTIFY**, **LBS_BORDER**.

¾ Other List Box Styles (prefixed by LBS_)

1) LBS_EXTENDEDSEL :
2) LBS_HASSTRINGS :
3) LBS_MULTICOLUMN :
4) LBS_MUTIPLESEL : Allow user to select more than one string from the List Box.
5) LBS_NOREDRAW :
6) LBS_OWNERDRAWFIXED :
7) LBS_OWNERDRAWVARIABLE :
8) LBS_USETABSTOP :
9) LBS_WANTKEYBOARDINPUT :
10) LBS_DISABLENOSCROLL :

¾ List Box Notification Codes (prefixed by LBN_)

1) LBN_DBLCLK :
2) LBN_KILLFOCUS :
3) LBN_SELCANCEL :
4) LBN_SETCHANGE :
5) LBN_SETFOCUS :

¾ Out of above LBS_ styles, the most common are three :

1) LBS_NOTIFY : Allow programmer to send message from the List Box to parent window procedure. The sent message is usually about the selected or de-selected item.
2) LBS_SORT : Sort the strings in alphabetical order. So that when a key is pressed, the List Box gets scrolled up to the desired alphabet.
3) LBS_STANDARD : Instead of giving two above properties separately, you can give this only, because it combines above two.

Note : *Surprisingly, there are no List Box messages with* ***LBM_*** *prefix, because these messages are sent explicitly by the programmers using* ***SendMessage ( )*** *or* ***SendDlgItemMessage ( )*** *with* ***LB_*** *prefixes. e.g. LB_ADDSTRING : strings will be added to List Box. LB_SELECTSTRING.*

Combo Box

In a structure, it is quiet similar to with List Box. The additional facility is Combo Box is edit field. In List Box, you can only select the choice among the give choices. But in Combo Box, you can enter or add the string as per your desire. If it is already present,

then it will be selected and if it is not already present, then you can write such a code which will receive the user input string and the next time, when the Combo Box will be displayed, it will be added to the Combo Box.

Another advantage of Combo Box is that, List Box is displayed of whatever size you specify. But if crowding of controls on parent window occurs, you can not reduce size of List Box of one line length and width. But in Combo Box, you can reduce the size, so that it will display its List Box only when you click the down arrow on Combo Box.

As its name suggest, it is 'Combo', means combination of List Box & Edit Box.

Diagram ….when displayed ----
Diagram….when clicked……

The resource statement for Combo Box is **COMBOBOX**.

The call will be something like follows :

COMBOBOX ID_CB, X, Y, Width, Height, Styles

The default styles are CBS_SIMPLE | **WS_TABSTOP** and all other explained in List Box (except LBS_NOTIFY, LBS_BORDER)

By default vertical scrolling of Combo Box is not available (For List Box, it is available by default). To allow vertical scroll bar for Combo Box, you should explicitly add WS_VSCROLL style to above styles.

¾ Important Combo Box Styles (prefixed by CBS_)

1) CBS_SIMPLE : Same as List Box displayed with full size, but with Edit Box.
2) CBS_DROPDOWN : Same as CBS_SIMPLE, but displayed in single line form and can get its full size, when clicked on the down arrow placed at the right side of the Combo Box.
3) CBS_DROPDOWNLIST : It is same as CBS_DROPDOWN, except instead of Edit Box, there is static control, means you can not enter the text, but as like List Box, you can select one of the strings. This style is usually used, when you actually want List Box, but your parent window does not have that much place to hold List Box. Hence you need shrunken List Box, which is Combo Box with CBS_DROPDOWNLIST.
4) CBS_SORT : Sorts the strings alphabetically & scrolls automatically when you enter the text in edit field of Combo Box.

¾ Other Styles of Combo Box.

5) CBS_AUTOHSCROLL :
6) CBS_HASSTRINGS :
7) CBS_OEMCONVERT :
8) CBS_OWNERDRAWFIXED :
9) CBS_OWNERDRAWVARIABLE :

10) CBS_DISABLENOSCROLL :
11) CBS_NOINTEGRALHEIGHT :
(All styles are same as **LBS_**, just replaced by **CBS_**)

¾ The Notification Codes of Combo Box (prefixed by CBN_).

1) CBN_CLOSEUP :
2) CBN_DBLCLK :
3) CBN_DROPDOWN :
4) CBN_EDITCHANGE :
5) CBN_EDITUPDATE :
6) CBN_ERRSPACE :
7) CBN_KILLFOCUS :
8) CBN_SELCHANGE :
9) CBN_SELENDCANCEL :
10) CBN_SELENDOK :
11) CBN_SETFOCUS :

Scroll Bar (as Control)

Usually, this control is used for some sliding action or for scrolling of small edit windows. The resource statement is **SCROLLBAR**, which can be used something like follows :

SCROLLBAR ID_SB, X, Y, Width, Height, Styles

The default style is SBS_HORZ, if not specified explicitly.

¾ Important Scroll Bar Styles (prefixed by SBS_)

1) SBS_HORZ : Designs horizontal Scroll Bar Control.
2) SBS_VERT : Designs vertical Scroll Bar Control.

¾ Other Scroll Bar Styles

3) SBS_BOTTOMALIGN : Scroll Bar Control gets aligned with the bottom border of parent window.
4) SBS_LEFTALIGN : Scroll Bar Control gets aligned with the left border of parent window.
5) SBS_RIGHTALIGN : Scroll Bar Control gets aligned with the right border of parent window.
6) SBS_TOPALIGN : Scroll Bar Control get aligned with the top border of parent window.
7) SBS_SIZEBOX :

8) SBS_NOBORDERS :
9) SBS_OWNERDRAW :
10) SBS_POPOUT :
11) SBS_SIZEBOXTOPLEFTALIGN :
12) SBS_SIZEBOXBOTTOMRIGHTALIGN :

¾ Scroll Bar Messages (prefixed by SBM_).

1) SBM_ENABLE_ARROWS :
2) SBM_GETPOS :
3) SBM_GETRANGE :
4) SBM_GETSCROLLINFO :
5) SBM_SETPOS :
6) SBM_SETRANGE :
7) SBM_SETSCROLLINFO :
8) SBM_SETRANGEREDRAW :

Icon (as Control)

¾ How to display Icon in Dialog Box / parent window ?

: Icon cane be put on the window, but it is static type of control; usually we do not specify any ID to it. Instead, we use it, just like other static controls. No message is receive or sent from or to Icon Control. creating Icon Control requires two Icon statements :

1. Icon resource statement.
2. Icon control statement.

If you want to put Icon on Dialog Box, then Icon resource statement must be outside and before the Dialog Box template and Icon Control statement should be inside the Dialog Box template.

Syntax of Icon resource statement :

<your desired resource name, e.g. DATAENTRY> ICON <"File Name">

e.g. **DLGICON ICON "MyIcon.ico"**

The '.ico' file must be in your working directory of project where '.c', '.rc' & '.h' files are located. If not then you have to give complete path of your '.ico' file.

Now inside the Dialog Box template, write Icon Control statement, whose syntax is : **ICON <"resource name">, ID (e.g. -1) X, Y, Width, Height, Styles.**

The default style is **SS_ICON**.

X, Y specify the location on Dialog Box, where you actually want to put the Icon. Width & Height are of no interest. Because Icon in '.ico' file has its own width & height. Just specify 0, 0 for that, if you just satisfied with current width & height of that Icon. But if the width and height of '.ico' file is larger & overlapping some of your textual

information, then by specifying your own Width & Height, you can display it. Hence it always better to recreate Icon by smaller dimensions than to shrink or expand it in Dialog Box.

¾ Some Important Points Concerned With Child Window Control :

We had already discussed much things about Child Window Control in topic of Dialog Box summary of them is :

1) In Dialog Box we displayed :
   a) Static Text.
   b) Edit Box.
   c) Radio Button.
   d) Push Button.

   As Child Windows having their only parent as Dialog Box.
2) We used resource statement method to create Child Windows called as Resource Script and we create all these resource statements of Child Window Controls inside Dialog Box template.
3) We discussed two ways of creating Child Window Controls by resource method.
   a) Resource specific statement.
   b) A generalized **CONTROL** statement.

We also said that, though creating Child Window Control by resource method is easier, it can also be created by most basic window creation function **CreateWindow ( )** or **CreateWindowEx ( )**.

As an example, we created a Push Button by both these methods on pushbutton (on page 112).

5) Child Window Controls are commonly placed on Dialog Box, placing of them on window is not a usual practice.
6) After creating Child Window Controls, we use to receive messages in **WM_COMMAND** message handler of our **DlgProc ( )** and we used to send messages to Child Window Controls in **WM_INITDIALOG**. But if the case is of Dynamic Enabling or Disabling, then we send message through **WM_COMMAND**>

Now here, we are mainly concerned with the creation of those four Child Window Controls on Main Window (which are placed on Dialog Box) and remaining Child Window Controls on both.

When we use **CreateWindow ( )** to create Child Window Control, then basic difference between Child Window Control and Main Window are :

	Main Window	Child Window
1)	We need **WNDCLASSEX** structure type variable and assign to its 12 members.	We don't need it.
2)	We have to resister our **WNDCLASSEX** structure by **RegisterClassEx ( )**	We don't need it.

3)	We need **CreateWindow ( )** call with first parameter as any our desired string	Call to **CreateWindow ( )** for Child Window requires first parameter as string of predefined class name. e.g. "button".
4)	We need **WndProc ( )** to be written by us to handle interaction with Main Window.	Child **WndProc ( )** need to be written our-self, it is already written by O.S. But if we wish, we can write our own Child **WndProc ( )** for predefined Child Window Control by using method by Window Sub-classing.

¾ Why the three things i.e. WNDCLASSEX structure, RegisterClassEx ( ) and WndProc ( ) are not necessary for Child Window Control ?

: The answer is in one word, because they are **predefined**.
Then a new question arises "Why our Main Window needs them ?".
The answer is that our Main Window has structure and behavior of our desire, means our Main Window is not predefined, but it is "**Programmer Defined**". Hence its registration to O.S. is must.

¾ Where to use the CreateWindow ( ) for creating Child Window Controls ?

: The place is usually in **WndProc ( )**, either in a message handler or any **WinMain ( )**, when you want Child Windows from beginning.

¾ How to process Parent – Child message communication ?

: Some like in Dialog Box, here our **WndProc ( )** will need **WM_COMMAND** message handler for processing Parent-Child communication. The difference in **WM_COMMAND** message handler of Dialog Box and of **WndProc ( )** is that, in **WndProc ( )** we will not call **EndDialog ( )** and we will mainly use break statement instead of using return statement.

¾ How to obtain the Child Window handle in WndProc ( ) ?

: There are 4 ways. You can use any of them :

1) The Child Window handle which you got as return value of **CreateWindow ( )**, when creating Child Window, should be declared globally. Hence any function can use it.
2) The Child Window handle which you got as return value of **CreateWindow ( )** function, when creating Child Window, should be declared locally **static**.

3) If you need Child Window handle in **WM_COMMAND**, then there is no need of above 2 ways, because **lParam** of **WM_COMMAND** is itself the Child Window handle. Thus remember that, while using this, you have to typecast it to **HWND** type.
4) Though above 3 ways are there, most programmers use this way, because it is more simple and convenient.
We can se **GetDlgItem ( )** to get window handle. Though, it looks odd that we are using "Dlg" type function for Main Window, but it works well. Thus, pass the Main Window handle in place of Dialog Box handle as first parameter. Remaining process of **GetDlgItem ( )** is same as explained in Dialog Box.
But remember that, this does not mean that other "Dlg" functions may or should work similar.

Working with Child Windows

I. How to get text from Edit Box, when Edit Box is on Main Window ?

:

a) First get the Child Window's handle.
b) Then use **SendMessage ( )** with BM_GETTEXTMESSAGE, where its **wParam** is size of the string and **lParam** is the empty string buffer in which the string is to be retrieved.

II. How to get user-entered int or real values from the Edit Box, when the Edit Box is on Main Window ?

:

a) Use above method to get number as string.
b) Then by using **atoi ( )** or **atof ( )**, convert them into **int** or **float**, respectively.
Similar **C** library function can be used for other type of numbers.

III. How to get the checked state of checked Radio Button/Check Box, when the Radio Button/Check Box is on Main Window ?

:

a) Get Child Window handle.
b) Send message with **BM_GETCHECK**, whose both **wParam**, **lParam** are zero.
c) Assign the return value of **SendMessage ( )** to an **int** type variable to get the checked state. If the return value is 1 or **BST_CHECK**, then the button is checked, otherwise return value is zero, means unchecked.

Note that : SendMessage ( ) *returns* ***LRESULT*** *type value. Hence before receiving it into* ***int*** *type variable, you have to use type-casting to* ***int*** *type.*

IV. How to get the pushed state of Push Button, when Push Button is on Main Window ?

:

a) Actually this is not necessary, because when user pushes the button, button immediately sends **WM_COMMAND** message and if not, there is not pushed. But still if you want to get this state without **WM_COMMAND**, then follow the procedure of point no. **III**, as it is. Just instead of **BM_GETCHECK**, here you use **BM_GETSTATE**.

V. How to do Dynamic Enabling or Disabling of Child Window Control, when Child Window Control is on Main Window ?

:

a) The procedure is same as explained in Dialog Box. First we should get Child Window handle, then use **EnableWindow ( )** with second parameter **TRUE** for Enabling and **FALSE** for Disabling.

VI. How to get the static text or the text on button or the text on right or left side of Child Window Control, when the all above texts are on Main Window ?

:

a) Procedure is same as explained in Dialog Box. We should use **GetWindowText ( )** by using the obtained Child Window handle as first parameter.

VII. How to do 'pre-displayed' initializations of Child Window Controls, when they are on Main Window ?

:

a) In Dialog Box we did this thing in **WM_INITDIALOG** message handler, so now on controls are on Main Window. Hence we will use **WM_CREATE** message handler.

VIII. How to set:

a) Text on Push Button
b) Text on right of left of Radio Button/Check Box
c) Static text, when they are on Main Window ?

:

a) first get the Child Window handle.
b) Use **SetWindow ( )**. For static texts, you should have ID.

IX. How to set text in Edit field, when Edit Box is on Main Window ?

:

a) Procedure is same as seen in getting text, just replace **WM_GETTEXT** by **WM_SETTEXT**, whose **wParam** is zero, while **lParam** is the string that you want to set.

X. How to set int or real value in Edit Box, when Edit Box is on Main Window ?

:

a) First convert the number into string by using **sprintf ( )** (or use **atoi ( )**) and then use same above method of setting text in edit field.

XI. How to set check on Radio Button/Check Box, when Radio Button/Check Box are on Main Window ?

:

a) Get the Child Window handle.
b) Use **SendMessage ( )** with **BM_SETCHECK** whose **wParam** should be **BST_CHECKED** or 1 and **lParam** should be zero.

XII. How to set a push on Push Button, when the Push Button is on Main Window ?

:

a) Actually this is not necessary. But if you wish, you can do it by using **BM_SETSTATE** message with **wParam** and **lParam** same as above of **BM_SETCHECK**.

XIII. How to make Child Window Control invisible ?

:

a) For this purpose, use **ShowWindow ( )** with Child Window handle as first parameter and **SW_HIDE**. When you again want to show it, use same function with **SW_SHOW** or **SW_SHOWNORMAL**.

Check Box

Note : Everything for Check Box is same as that of Radio Button for both, when the Check Box is on Dialog Box and when the Check Box is on Main Window.

List Box

The discussion following, now applies for both, List Box on Dialog Box and List Box on Main Window.

I. How to add strings to List Box ?

:

a) First of all, you should have valid array of pointers to string, means it should contain valid strings.
b) Now use '**for'** loop from zero to above array size **- 1**. Then inside the 'for' loop, either use **SendDlgItemMessage ( )**, if your List Box is on Dialog Box or use **SendMessage ( )**, when List Box is on Main Window. (It is assumed that before stating loop, you already had got the handle of the List Box on Main Window) and inside **SendDlgItemMessage ( )** or **SendMessage ( )**, use **LB_ADDSTRING** message with **wParam** zero and **lParam** string array with its index.
For Dialog Box code will be something like follows :-

```
case WM_INITDIALOG :
…..
…..
….
….
for ( i = 0 ; i < = ARRAYSIZE – 1 ; i ++ )
        {
        SendDlgItemMessage ( hDlg, ID_LB, LB_ADDSTRING,
                                (wParam) 0 , (lParam) (LPCTSTR) pstring [i] ) ;
        }
…..
…..
…..
…..
break ;
```

When the List Box is on Main Window, then code will be :

```
case WM_CREATE :
…..
….
….
….
hwndchild = GetDlgItem ( hwnd, ID_LB ) ;
for ( I = 0 ; I < = ARRAYSIZE – 1, I ++ ) ;
        {
        SendMssage ( hwndchild, ID_LB, LB_ADDSTRING,
                        (wParam) 0, (lParam) (LPCTSTR) pstring [i] ) ;
        }
…..
….
break ;
```

II. How to set the selection of 0th item (1st selection in List Box) in List Box. ?

: The message used for this purpose is **LB_SELECTSTRING**. Its usage is like follows :

For Dialog Box :

SendDlgItemMessage (hDlg, ID_LB, LB_SELECTSTRING, (wParam) 0, (lParam) (LPCTSTR) STR) ;

Here **STR** is the string array in which you want to set the selection. **wParam** value is the index. We want to set the selection at 0^{th} item, hence it is **(wParam) 0**.

The **LB_SELECSTRING** searches the List Box for your given index. If you give -1 as **wParam**, then entire List Box is searched. This facility can be used, when you want to search single string and not only search, you want to select string. In such cases use -1 as **wParam** and then the string which you want to select and search as **lParam**.

III. How to select the nth item of List Box ?

: The procedure is quiet similar as point. No. **II**. Such facilities are required in Data-base program where you may want to display a last-entered record, whenever database program starts. Obviously, you will need such a variable, say **iSelect**, which should be declared globally or locally as static, because you are going to use in both, in **WM_INITDIALOG** (**WM_CREATE**) and **WM_COMMAND**.

Now you will first get user's selection in **WM_COMMAND** in **iSelect** variable. (How to do this we will discuss next) and then use it in **WM_INITDIALOG** to select on this value. For this purpose you can use **LB_SELECTSTRING** as explained before, but LB_SELECSTRING is for both, searching & selecting. There is yet another message, which is only for selecting and i.e. **LB_SETCURSEL** whose **wParam** will be the **iSelect** value and **lParam** will be zero.

IV. How to get the user-selected item from the List Box ?

: For both List Box on Dialog Box and List Box on Main Window, this has to be done in **WM_COMMAND** message handler. Then message used is **LB_GETCURSEL** whose **wParam** will be zero and **lParam** will be zero and the call **SendDlgItemMessage ( )** or **SendMessage ( )** should be assigned to the index variable, say **iSelect**.

```
iSelect = SendDlgItemMessage ( hDlg, ID_LB, LB_GETCURSEL, (wParam) 0,
                                                    (lParam) 0 ) ;
iSelect = SendMessage ( hwnd, ID_LB, LB_ GETCURSEL, (wParam)
                                                    0, (lParam) 0 ) ;
```

Now if you want to program the task like IV, means getting **iSelect** in **WM_COMMAND** and according to value of **iSelect** you want to set the selection in **WM_INITDIALOG** (or **WM_CREATE**), then you should check **iSelect** variable, whether it contains any valid index or not. This can be checked by 'if-else' block with **LB_ERR** value.

V. How to do multiple selection in List Box ?

: Note that multiple selections is not supported by List Box by default, you have to add **LBS_MULTIPLE**, while creating the List Box, either in resource script or in **CreateWindow ( )**. The logic of selection is same as in II, III, IV except that instead of using **LB_GETCURSEL**, use **LB_GETSEL**. You can get multiple values in **WM_COMMAND**, by using **while (TRUE)** like loops.

Combo Box

Everything is same as List Box, just replace all **LB_** prefixes by **CB_**.

Note :- As Combo Box has Edit Control, it is obvious that it will not support multiple selection.

Scroll Bar as Control

Most of the logic of Scroll Bar Control is same as the logic of Main Window's Scroll Bar as described in the topic of 'Text', but there are four major differences :

1) You can specify **SB_HORZ** or **SB_VERT** for Main Window's Scroll Bar. But when you want Scroll Bar as control, then you have to use **SBS_CTL**, which also means that one control will be only either horizontal or vertical, never the resource script or **CreateWindow ( )** call should have **SBS_HORZ** for horizontal Scroll Bar or **SBS_VERT** for

vertical Scroll Bar. As like **SB_BOTH**, there is no such SBS_BOTH for Scroll Bar Control.

2) We expect that, Child Window Control should send **WM_COMMAND** message to its parent. Obviously, we will expect that Scroll Bar Control should also send **WM_COMMAND** message to its parent. But this is not the case unlike other Child Window Controls. Scroll Bar Control sends **WM_VSCROLL** or **WM_HSCROLL** to its parent, depending upon which type of Scroll Bar you used. (This is a similarity between Main Window Scroll Bar and Scroll Bar Control.)
3) Now an interesting situation may occur……Usually Scroll Bar controls placed on Dialog Box, but if you decide to place it on Main Window and if your Main Window also has its own Scroll Bar, then a problem will arise, either **WM_VSCROLL** or **WM_HSCROLL** handler had to decide whether incoming message is from Main Window's Scroll Bar or from Scroll Bar Control. Here **lParam** helps us. When the **WM_HSCROLL** or **WM_VSCROLL** is sent from Main Window's Scroll Bar, then the **lParam** is zero. But when one of them is sent from Scroll Bar Control, then the **lParam** is handle of the Scroll Bar Control's Window. So by using 'if-else' statements with the **lParam** value, you can proceed respective scrolling. But this type of situation is core, because putting control on Main Window is also rare, as we saw controls are mainly put on Dialog Box. Though **lParam** differs, **wParam** remains same for both, Main Window's Scroll Bar and Scroll Bar Control.
4) Main Windows Scroll Bar are usually kept right side of the window (Vertical Scroll Bar) or left side of the window (when you need **RTL** type reading). But Main Window's horizontal Scroll Bar is always of bottom. This restriction is not for Scroll Bar Control, you can put vertical Scroll Bar Control, either on left or right and you can put, either at bottom or at top. Style of this positioned Scroll Bar Control **SBS_RIGHTALIGN**, **SBS_TOPALIGN**, **SB_LEFTALIGN**, **SBS_BOTTOMALIGN**.

¾ How to colour the Child Window Control ?

: Windows O.S. have six messages for colouring Child Window Controls :

1) **WM_CTLCOLORSTATIC** : To change the colour of static text including static text concerned with Radio Button.
2) **WM_CTLCOLORBTN :** To change colour of Button Control i.e. Push Button, Radio Button, Check Box.
3) **WM_CTLCOLORSCROLLBAR** : To change colour of 8 Scroll Bar Controls.
4) **WM_COLOREDIT** : To change the colour of Edit Control, but the Edit Control must not be read-only or must not be disabled while creating.
5) **WM_CTLCOLORLISTBOX** : to change the colour of List Box, can be used for Combo Box.

6) **WM_CTLCOLORMSGBOX** : This was in Win 16, but it is now obsolete (removed).

The logic of use these above first 5 messages in program is same as explained in **WM_**COLORDLG message processing for colouring custom created Dialog Box.

We will take one example of colouring the static text :

```
static HBRUSH hStatBrush ;
....
....
....
....
case WM_CTLCOLORSTATIC :
hStatBrush = CreateSolidBrush ( RGB ( 255, 0, 0 ) ) ;
SetBkColor ( ( HDC ) wParam, RGB ( 255, 0, 0 ) ;
SetTextColor ( ( HDC ) wParam, RGB ( 0, 0, 255) ) ;
return ( BOOL ) hStatBrush ;
```

Above code is for colouring the static text on Dialog Box. When static text is on Main Window, just make a single change in return statement. Instead of using **BOOL** typecast, use **LRESULT** typecase. Because the previous code is used in **DlgProc ( )** and return value of **DlgProc ( )** is **BOOL**. Hence the **BOOL** typecasting is done. But as **WndProc ( )** is returns **LRESULT** type value, you have to typecast to **LRESULT** type.

All above 5 colouring messages have their **wParam** as handle to the device control of Child Window and **lParam** as handle of the Child Window.

Above code will point all static texts, but when you want different static texts controls to get painted by different colours, then sight modification in above code should be made. First you should have ID to the static text statements that you want to point. Suppose there are two static statements having IDs : ID_STAIC1, ID_STATIC2 and you want to point one static control by blue colour and other static control by white colour, then in above code first check the coming hwnd in **lParam** and hwnd given y **GetDlgItem ( )**, then use **SetTextColour ( )**. Obviously, background colour is going to remain same. Code will be :

```
....
....
staic HBRUSH hBrush ;
....
....
case WM_CTLCOLORSTATIC :
hBrush = CreateSolidBrush ( RGB ( 255, 0, 0 ) ) ;
// Set background colour to above brush
SetBkColor ( ( HDC) wParam, RGB ( 255, 0 , 0 ) ) ;
if ( ( hwnd ) lParam = = GetDlgItem ( hDlg, ID_STATIC1 ) )
SetTextColor ( ( HDC ) wParam, RGB ( 0, 0, 255 ) ) ;
// One line by Blue Colour
if ( ( hwnd) lParam = = GetDlgItem ( hDlg, ID_STATIC2) )
SetTextColor ( ( HDC ) wParam, RGB ( 255, 255, 255 ) ) ;
// Text in white colour
return ( BOOL ) hBrush ;
```

Above code is for static control on Dialog Box. For Main Window, replace hDlg by hwnd and in return statement instead of casting **BOOL**, cast to **LRESULT**.

5th Chapter

Menu

Menu Diagram

Important features of Menu are shown in above figure. Menu is the most primary, rather first way of user-interaction introduced in **GUI**. A typical Menu has following parts :

a) Menu Bar or also called as Main Menu.
b) Top-level Menu items : i) Simple (which directly does some action)
ii) Popup (which displays another Menu)
c) Popup Menu.
d) Popup Menu items : i) Simple (which directly does some action)
ii) Sub Popup (which displays another Menu)
e) Menu Separator.
f) Menu Break / Menu Bar Break.
g) Hot Key or Shortcut Key.
h) Accelerator Key.

Menu Programming

Menu programming requires 3 steps :

I. Creating the Menu.
II. Invoking the Menu.
III. Processing the Menu.

I. Creating The Menu

Menu is the resource. It can be created by two ways :

A) Writing resource script for Menu.
B) Using *API* in source code.

Out of these two ways, 1st is most common and simple.

A) Writing resource script for Menu :

The syntax of creating Menu as resource '.rc' file is similar to the creating Dialog Box as resource. The resource statement for Menu is **MENU**. The syntax is :
<Desired Menu resource name> MENU <options>

This resource statement is then followed by **BEGIN** and **END** which forms a block and inside that block, Menu items are written.
The resource statement for Menu item is **MENUITEM**. The syntax is ….
MENUITEM < "string" >, <ID value>, <options>, <styles>

The **MENUITEM** resource statement is used for those Menu items which directly do some actions.
But some Menu items are such that they don't do a direct action, but instead they show another Menu. Such menus are called as Popup Menu. The resource statement for creating Popup Menu is **POPUP**. The syntax is same as MENU resource statement……..
POPUP < "string" >, <options>
Besides above important things, they are five additional things :

a. Menu Separator : Its resource statement is
   MENUITEM MENUSEPARATOR
b. Menu Break : When the number of top-level Menu items cannot fit in maximized window's Menu bar, then Main Menu can be split into two Menu Bars. The option is **MENUBREAK** which should be used in the **MENUITEM** or **POPUP** resource statement of that item from which you want to break the Menu Bar into the new Menu Bar, means Menu item to which you used **MENUBREAK** options will also get displayed in new Menu Bar.
c. Menu Bar Break : When the number of Menu items cannot fit in Popup Menu, then Popup Menu can be split into two Popup Menus. The option is **MENUBARBREAK** which should be used in the **MENUITEM** or POPUP resource statement of that item from which you want to break the Popup Menu into the new Popup Menu, means item to which you used **MENUBARBREAK** option will also get displayed in new Popup Menu.
d. Hotkey or Shortcut Key : If we carefully see the string of Menu item, there is an 'underscore' mark below one of the letter called as hotkey. It indicates that whenever a Menu is displayed the related Menu items can be directly invoked by pressing these hotkey on keyboard without using mouse. But note that, in a single Menu, every hotkey for different Menu items must be unique. e.g. For Save & Save As Menu items, we cannot use 'S' as hotkey for both. Instead, 'S' will be hotkey of 'Save' and 'A' will be the hotkey for 'Save As' as shown in the figure.
   How to give such hotkeys ?
   : Specifying the hotkey for Menu item is very simple. Decide the unique letter and just type '&' before it. e.g. '&Save' , 'Save &As'. On display this '&' is converted into underscore and placed below the letter before which you used it.
e. Accelerator Key : We will see it later on.

Now using above syntax, we will create Menu as shown in figure. Open your resource script (.rc file) in text mode and type as follows :

```
MYMENU MENU
BEGIN

POPUP "&File"

BEGIN
    MENUITEM "&New", IDM_NEW
    MENUITEM "&Open", IDM_OPEN
    MENUITEM "&Save", IDM_MSAVE
            POPUP "Save &As"
                    BEGIN
                            MENUITEM "&Text", IDM_TEXT
                            MENUITEM "&Hex", IDM_HEX
                            MENUITEM "&Ascii", IDM_ASCII
                            MENUITEM "&Binary", IDM_BIN
                    END
    MENUITEM MENUSEPARATOR
    MENUITEM "&Print", IDM_PRINT
    MENUITEM "P&rint Preview", IDM_PVIEW
    MENUITEM MENUSEPARATOR
    MENUITEM "&Exit", IDM_EXIT
END

POPUP "&Edit"
        BEGIN
                ……..
                ……..
                ……..
                ……..
                ……..
        END

MENUITEM "&Help", IDM_HELP
        BEGIN
                ……..
                ……..
                ……..
        END
```

B) Using *API* in source code :

We had already seen in Dialog Box that use resource script makes the program simpler, more modular and more portable for internationalization. But if we wish we can create all resource using *API* in our source code. So instead of '.rc' file for Menu creation, we can use **CreateMenu ( )** and **AppendMenu ( )** to create same above Menu. Steps are :

1. Declare two Menu handles of **HMENU** type. One for Main Menu handle like **hMenu** and other for file Popup Menu like **hFilePopupMenu**. Use **CreateMenu ( )** for **HMENU** and then for **hFilePopupMenu**.
2. Then use **AppendMenu ( )** to append 9 items to **hFilePopupMenu**.
3. And finally by using **AppendMenu ( )** append **hFilePopupMenu** to **hMenu**.
   Even there is another *API* which also can create the Menu and i.e. **CreateMenuIndirect ( )**, which uses Menu template.
   But both above ways are not much practiced today, because they are harder and do not give benefits of resource script.

¾ **Some Important Points :**

As like DIALOG resource, Menu resource also has load options and memory option, but they are not used and kept default. Some common Main Menu styles are **GREYED**, **CHECKED**, **DISABLED**, **INACTIVE**. Some options are there for popup Menu items. But usually Main Menu items do not use **CHECKED** style.

II. Invoking The Menu

Though we created Menu is first step and execute the application Menu will not be displayed, but will be present in memory. Usually program displays Menu in two ways :

A) Right from the beginning
B) Not from the beginning, but sometimes afterwards.

A) Displaying Menu right from the beginning : When we need to display Menu right from the beginning, then just go to the **WinMain ( )**, go to the statement '**wndclass.lpszMenuName**' (now we have NULL assigned here) and specify your Menu name given in resource script (e.g. MyMenu) in double quotes.
B) Not from the beginning : If you don't want to display Menu right from the beginning, but wanted it to get displayed after some user actions, say keypress or pressing a spacebar or clicking left mouse button, then go to respective message handler and inside it, use two functions : i) **LoadMenu ( )** ii) **SetMenu ( )**

e.g. :

```
HMENU hMenu ;
…..
…..
…..
…..
case WM_..... : (e.g. case WM_LBUTTONDOWN for left mouse button click)
hMenu = LoadMenu ( hInst, "MyMenu" ) ;
SetMenu ( hMenu ) ;
break ;
```

This method has one disadvantage. When we display Menu by specifying it by **wndclass**, then destruction of window automatically destroys associated Menu too. But in late way in **LoadMenu ( )**, destructing window will not free occupied by Menu. We should do it explicitly by using **DestoryMenu ( )** *API* as : **DestoryMenu (hMenu)**. This is usually done in **WM_DESTROY**.

III. Processing The Menu

It is very much same as Dialog Box, because Menu items when clicked send the most important message **WM_COMMAND** to its parent window's **WndProc ( )**. In this **WM_COMMAND**, **LOWORD (wParam)** is ID value of Menu item, **HIWORD (wParam)** is zero and **lParam** is also zero. Hence **WM_COMMAND** message handler will have switch of **LOWORD (wParam)**, in which there will be cases of Menu item's IDs, under which whatever action programmer wants to give to the application will be specified.

```
case WM_COMMAND :
switch ( LOWORD ( wParam ) )
        {
                case IDM_NEW :
                …..
                …..
                break ;

                case IDM_OPEN :
                …..
                …..
                break ;

                case IDM_TEXT :
                …..
                …..
                break ;
```

```
            case IDM_BINARY :
            …..
            …..
            break ;

            case IDM_SAVEAS :
            …..
            …..
            break ;

            case IDM_EXIT :
            …..
            …..
            break ;

    }
    break ;
```

- Up till now, we used our desired resource names like DATAENTRY for Dialog Box, MyMenu for Menu. Instead of using such string like names, we can use integers directly. Just be sure they should be unique.
  If you specify integers as resource identifiers, then in **Load….. ( )** function calls in source file like **LoadIcon ( )**, **LoadMenu ( )**, **LoadBitmap ( )**, **LoadAccelerators ( )** while giving their second parameter, we should use **MAKEINTRESOURCE ( )** either as **MAKEINTRESOURCE (9) ; or MAKEINTRESOURCE ("#9") ;** where 9 is supposed to be give resource identifier instead of name.

Dynamic-Link Library

Contents:

1. Definition

Dynamic- Link Library or dll is a binary file which acts as a shared library of functions, which can be used simultaneously by multiple applications though there is single copy of dll in memory.

Though double clicking on dll or executing from command line usually doesn't generate any output, it is an **executable file** because it is compiled linked and then stored separately.

Any application (i.e. exe or other dll) now can call library functions from this newly created dll. Not only functions but data an resources also can be shared by different applications.

Synonyms:

dll is also called as …

- Dynamic Library
- Dynalink Library
- Library Module.

2. Static Linking vs Dynamic Linking

As windows program grows bigger "a single file program" becomes difficult to maintain. Thus "multifile project" appears in picture. After multifile project, there arises a need of such utility which will have a group of functions, available to all. We call this utility as library of functions.

The problem with such library is that, suppose our program needs a single function from a library and suppose library has 10 functions, then including such library in our program unnecessarily add the space of 9 more functions, which are not needed. As library is provider provides library as binary code, we even don't have facility to cut the portion of unnecessary 9 functions and keeping the code of one function that we need. So inclusion of such library increases size of our program's executable file. Such library remains in our code until the end of our program, occupying the memory all the time. It means out of say 100 lines of our program we call the needful function from library at line 5 and then throughout the next program we don't need any function from the library, then too library remains with our program until its end. This is because library is now part of our program's body that can not be cut off. Such type of linking of library with our executable is called as static linking (.lib file).

So we need solution 2 main problems …

1. Library's code should not get added with our executable, increasing executables size.
2. We should be able to cut off the library whenever our task with library gets completed.

That's why dynamic link library is invented. The code of the dll does not get added into the caller's executable and we can free the library whenever we required. This is because the dll is called whenever function inside it is called by the caller's program, at run time (on fly). Such type of run time linking is called "dynamic linking".

3. More About Dll

3.a Advantages of dll

- Code of the dll doesn't get added into the caller program's executable. So executables size remains as it is.
- dll can be called anytime and also can be freed any time. So it won't remain throughout caller program's life.
- Saves memory and reduce swapping.
- Many programs can use single memory copy of dll. In contrast, different copies of static link library must be created for different programs which call them.
- Saves a disk space. Because single dll is enough for many applications which need to call the function in that dll. while when programs need one static library, the code of static library gets added into program's executable making all executables of large size and thus wasting disk space for the single static library usage.
- As dll is compiled, build, linked and stored as separate module, when change into dll are made only dll is to be recompiled, rebuilded & relinked. There is no need of compiling, building and linking of the program which needs that dll. In contrast, when static library is changed then the library and the application calling it has to be recompiled, rebuild and relinked.
- the language of a program and of the dll need not be same. Means VB programs can call VC++ dll. Rather this is the actual method by which VB like interpreted (non-compiled) languages works.
- By placing resources in dll we can ease the method of internationalization of our program.

3.b Disadvantages of dll

Program using dll does not remain self-contained. Means your program depends upon existence of dll. If dll is absent your program's desired task can not be can not be done. Hence sometimes a program requiring one or more dll becomes dependable.

3.c Common examples of dll

The most common dlls that used by virtually all windows programs are kernel32.dll, user32.dll, gdi32.dll. Even Windows OS itself needs them to become minimum operable. Other examples are …

- .fon (fonts)
- .drv (drivers)
- .ocx (VB AciveX Control)
- .exe

3.d About File Extension of dll

A dll can have any extension like .fon, .drv, .exe(though it does not do anything after execution). But as a convention Windows OS uses .dll extension and only .dll extensions having dlls can be loaded automatically by Windows OS. dlls having different extensions need to be load explicitly by calling application.

3.e Where the dlls are kept

The dlls are kept at one of the following four locations …

- In Windows system directory. (This is the most common place of keeping dll. If you see this directory you will find 99% files as dll).
- In Windows directory itself.
- In the caller program's executable directory.
- In any other directory which is in MS-DOS path. It means any directory which is included in 'PATH' variable of autoexec.bat file.

4. How applications and dll work together

When a program is executed, its executing instance in memory is called process. In Win32 every process is given its own 4GB of memory space. Though our memory(RAM) is not of 4GB(it is actually much less than 4GB), Windows Os gives virtual memory space of 4GB to all running processes of their own. This memory is then occupied by program's data code and text. Now as memory is filled by valid contents, it is called as address space. Not a single program is of 4GB, so when a program occupies 4GB memory much space is left. Now when a program calls dll either by loading the dll or by calling one of the functions in dll), though it is a executable type, it won't get its own 4 GB. Instead it is added into the "left space" of 4 GB of calling program. In other words it is said that **"Loaded dll is mapped into the same memory spaceas that of the calling process".** The word mapping in above sentence is very important and is one of the crucial task of the OS.

The actual story of mapping is that – Compiler does not know anything about size of your RAM. So when your compiler compiles the program it gives arbitrary addresses to the variables and functions in your program. As a compiler itself is a program, the addresses given by it are also logical.(It means these addresses are not actual hardware or actual physical addresses). Thus it is the duty of OS to get these logical addresses, then to look in RAM, whether these actual physical addresses are empty or not of the same number as that of logical addresses. If a logical address xyz is there on the RAM and if it is empty, then well, OS assigns actual xyz on RAM to the logical xyz. But if xyz is not on the RAM or if it is there but right now not empty, then OS maps this logical xyz to some physical pqr address. This process is called mapping and note that it is common to all executables(including dll because it is also one of the type of executable).

Now coming back to the dll, we know that dll is mapped into the process's address space. When the process calls one of the functions in the dll, then OS doesn't matter with the name of the function. Rather OS gets pointer to the function and looks for mapped physical addresses in the memory and when finds, executes the function. The occupation of the process's memory by dll is temporary. Means process's address space is occupied by dll whenever process calls (or loads) the dll or calls one of its functions. When the dll is freed(unloaded), the dll's occupied memory is also freed which can be used by the another process to load another dll or to do other task.

In contrast when the library is linked statically(.lib), the it becomes the part of the executable program. Thus memory occupied by the program becomes … Program's actual memory + library memory.

Such memory can not be freed and remains occupied until program itself gets terminated.

When all virtual memory of 4GB of a process get occupied completely(which is very rare situation) , the OS shows 'out of memory' message.

5. Linking of Application with dll

In above paragraph we said that dll gets loaded whenever we either load dll and then call require function from dll without loading it. These two "either … or" situations give us two types of linking of dll with our program …

- Implicit Linking
- Explicit Linking

6. Implicit Linking

When we create a dll, visual studio also creates .lib file. This .lib file is called as **import library**. Note that this .lib file is different from static .lib library that is called as **object library**. As stated before the static link library (i.e. object library contains code of functions that gets added into caller program's executable code. In contrast, the import library does not have the code of functions. Instead it has the information for the linker to set up the relocation tables within the executable file for linking with the actual dll. In other words, it has the information about where to put the function addresses in calling process's 4GB memory. Obviously this information is OS dependant because as seen before "mapping" is the task of OS. The .lib import library is the special library file.

In the implicit linking this import library is linked to the caller's program by adding it **project->setting->link** tab in VC++. Then OS maps the dll when when program(calling the dll) starts. It means the dll is present with the program when program the program is started and dll ends when the program ends.

The caller program now anytime can call any function in that dll as iif the function is in itself.

To the implicit linking, following task must be done …

6.a Task of the dll creator:

- A header file must be given to the caller program. This file contains the prototype of the functions that dll allow the caller to call.
- The .lib import library also must be given to the caller.
- And finally the actual dll is given to the caller.

So 3 files must be given by dll provider to the caller.

6.b Task of the dll caller:

- The header file given by the provider must be kept in the projects directory or in the include directory of VC++.
- It must be included in the source code by #include directory.

- The .lib import file must be kept at one of the 4 locations given on page 5.
- The .lib import file must be linked with the project by adding it into the **project\setting\link\object/library modules** edit box.
- Keep the provider given dll in one of the 4 location given on page 5.

Note – if .lib import lib is not linked as stated above the caller program compiles fine but during build or link or execute it gives the error of "Unresolved external symbols".

In implicit linking dll is mapped right from the beginning of the program but does not linked with it. It gets linked with calling program when one of the function inside the dll is called. So loading of the dll and linking with the dll are two different procedures. In other words, dll is mapped with the process's address space when process starts but does not actually loaded or linked. It gets loaded into the memory when one of its functions is called by caller program.

6.c Advantage of implicit linking

The only advantage of implicit linking is the easiness of the programming No extra loading or freeing of dll is required.

7. Explicit Linking

In explicit linking there is no need of .h and .lib import library of dll to provide. Just the dll and the text information about its functions prototypes and parameter meanings may be enough. But this method is slightly harder than the implicit linking.

7.a Task of the dll creator

- Provide .h file is better. It should have dll's function prototypes, parameter and return value meanings and the declarations of the class and structure if they are parameters or return values of dll functions
- The actual dll

7.b Task of dll caller

- if dl provider gives .h file, then include it in the project and keep it in either project's directory or in include directory of VC++
- The dll file given by the provider must be kept at one of the 4 locations given on page dll-5.
- Load the dll by using its name with either LoadLibrary() or LoadLibraryEx() functions. By this way get the dll program's instance handle.

- Get the required dll function's address by using GetProcAddress() API.
- Now use above function address as if it is your required function.
- Free the library using FreeLibrary() API.

7.c About LoadLibrary() :

Loads or maps the specified dll.

Prototype:
HINSTANCE LoadLibrary(LPCTSTR);
It takes the dll name as its parameter. You can give the path of the file(just recall the method of using \ character in a string by using \\).

Return Value:
On success, it returns handle of the instance of dll module.
On failure, it returns NULL.

If you don't specify the path OS searches at 4 locations given on page dll-5 and if it can't find dll returns NULL in return value.

7.d About LoadLibraryEx():

Loads or maps the specified dll. This can be .exe.

Prototype:
HINSTANCE LoadLibraryEx(LPCTSTR,HANDLE,DWORD);
Param 1: Name of the dll or exe with or without path. Param 2:
reserved, must be NULL.
Param 3: flag which specifies the action when dll is locked. (see MSDN for more details).

Return Value:
On success, it returns handle of the instance of dll module.
On failure, it returns NULL.

7.e About GetProcAddress()

Return address of the specified dll function.

Prototype:
FARPROC GetProcAddress(HMODULE,LPCTSTR);
Param 1: Handle of dll instance returned by LoadLibrary() or LoadLibraryEx(). Note- though LoadLibrary() or LoadLibraryEx() return HINSTANCE type handle, it can be directly used as HMODULE parameter at this function. No typecasting is necessary.
Param 2: Name of the function as string.

Return Value:
On success, it returns void address of the dll's required function.
On failure, it returns NULL.

7.f About FreeLibrary():

Frees the loaded library()(i.e. un-maps the library from the process's address space).

Prototype:
BOOL FreeLibrary(HMODULE);
Param 1: The only parameter is the handle of dll's instance.

Return Value:
On success, it returns TRUE;
On failure, it returns FLASE.

Note:

Every call to LoadLibrary() or LoadLibraryEx() for a dll must have corresponding call to FreeLibrary() for the same dll.

Also note that calling FreeLibrary() frees dll from your program's address space. Other program can continue with the same dll.

7.g Sample Code of Explicit Linking:

```
WndProc()
{
        //variable declaration
        int iNum=5,iSquareOfNum;
        HINSTANCE hLib; // it can be HMODULE
        hLib; typedef int (*FuncPtr ) (int); FuncPtr
        pfnSquare;
        hLib=LoadLibrary("Square.dll");
        pfnSquare=(FuncPtr)GetProcAddress(hLib,"MakeSquare");
        iSquareOfNum=pfnSquare(iNum);
        FreeLibrary(hLib);
}
```

In above sample code we assumed that …

- There is Square.dll file given by provider and kept by caller at one of the 4 locations given on page dll-5.
- dll provider in supporting textual help files gives prototype of function as
  int MakeSquare(int);

Now LoadLibrary() loads the dll. We declare the pointer to a function similar to given prototype by typedef. So here FuncPtr is the representative of MakeSquare(). Then we create a variable of FuncPtr type as pfnSquare. Then by calling GetProcAddress() by giving it the actual function name "MakeSquare" we get its address in pfnSquare variable. Note that the return value of GetProcAddress() is FARPROC and we need address of address of FuncPtr type function. Hence we typecast the return value to our required type. Now calling pfnSquare() is nothing but calling MakeSquare(). After completing the task with dll we free the library by using the FreeLibrary().

7.h Advantages of explicit linking

Though the process of explicit linking is harder than implicit linking, there are many advantages of explicit linking over the implicit linking …

- No need of .lib import file keeping and linking.
- In implicit linking dll is mapped right from the beginning of execution of the program. In explicit linking you do this whenever you need by calling LoadLibrary().
- If required dll is not found, OS terminates the program in implicit linking. While in explicit linking if LoadLibrary() fails, then by checking its return value you have an option of loading another dll having same type of functionality that you require.
- If your dll has DllMain() function, and if it fails then implicit linked dll and the calling program, both gets terminated by OS while in explicit linking even if DllMain() fails, error is flagged but calling program keep on going.

- In implicit linking, if .lib import file is changed, the program which which calls the dll must be relinked. While in explicit linking, as there is no question of import library, program needs no relinking even no recompiling and no rebuilding.

But explicitly linked all may cause some problems in multithreading use o4f that dll.

8. Exporting functions from dll:

We know that an application starts the required dlland then calls one or more functions from that dll to do its task.

But allow an application to use such dll functions, dll must export such functions. Thus dll contains export function table. Dll may have 2 functions in it. The exportable functions which are going to be used by any application which calls this dll and local functions which are called within the dll (just like any local function) and thus are not exportable and are private to dll.

To see theexportable function list in a dll, we can use DUMPBIN program with /EXPORTS switch and the dll's name.

Exporting of the functions from a dll can be done by 2 ways ...

- Using __declaspec(dllexport) keyword.
- Using module definition (.def) file

Both these methods must use standard calling conventions (i.e. __stdcall) VC++ sets this by default by using /Gz compiler switch. (__stdcall & WINAPI are on and the same).

8.a Using __declspec(dllexport) keyword

This is the recent method of exporting data or functions from a dll. In old days and yet now the method alternative to this is "exporting by .def file"

Just prefix this keyword at left to the function declarator, while writing the exportable function's body.

E.g.: In our example we have exportable function MakeSquare(), So its body will now look like …

```
__declspec(dllexport) int MakeSquare (int
num); {
        :
        :
        :
}
```

Use of this keyword does 2 things …

- No need of .def file now.
- It writes the necessary "export function table" in dll

Instead of using __declspec(dllexport) directly, Microsoft encourages a method of creating a macro for this keyword and use this macro wherever necessary. e.g.: #define EXPORT __declspec(dllexport)
Now onwards use the macro EXPORT whenever __declspec(dllexport) necessary.

Advantages:

- Conviniance
- Easy to use
- No need of .def file

Disadvantages:

This method is good if the dll and its applications both are in control. Because if new exports are added both dll and applications have to be rebuilt.

8.b Using Module Definition File (.def file)

This is a text file and must be included in your dll project before building the dll. So open your project's workspace in MS VC++, then click on new, then in "new" property sheet click on "Files" tabs, then select the "Text File" option. On right side of the wizard, confirm for your project name in "Add to Project" edit box. Also confirm that that the checkbox of "add to project" is checked. Then in "file name" edit box, type desired file name with .def extension and press ok. MS VC++ editor will open an empty window with your .def file name. Now we are ready to type .def file.

The .def file for a dll requires 2 necessary and 1 optional statements. The 2 necessary statements are …

- LIBRARY (dll name with or without extension)
- EXPORTS statement which follows the names of your exportable functions with or without ordinals.

The optional statement is DESCRIPTION which is followed by the string (say copyright string). Enclosed in single quotes. According to the above syntax the .def file of our square dll will be square.def which is as follows … LIBRARY square.dll

```
DESCRIPTION 'copyright : astromedicomp'
EXPORTS
        MakeSquare
```

Notice that the exportable function is right intended for better readability.

8.c Ordinance:

Ordinance or ordinal value is a unique integer number given to the exportable data or a function after the name of the function, in EXPORT section of the .def file. The syntax is @<number>. So under EXPORT section our exportable MakeSquare() function can be written as …

```
:
:
EXPORTS MakeSquare @100
```

Here 1 is the ordinal value of the MakeSquare(). The range of ordinal value is 1 to N where N is the number of exported functions from your dll. The sequence is not important, but if you have say 5 exportable functions, then your ordinal value would be @100 to @104.

Note:

1. *While building a dll that uses .def file, the compiler first creates .lib import file, then creates .exp export file and then uses ordinals or names in .exp file to link with import library to create final .dll file.*
2. *Besides these 3 statements, the .def file also may containNAME,STA CKSIZE,SECTIONS,VERSION statements(see MSDN for details about these).*
3. *As compiler keeps ordinal value in "Exported Function Table" of dll, it also links the ordinal value of a function with its name in .lib import file. So application using dll can call function by its name. Application need not to worry about ordinal.*

8.d Advantages of using ordinal values with function names in EXPORT section of .def file

If you use ordinal value with the exported function with NONAME attribute as …

```
EXPORTS
    MakeSquare @100 NONAME
```

Then the ordinal value is assigned to your function name and compiler, in your dll's "exported function table" stores the ordinal value of your exported function rather than function name's string. If your dll is going to export many functions, then storing all these function names as strings in dll's "exported function table" will increase the size of dll. But NONAME attribute will force the compiler to store functions as ordinal value and thus optimizes the size of your dll.

8.e Advantages of .def file

- The most important benefit of using .def file is that, even if you add more exported functions to it (by assigning higher ordinal values than present one), only dll is needed to be rebuided, No need of rebuilding of application is required. This is important particularly in the situations where you are going to build such a dll (like system dll) which is going to be used by many applications. Here using __declspec(dllexport) is hazardous because many applications are to be rebuidled. This is th reason that Windows OS itself builds its dll by using .def file. And this is the reason that even if the system is updated (from Win95 – Win98 or to Win2000) our application using those dlls need not to be rebuilded.
- The second advantage is of ordinal value as explained before.

8.f Disadvantage of using .def file

While compiling a dll, compiler, compiler does same "Name Decoration" to your dll exported functions. This can cause problems when a dll created by c++ is going to be used by a "c" program. The .def file having dll work smoothly for a c++ program with c++ dll but not with "c" program.

As a solution …

- Make your c++ dll as it is. Then use DUMPBIN utility for your c++ dll and get the decorated names of your exported functions. Write them down and then type as it is in your .def file and rebuild the dll OR
- Keep .def file as it is but in your c++ dll's .cpp source code file. prefix decorator of every exported function by extern "C" linkage specifier.

Another problem of the 1st method is that if you place the decorated function names of your exported functions in .def file, both the dll and the application using the dll has to be rebuilded with the same version of VC++. Thus the 2nd method is much better for c++ dll.

See MSDN for more information on "decorated names" and on "how to use DUMPBIN to find decorated names".

9. Exporting data from dll:

There are 4 methods of exporting data from dll …

- You can use __declspec(dllexport) keyword or its macro as prefix to the exportable data type. You should also provide .h file informing about your dll's exportable data to user that wants to use your dll.
- You can use .def file to export data where the exportable data name is written under EXPORT section with or without ordinal value. Here too you should provide .h file to the user who wants to use your dll.
- By putting exportable data in shared memory section. To do this you should use #pragma data-seg directive as follows …
  # pragma data-seg (<any desired name say temp">)
  :
  :
  exportable data with its data-type and initialized.
  Note: Initialization is must, if pointer initialized it to NULL. If array then initialized it like char str[255]={"\0"};
  :
  :
  # pragma data-seg()
  By above code, we force the data to be kept in shared memory section.
  Now to tell about our exported data to the linker, add the following statement to the source code file of your dll … #pragma comment (linker,"/SECTION:temp,RWS")
  there RWS stands for Read, Write and Shared attributes resp.
  All above 3 methods are theoretically correct. But 1st and 2nd are quite unreliable. 3rd is good but occupies a lot of shared memory when the data is large.
- By making exported data as return value of exportable functions:
  This is the best and most reliable method of exporting the data. Some reconstruction of the functions might be necessary but failure is impossible. If large data or more data members needs to be exported then it is better to declare them globally in dll's source code and then create a structure which includes all exportable data. Then write an exportable function which returns this structure as its return value. Also provide .h file to the caller stating your defined structure.

10. The DllMain() function :

As like windows program has WinMain() function, the dll has its DllMain() function. But surprisingly this is optional. Means your dll program may not have DllMain() function if dll is going to be used solely by import library.

But if caller is going to use this dll in multitasking/multithreading program or caller wants to load dll by LoadLibrary(), then use of DllMain() in dll is must. It is called twice by the system(starting and ending of dll).

So if you are both the caller and dll maker then you can avoid DllMain() in your dll. But if you are going to create a third party dll i.e. your dll is going to be used by many applications tha you don't know, then you can not anticipate that whether your dll will be called by LoadLibrary() or not. Hence it is better to include a placeholder (i.e. Doing noting – you may write your own DllMain() like function) DllMain() in your dll program.

DllMain() is known as "Entry Point Function" of dll application.

Prototype:

BOOL WINAPI DllMain(HINSTANCE, DWORD, LPVOID);

Param 1: A handle of dll. This value is the base address of dll (i.e. 0^{th} position of file Pointer). This handle is passed to your dll by the OS and this is the handle retrieved by the caller using LoadLibrary().

Param 2: This is the flag indicating for what reason this dll is called. There are 4 possible vallue of this parameter.

- DLL_PROCESS_ATTACH:
  Indicates that the dll is called by caller's process. The message handler of this value in DllMain() can do process specific initialization.
- DLL_PROCESS_DETACH:
  Indicates that now calling process is unloading the dll from its address space. The message handler of this value can do uninitialization of the initializations done in DLL_PROCESS_ATTACH.
- DLL_THREAD_ATTACH:
  Indicates that the calling process is created a new thread and this new thread wants to load this dll. Thus message handler of this value can do thread specific initialization.
- DLL_THREAD_DETACH:
  Indicates that the calling process's calling thread is exiting. Means obviously it wants to unload the dll. So the message handler of this value can do un-initializations of initialization if made in the handler of DLL_THREAD_ATTACH.

Param 3: This is reserved and usually NULL for dynamic linking.

Return Value:

On success, the return value of DllMain() is TRUE.

On failure, the return value is FALSE.

If dll is called by LoadLibrary() and if dll's DllMain is fails, then the handle return by LoadLibrary() to calling process is obviously NULL.

11. How to use DllMain() in dll's source code:

As explained before you can omit DllMain() in your dll, when you want ot use it, you can use it in one of the following form …

- BOOL WINAPI DllMain (HINSTANCE hInstance,
  DWORD dwReason,
  LPVOID lpvReserved)

```
{
        return (TRUE);
}
```

- BOOL WINAPI DllMain (HINSTANCE hInstance,
  DWORD dwReason,
  LPVOID lpvReserved)

```
{
        switch(dwReason)
        {
                case DLL_PROCESS_ATTACH:
                // process specific
                initializations break;
                case DLL_THREAD_ATTACH:
                // thread specific initialization.
                break;
                case DLL_THREAD_DETACH:
                // thread specific un-initialization
                break;
                case DLL_PROCESS_DETACH:
                // process specific un-initialization
                break;
        }
        reutrn(TRUE);
}
```

Notes:

- *You can keep all message handlers empty with just case and break statements of every message handler.*
- *You can write only two mutual (i.e. either 2 process or 2 threads) message handlers with or without initialization and un-initializations.*
- *But return(TRUE) statement is must for both ways.*
- *After closing the brace write dll's local and exportable function bodies. You must include windows.h in every dll to validate macros and functions.*

12. Resource-Only dlls:

As explained on page dll-4 dlls need not contain only functions, we can export only resources from a dll too. Such type of dll will then be called as resource-only dll. To create such a dll follow the same procedure as of creating dll as explained before.

- In .c source code filewrite DllMain() by 1st method.
- Create .rc file, and write all exportable resource statements(if they have IDs then declare IDs in .h)
- Compile and rebuild the dll.

The caller program should load the dll and then can call Load<resource>() functions such as LoadBitmap() or LoadIcon() as usual. The header file given by creator will give the resource name or identifiers in .h file. Just remember that all Load functions use library instance handle.

12.a Using __declspec(dllimport)

On pages dll- 7 to dll-14 we explained how a caller program can call dll'd function, there is yet another way. In caller program you can import dll's exported functions by __declspec(dllimport) specifier, and then use such functions or data as usual. Theoretically this specifier is clear but in real practice it may fails.

12.b Final Words about Dll

- Usually dll does not send or receive messages, but if you wish you can use GetMessage() or PeekMessage() in your dll. It pulls the messages from the message queue of the calling program.
- The resource loading functions of the calling program should use the dll's instance handle (returned by LoadLibrary() or GetModuleFileName()) as their 1st parameter. If the dll wants to call resource loading functions for resources within the calling program, then dll should use the calling program's instance handle given by GetModuleFileName().
- If you want to create a window in dll so that your dll becomes to give UI (unusual in practice) then you must use dll's instance handle in WNDCLASSEX declaration and CreateWindow() call.
- If your window wants to display dialog box of its own, then you can use dll's instance handle (i.e. 1st parameter of DllMain() in DialogBox() function and set its parent handle to NULL.
- If your dll exports data member by method 3 on page dll-20, then calling program should use WM_DATACHANGE message and process its handler.

MULTITASKING & MULTITHREADING

Multitasking is the ability of an operating system to run multiple programs concurrently (or say simultaneously).

Though we saw multiple processes (i.e. executing programs) running simultaneously, it is actually an illusion. And not the reality. This is because CPU can give specific scheduled time to a single process at a time. But if time-slices are very short, then switching of CPU from one process to another occurs very fast and we can't experience the gap between two time slices, looking like two programs are running simultaneously.

Win9x and win NT supports 2 types of multitasking.

1. Multiprocessing:

Here two or more executable programs (i.e. processes) can run concurrently. His is also called as process-based-multitasking.

2. Multithreading:

Here two or more parts of same executable program (i.e. process)-say two or more functions can run concurrently. This is also called as thread-based-multitasking. So multithreading is the ability of a program to multitask within itself.

- window 3.1 (i.e. 16 bit windows) somehow supports process-based multitasking but does not support thread-based-multitasking.
- DOS 6.22 though does not support both types of multitasking, TSR(transient-stay-resident) programs give slight ability of multitasking to DOS 6.22.

➢ When multitasking is necessary?

Multitasking is necessary mainly for 2 reasons…

1. When a program wants to use functionality of some other program. In such cases program A starts program B in its code and both programs may run concurrently.
2. A program has one big job and another small jobs. If small jobs are going to be completed irrespective of the big job, then keeping small jobs waiting for the completion of big job is not good programming practice. Here we should start the big job, then tell it to run in background and during its background working allow small jobs to complete their tasks in the foreground.

➢ What are the ways of multitasking?

1. Using Spawn or exec functions of C library with _P_NOWAIT Flag (because of P_WAIT flag is used, then parent process will start working only after completion of child process so this is not multitasking) This Flag returns immediately and big job keeps working in background, while small jobs may then work in foreground.
2. Creating a thread inside our program to do a big job, while we can carry out another small jobs till the big job is going on in the background.
3. Using PeekMessage( ) function instead of using GetMessage ( ) function, because GetMessage ( ) function is used to retrieve the next message in the message queue, so if message is empty, means there are no messages in the message queue GetMessage ( ) will not return. On the contrary the PeekMessage ( ) Return's quickly, even if there are no messages in the message queue. Thus a program can do long jobs in the background while performing small jobs in the foreground.

Multithreading Environment:

In Multithreading scenario, a program can split itself into separate pieces known as "Threads of Execution". These threads can run concurrently. In simple terms you can make a function in your program as a thread, rather splitting a program into threads means splitting the programs in multiple functions as usual, and then assign a thread to each (or required) function. These causes all threads to run concurrently and thus no CPU time is wasted.

Note that every program (i.e. process) is a thread in other words every process has at least one thread. Then a question may arise, which function is the thread. Obviously if your program is a C program, then its thread will the Main ( ) function and if your program is a window program in C, then the thread will be the WinMain ( ) function

Though every program has one thread at least, it is not a multithreaded program, because it has only one thread by default. Hwne you assign other functions in your program as threads means func1 as thread 1, func 2 as thread 2 and so on, then there will be multiple threads and now you can call your program as multithreaded program. The default thread (i.e. of Main( ) or of WinMain( )) is called as Main thread or Primary Thread. Other threads in your program are secondary threads.

➢ How to design your program to be multithreaded?

1. leave UI part &messaging part to main thread:

Means window creation (single or multiple), window produces (single or multiple), menus, dialog boxes, child window controls, common controls and all users interactions i.e. message handling should be done by main thread, i.e. by WinMain( ) &WndProc( ) or DlgProc( ) as usual.

2. other threads should do performing iterative calculations etc. all other tasks, such as opening & reading large files, or writing data to multiple files.

In other words, your program's primary thread should be the big boss and other threads should be its staff members. Big boss will call respective staff member o perform a job and while job is "going on" big boss will continue to communicate with the outer world.

This does not mean that, other thread cannot do UI or messaging. They can do this but only under control of the main thread. You can think modeless dialog box for the understanding purpose. In modal dialog box, you cannot switch to other part of your program until you close the modal dialog box. So this is obviously not multitasking. Now think about modeless dialog box, here you can minimize modeless dialog box and without closing it you can switch to other part of your program, so modeless dialog box works as thread and it has its own UI and also its controls have their own massaging task with its parent.

- **Threads & Resources.**

1) Threads in a program are all parts of the same program
2) Thus they share the programs memory and other resources, in other words, they are not self executing, hence they do not have their own memory, but they get the memory from the main processes memory on the basis of the execution flow and priorities defined.
3) They also share open files, global variables and static variables of a program (Thread local storage-TLS)
4) But the very important note is that, they have their own stack, means local variables of a thread remain local to it and only to it.
5) Thread also gets its own process and math co-process state, thus it gets its own CPU time.
6) Though program designers are advised to keep massaging task to main threads and not to secondary threads, this is for their own convenience. In actual sense, thread gets its own message queue as soon as it is created
7) One thread can kill another thread, if some undesirable things occurs and on the top, main thread can kill or suspend any thread at any time if it desires.
8) Using thread local storage (TLS) two or more threads can use each other's static variables and also can use same function in that program.
9) Though any thread can kill another thread it is always better, rather best, to let the thread get terminated on its own when its associated function finishes or let the main thread allow terminating the secondary thread on its exit.
10) Once started, every thread is independent of each other, but if you exit the program, that is when main thread terminates, all its secondary threads gets terminated automatically.

- **Problems of multithreading:**

1. as win 32 system is "preemptive" in nature, os can iterupt any thread at any time and thus can cause some undesirable effects. Unfortunately no solution.

2. Race condition: as it's name suggests, it is like a competition. Multiple threads may rush at a time to access a resource such as file. Or any thread may expect to get some data from another thread, and the data is yet not prepared.
   To overcome such situations, os uses a method of "synchronization", which can be implemented by the programmer by using semaphores, critical section, event objects etc.
3. synchronization itself may cause problem called as "Dead Lock". This happens when two threads block each other's execution and only possible to resume the execution is by proceeding, but as they are blocked, they can not proceed. Such continuous circle is called as the "Dead Lock". Careful organization of multithreaded program can avoid such dead locks.

➢ **How to create a thread?**

The win32 API for creating a thread is CreateThread( ). The prototype is…….

```
HANDLE
CreateThread(LPSECURITY_ATTRIBUTES,DWORD,LPTHREAD_START_ROUTINE,LPVOID,DWORD,LPDWORD);
```

1st parameter- it is a pointer to SECURITY_ATTRIBUTES structure. This determines weather the handle of the thread, returned by these functions is inheritable to child processes or not. This parameter is applicable to NT hence win9x programmers keep this as NULL. Which allows the program to run in NT system too. Specifying NULL gives default security attributes in NT.

2nd parameter- as stated before every thread has its own stack. If you wish you can give desired stack size in this parameter. If this parameter is kept 0 (and usually programmers do the same), then system assigns default stack size. On demand os can expand this size on it's own.

3rd parameter- this is the name of the function that you want to get executed when this thread is created. This function when defined, must have prototype…..

```
DWORD WINAPI ThreadProc(LPVOID);
```

Here "ThreadProc" is just aplaceholder. You can use your function's name in its place.

4th parameter- if you want to pass a value to the newly created thread, you can pass it through this parameter by casting your value to LPVOID type.

5th parameter- any additional flag that controls the behavior of newly created thread is written as this parameter. The possible value is CREATE_SUSPENDED. If this value is used, then the newly created thread

gets createdin suspended state (means ready to execute but right now not executing). To execute such suspended thread you must use the ResumeThread( ) API. If this parameter is 0, then the newly created thread starts executing immediately as soon as it is created.

6th parameter- os returns thread ID in this parameter. In win9x, this parameter should not be NULL. In NT, this parameter can be NULL and if it is NULL, os does not return threads ID.

On success this function returns HANDLE type thread's handle which has THREAD_ALL_ACCESS attribute. So this handle can be used in any other function too. The thread handle can be explicitly destroyed by using CloseHandle( ) API. Or otherwise the thread handle automatically gets destroyed, when main thread (i.e. program) ends. **On failure** this function returns NULL.

* The CreateThread( ) function has a little problem. When it is used, you should avoid to use old c library functions such as sprintf( ), strlen( ), strcpy( ), strcmp( ) etc. because some amount of memory Leak may occure. (Memory Leak- loss of small amount of memory). Thus instead of these old c library function you must use new one, like wsprintf( ), lstrlen( ), lstrcpy( ) etc.

if you want to use old c library functions necessarily and wanted to create a thread, then don't use CreateThread( ). Instead you should use _beginthread( ) or –beginthreadex( )functions. To use any of these functions you must include process.h file in your program. Though _beginthreadex( ) has some extra functionality than the _beginthread( ), the –beginthread( ) works much fine in most of the multithreading situations. Its prototype is……

unsigned long _beginthread(ThreadProc, unsigned int, void *);

where,

1st parameter- it is the name of the function that you want to get executed when the thread is created. This function must have the prototype……

void _ _cdecl ThreadProc(void *);

2nd parameter- it is the stack size. You can use 0, in this the default stack size as equal as the calling thread(i.e. main thread) is use.

3rd parameter- it is any value that an application wants to pass to the newly created thread.

So _beginthread( )'s parameters resembles with CreateThread( )'s parameters. I.e. 1st parameter of _beginthread( ) resembles with 3rd parameter of CreateThread( ). The 2nd parameter of _beginthread( ) resembles with 2nd parameter of CreateThread( ) and the 3rd parameter of _beginthread( ) resembles to the 4th parameter of CreateThread( ).

* Threadproc:

as stated in the beginning, thread usually executes a user defined function. Both CreateThread( ) & _beginthread( ) requires name of this function as one of their parameter. This function is called as "ThreadProc". Note that, "ThreadProc" string is just a place holder, you should use your function name in its place.

The most important thing about this function is that, you can not choose any returns type or any parameters. You must choose the given prototype for ThreadProc( ) for your thread creating function. Means if you use CreateThread( ) to create the thread, then your ThreadProc must have prototype ....

DWORD WINAPI Threadproc(LPVOID);

And if you use _beinthread( ) function to create the thread, then your ThreadProc must have prototype.....

Void _ _cdecl Threadproc(void *);

This is the function which will get executed when your thread starts execution. Obliviously you should include "thread related code" into this function.

* **project setting** for standard C library based multithreading application:

when you cerated a new project in MS VC++ for multithreading application, you should do following changes to project/setting....

1. go to project menu, click on settings.
2. click on C/C++ tab.
3. in "category" combo box, select "code generation".
4. in "use run-time library"combo box, if your project is created for "debug version", then select "debug multithreaded" option or if your project is created for "release version", then select "multithreaded" option.

Why? Usually your single threaded .exe/dll application has compiler flag set to /ML. And your runtime library is LIBC.LIB. when youu create multithreaded application, the compiler flag must be runtime library LIBCMT.LIB(instead of LIBC.LIB).

This is because standard library of above 2 .lib files contains different codes for those library functions which has some static variables shared among their executions. To allow this sharing in multithreaded application, a single thread supported library LIBC.LIB must be changed to LIBCMT.LIB. (here MT stands for multithreading).

5. click on 'ok' and then proceed to write source code.

➢ **How to terminate a thread?**

The programmers who use CreateThread( ) to create a thread should use TerminateThread( ) or ExitThread( ) functions to terminate execution of thread.

TerminateThread( )- prototype is....

BOOL TerminateThread(HANDLE,DWORD);

Where,
1st parameter- it is the handle of the thread that we want to terminate. This handle is same that returned by CreateThread( ) function)

2nd parameter- it is the exit code for the thread. To get this exit code we should use the GetExitCodeThread( ) function whose prototype is…
BOOL GetExitCodeThread(HANDLE,LPDWORD);
Os puts the exit code of the thread in 2nd parameter of this function. This exit code then can be used as 2nd parameter of TerminateThread( ) function.

On success function TerminateThread( ) returns TRUE(non zero) or else FALSE(zero).

ExitThread( )- prototype is….
VOID ExitThread(DWORD);
The only parameter is the exit code of the thread that can be obtained by using GetExitCodeTherad( ) as shown above.

* the function ExitThread( ) is more preferred than TerminateThread(0because TerminateThread( ) on termination of the thread does not deallocate stack of the thread, while ExitThread( ) after executing the thread, deallocates thread's stack smoothly.

* actually programmers are advised to allow the thread to terminate on its own when its ThreadProc( ) gets completed or when main thread terminates (i.e. program exits) all its child threads get terminated automatically.

But for some reasons you want to explicitely terminates a thread, then it is better to use ExitThread( ) when the thread is created by CreateThread( ).

Now if parameters do not use CreateThread( ) to create a thread, instead they used _beginthread( ) then the function to terminate the thread is _endthread( ), whose prototype is….
Void _endthread(void);
No parameter and no return value, hence easy to use.

- **Suspending and resuming a thread:**

Sometimes it may be necessary to suspend execution of a thread for some time and then again resume its execution after certain condition if fulfilled.

To suspend a thread, the win32 API is the SuspendThread( ) function, whose prototype is….
DWORD SuspendThread(HANDLE);
The parameter is the handle of the thread, returned by the CreateThread( ) function.

To resume a suspended thread, the win32 API is the ResumeThread( ) function, whose prototype is….
DWORD ResumeThread(HANDLE);
Here too, the parameter is the handle of the thread, returned by CreateThread( ) function.

Note- both above function return DWORD type, which is the suspend count. Means system keeps suspend count of the suspended thread. Obviously if the suspend count is 0, the thread is not suspended and if it is nonzero, then the thread is suspended. Each call to SuspendThread( ) increments the suspend count whereas each call to ResumeThread( ) decrements it. Thus in a multithreaded program there must be equal number of calls to SuspendThread( ) and ResumeThread( ) functions.

Thus both functions returns suspend count (weather incremented or decremented) on their success and on failure they return –1.

Sleep( )- when a call to SuspenThread( ) is made, the thread remains suspended until the call to ResumeThread( ) is made. But when we are sure that we want to suspend a thread for short duration of time, then instead of using SuspendThread( ) and ResumeThread( ) combination, we can use Sleep( ) function, whose prototype is....

VOID Sleep(DWORD);

The parameter is the time in milliseconds for which we want to suspend the respective thread. After the time duration gets elapsed, threads resumes automatically.

As there is no thread related parameter (such as handle of thread) to Sleep( ) function, it is a generalized function and thus should be used in respective thread's ThreadProc( ) function.

* Another use of Sleep( ) function is that, when a thread wants to terminate its time slice (prior to the end of actual time slice) and give it to another thread, then this function can be called with parameter 0 specified.
* When Sleep( ) function is called, only the thread, in which it is called, gets suspended, thus system can continue running other threads.
* Don't call Sleep( ) in primary thread (i.e. in Main( ) or WinMain( )) in which there are GUI. Because this slows sown performance considerably.

- **Thread priorities:**

The discussion up till now clearly states that, though programmer creates, suspends, resumes or terminates a thread, the os has supreme most control on it. As seen in "problems of multithreading", the os can interrupt any thread at any time. This happens because os gives a created thread certain priority and if higher priority dominates the situation and thus if os decides a thread to terminate then the comparative lower priority thread may get terminated by os.

So "Priority of a thread" is an important aspect od multithreading. This is not mainly concert with the programmers. Rather programmers are suggested not to tamper with the default priority of a thread. But knowledge of it may be useful in certain situations.

Each thread has its own priority, set by os. The thread's priority is the entity, which determines how much CPU time slice it receives. Low priority

thread receives smaller time slice while high priority thread receives comparatively larger time slice.

The thread's priority setting is the combination of 2 values- priority class of the process and priority setting of the individual thread, relative to the priority class of the process.

In other words, thread priority is determined by,
- process's priority class &
- thread's priority level.

* **priority class-** priority class is for the process. So it becomes mandatory to all threads created by that process.

We can get the current priority class of a process by calling GetPriorityClass( ) API, whose prototype is….
DWORD GetPriorityClass(HANDLE);
Note that, the parameter is not the handle of the thread. As priority class is concerned with process, it is the handle of the process. This is the handle returned by OpenProcess( ) (for current process) or CreateProcess( ) (for new process). This function on success, returns the current priority class of the process as DWORD type value. On failure it returns 0.

The available priority classes are as follows…
1. REALTIME_PRIORUTY_CLASS – highest priority
2. HIGH_PRIORITY_CLASS
3. NORMAL_PRIORITY_CLASS
4. IDLE_PRIORITY_CLASS – lowest priority

by default os gives NORMAL_PRIORITY_CLASS to all running processes.

As stated before you should not change this default priority class, because changing this may impact on your system performance. Cut if deliberately you want to do this in special situation, then you can do it by using SetPriorityClass( ) API, whose prototype is….
BOOL SetPriorityClass(HANDLE,DWORD);
1st **parameter-** it is the handle of the process of which you want to set the priority class.

2nd **parameter-** it is the priority class that you want to set. This parameter has one of the above 4 priority classes.

* **thread priority:** this is the CPU time slice of a thread in its process.

To get the current thread priority, you can use GetThreadPriority( ) function, whose prototype is….
Int getThreadPriority(HANDLE);
The parameter is the handle of the thread. On success this function returns the current thread priority. And on failure it returns the THREAD_PRIORITY_ERROR_RETURN value.

The available thread priorities in win3 are….
1. THREAD_PRIORITY_TIME_CRITICAL - (value) 15 – haighest

2. THREAD_PRIORITY_HIGHEST – 2
3. THREAD_PRIORITY_ABOVE_NORMAL –1
4. THREAD_PRIORITY_NORMAL – 0
5. THREAD_PRIORITY_BELOW_NORMAL - -1
6. THREAD_PRIORITY_LOWEST - -2
7. THREAD_PRIORITY_IDLE - -15 – lowest

by system gives THREAD_PRIORITY_NORMAL type thread priority. We can change this by using a function SetThreadPriority( ) whose prototype is….

BOOL SetThreadPriority(HANDLE,int);

Where,

1st parameter- it is the handle of the thread, whose priority we want to change and

2nd parameter- it is one of the above 7 values.

On success, this function returns TRUE or non zero while **on failure** returns FALSE or Zero.

Unlike priority class, which is which is suggested to be restricted area, thread priority is not! Means we can freely experiment with the thread priority.

After observing the multithreading concepts and some thread related APIs now we will see a simpe multithreading example, first with win3 APIs likeCreateThread( ) and then with standard C library function _beginthread(.)

A simple win32 API library based multithreaded program-> create a new "win32 application" project and give it a name WithWin32. then 3 files in it as … WithWin32.c, WithWin32.rc & WithWin32.h.

WithWin32.c:

```
#include<windows.h>
#include"WithWin32"
// global function prototypes
LRESULT CALLBACK
WndProc(HWND,UINT,WPARAM,LPARAM);
DWORD WINAPI MyThreadProcOne(LPVOID);
DWORD WINAPI MyThreadProcTwo(LPVOID);
//WinMain()
int WINAPI WinMain(…….)
{     …….
      …….
      As it is
      …….
```

```
                for(I=0;I<= 32767;I++)
                {
                        Wsprintf(str,"Thread1->
increasing order output=%d",I);

TextOut(hdc,5,5,str,lstrlen(str));
                }
                ReleaseDC((HWND)param,hdc);
                Return(0);
}
DWORD WINAPI MyThreadProcTwo(LPVOID param)
{
              // local varialble declarations
                HDC hdc;
                Int I;
                TCHAR str[255];
                // code
                hdc=GetDC((HWND)param);
                for(I=32767;I>=0;I--)
                {
                        Wsprintf(str,"Thread2->
decrementing order output=%d",I);

TextOut(hdc,5,20,str,lstrlen(str));
                }
                ReleaseDC((HWND)param,hdc);
                Return(0);
}
```

WithWin32.rc:

```
#include<windows.h>
#include"WithWin32.h"
//menu
MyMenu MENU
BEGIN
                MENUITEM "&About",IDM_ABOUT
END
```

WithWin32.h:

```
#define IDM_ABOUT 100
```

➔ when above program is compiled, rebuild & executed, we get a window on screen with rapidly changing 2 text lines, one gives increasing order output message of thread 1 with changing values from 0 to 32767. the other line

```
        Wndclass.lpszMenuName="MyMenu";
        ……
        ……
}
LPRESULT CALLBACK WndProc(……)
{
        // local variable declarations
        HANDLE hThread1,hThread2; // thread
handles
        DWORD dwID1, dwID2; // thread Ids
        // code
        switch(iMsg)
        {
         case WM_CREATE:
                    hThread1=CreateThread(NULL, 0,
(LPHREAD_START_ROUTINE) MyThreadProcOne, (LPVOID)
hwnd, 0, &dwID1);
                    hThread2=CreateThread(NULL, 0,
(LPHREAD_START_ROUTINE) MyThreadProcTwo, (LPVOID)
hwnd, 0, &dwID2);
break;

          case WM_COMMAND:
                    switch(LOWORD(wParam))
                    {
                     case IDM_ABOUT:
                                MessageBox(hwnd, "This
is a win32 API library based Multithreading
program","about",MB_OK);
                                Break;
                     }
                     break;
          case WM_DESTROY:
                     as it is
                      ……
                    break;
          }
          return(DefWindowProc(…….));
}
DWORD WINAPI MyThreadProcOne(LPVOID param)
{
          // local varialble declarations
           HDC hdc;
           Int I;
           TCHAR str[255];
           // code
           hdc=GetDC((HWND)param);
```

gives decreasing order output message of thread 2 with changing values from 32767 to 0.

The window also shows a menu bar with only one menu item "About" in it. If this menu-item is clicked a message box gets displayed.

Note-

1. the two threads run concurrently.
2. even message box is displayed, threads go on running in background.
3. this shows multithreading clearly.

➔ some points from above program:

1. the ThreadProc( ) functions must be declared globally as os & program both use them.
2. CreateThread( ) function is called with 1st parameter NULL, second 0, third, the ThreadProc( ) name casted to (LPTHREAD_START_ROUTINE)<function name>, 4th parameter is the data passed from parent thread to child thread. We here pass handle of the window hwnd by casting with (LPVOID) . this allows Threadproc functions to create hdc from this handle, 5th parameter is 0 and sixth parameter is the address of thread ID variable(declared locally). When these two calls to CreateThread( ) execute successfully, they return thread handles hThread1 & hThread2 respectively.
3. in Threadproc( ), we use the passed window handle by recasting back to (HWND) type for creating & releasing hdc.
4. then we use "for loop" to output incrementing and decrementing order integers.
5. the interesting point is about TCHAR, wsprintf & lstrlen( ). As we have seen the memory leackage problem, if we use sprintf( ) or strlen( ), then memory leak may occure. Hence we used wsprintf( ) & lstrlen( )- new win 32 supporting C library functions. The parameter of wsprintf( ) is LPCTSTR and not char *, hence our str variable can not be char str[255]. It should be of wide character type. Hence we used MACRO TCHAR (defined in windows.h) type for str variable.
6. as the standard prototype of ThreadProc( ) function is ….
DWORD ThreadProc(LPVOID);, both our ThreadProv return 0 as DWORD type value.
7. in calls to TextOut( ) we changed y co-ordinate from 5 to 20. this avoids overlapping of outputs of to threads in the window.

A simple standard C library based multithreading program:

The program is mostly same as the previous program. Just we call our project as- WithStdLib and our 3 files will be WithStdLib.c, WithStdLib.rc & WithStdLib.h.

*** changes in WithStdLib.c with respect to WithWin32.c**

1. change the included heard file name WithWin32,h to WithStdLib.h.

2. also include stdio.h (for sprintf( )) and then process.h (for _beginthread( )).
3. keep ThreadProc function names as they are. Just change their return type from DWORD to VOID _ _cdecl and then change their parameters from LPVOID to void *.
4. in WM_CREATE of Wndproc( ) now declare z variable ulThread1 & ulThread2 of unsigned long type. Cut the lines of HANDLE & DWORD declarations.
5. call _beginthread( ) as…..

```
ulThread1=_beginthread(MyThreadProcOne,0, (void *)hwnd);
ulThread2=_beginthread(MyThreadProcTwo,0, (void *)hwnd);
```

6. while writing the function declarators of threadproc, make appropriate changes with respect to step 3.
7. now you can use char in place of TCHAR. Also you can use sprintf( ) & strlen( ) as usual.
8. there will be no return statement in threadproc as they are void type.

*changes in WithStdLib.rc with respect to WithWin32.rc:
just change the included header file's name from WithWin32.h to WithStdlib.h.

*changes in WithStdLib.h with respect to WithWin32.h:
no changes.

➔ after saving above 3 files , now if you compile the project, you will get the warning of _beginthread funcion. This is because our program is multithreded and the library of comilation is single threaded hence do the necessary changes in project's setting as explained before.

➔ after making desired changes, if you compile rebuilt & execute the program , you will get the same output as explained before.

➢ THREAD SYNCHRONIZATION:

As stated before multithreading may cause problems. Consider following cases…..
1. when two or more threads need to access to a shared resources at the same time and the rule is suppose that only one thread can access it at one time.
2. if one thread is writing to a file and at the same time another thread wants to write on it.
3. if one thread is waiting for an event to occur (means it is suspended) and that event is going to be caused by some another thread.
4. if one thread is initializing a data structure (suppose a linked list) and if suppose another thread accesses the same data structure while its initialization is not yet completed.

All above situaltion need to be handled very carefully. Otherwise there may be inconsistent data or program or even os may crash.

To solve above situation , win32 multithreading subsystem gives a powerful object called as "synchronization". Win9x & NT supports 5 types of synchronization objects....

1. classic semaphore
2. mutex semaphore
3. critical section
4. event object & event signaling &
5. waitable timer.

1. classic semaphore➔

also called as "standard semaphore" semaphore synchronization access to a resource. Means the semaphore can be used to allow, a limited number of processes or threads, to access a resource. Semaphores are implemented by using counter. The counter is incremented when a process or thread releases the semaphore and the counter is decremented when a process or thread completes it's waiting time and ready to access the resource.

The win32 API to create the semaphore is Createsemaphore( ) whose prototype is.....

HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES, LONG, LONG, LPSTR);

Where,

1st parameter- it has same meaning as explained in CreateThread( ).

2nd parameter- it is the initial count which must be greater than or equal to zero and must be less than or equal to the third parameter.

3rd parameter- it is the maximum count which must be greater than the initial count.

4th parameter- it is the name of semaphore object as you desire. If it is NULL, the semaphore is created without name.

On success function returns handle to the newly created semaphore. On failure it returns NULL.

As this function requires "WAITING" of some thread, another function is used to allow waiting of a thread for a semaphore. Such function is WaitForSingleObject( ) whose prototype is....

DWORD WaitForSingleObject(HANDLE<DWORD);

Where,

1st parameter- it is the handle of object to wait for (means it is used with semaphore, thee first parameter will be handle of semaphore)

2nd parameter- it is time interval in milliseconds.

This function returns WAIT_OBJECT_O if the state of the object is signaled or returns WAIT_TIMEOUT if the time interval elapses. If we want to wait for infinite time, the second parameter can be set to INFINITE constant. In such cases function will return only on WAIT_OBJECT_O and not on WAIT_TIMEOUT as the time interval is infinite.

To release a semaphore the win32 API is Releasesemaphore( ) whose prototype is…..

BOOL ReleaseSemaphore(HANDLE,LONG,LPLONG);

Where,

1st parameter- handle of the created semaphore

2nd parameter- it is the amount by which the semaphore's current count is to be incremented. This must be greater than 0.

3rd parameter- it is a counter which receives the previous count of the semaphore object. If it is not requires, you can give NULL.

This function returns non zero on success and 0 on failure.

Besides above 2 semaphore functions, there is yet another semaphore function called as Opensemaphore( ), which is used to get the handle of an existing semaphore. The prototype is….

HANDLE OpenSemaphore(DWORD, BOOL, LPCTSTR);

This function is not commonly used.

2. Mutex→

the word is created by combination of two words " mutual exclusion". As it's name suggests, mutual exclusion between processes and threads is done when they try to access same resource at same time. In other words, mutex allows only one thread to own it. Means to prevent 2 or more threads to write to a file at a time, each thread owns a mutex then writes to a file (during this time another thread is waiting for the mutex to get free so that it can be owned) and then releases the mutex.

To create a mutex object, win32 API is CreateMutex( ), whose prototype is….

HANDLE CreateMutex(LPSECURITY_ATTRIBUTES,BOOL,LPCTSTR);

Where,

1st parameter- same as before

2nd parameter- flag of initial ownership.

3rd parameter- name of mutex object.

And it returns the handle of the newly created mutex.

To release a mutex object, win32 API is Releasemutex( ), whos prototype is…

BOOL ReleaseMutex(HANDLE);

The parameter is handle of mutex which is to be released. On success, function returns non zero and zero on failure.

The OpenMutex( ) API is used to open an existing mutex object whose prototype is same as OpenSemaphore( ). Less commonly used.

Note that- you can use CreateSemaphore( ) function to create mutex like object by specifying initial & max count both as 1. so only one thread can access it.a mutex can be used for multiple processes too.

3. Critical Section →

consider the situation, if second thread tries to accesss a data which is supposed to be initialized by thread1. if initialization by thread 1 is not yet

completed and if thread 2 access the data, then it is sure that thread 2 is going to have un-initialized, inconsistsant data which may crash a program or even can crash os.

Such a section of code is called as critical section. And is more common in case of data structures like liked list implemented in a multithreading program.

*** Steps to use the critical section :**

1. declare a global variable of type CRITICAL_SECTION. Actually CRITICAL_SECTION is a structure but members are only os accessible and hence not shown or not used. They only reserved. Eg. CRITICAL_SECTION cs;
2. the thread which wants to initialise link list like critical, pointer oriented data structure (and obviously needed by another thread), must then initialise the critical section by InitilizeCriticalSection( ) API whose prototype is…
VOID InitializeCriticalSection(LPCRITICAL_SECTION);
This can be used as …..eg. InitializeCriticalSection(&cs);
3. note that you have no concerned with the critical section. MSDN says that "do as given. Even don't look at it. Os will take care of it".
4. now the thread owns the critical section. Further your thread, which initializes the critical section, must enter into it by using EnterCriticalSection( ) API whose prototype is same as InitializeCriticalSection( ).
Eg. EnterCriticalSection(&cs);

Now the ownership of the therad for the critical section is concreted. Thus no other thread can access the code written following bove section. EnterCriticalSection( ) in it (as cs is global other threads can access it), that thread is suspended automatically by the os, and will be allowed to acess the critical section only when first thread leaves the critical section by using LeaveCriticalSection( ) whose prototype is same as InitializeCriticalSection( ).
Eg. LeaveCriticalSection(&cs);

*** some points about critical setion:**

a. critical section is also of "mutual exclusion" type object.
b. only one thread can own a critical section at any time.
c. you can define multiple critical sections in one program for different data structures and for different threads.
d. don't use critical section in primary thread, because if secondary thread also has it's own critical section and spends lot of time dealing with it. Then primary thread may hang
e. critical sections can be used in single process only. Two different processes can not manipulate same critical section.

4. Event object and event signaling →

if one thread or an process is trying to send an event or message to another thread then Event object or event signalling is used. So these two

methods are notification of onemthread to another either by generating an event or by sending a message.

An event object is created by CreateEvent( ) API, whose prototype is ….
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES,BOOL, BOOL, LPSTR);
Where,
1st parameter- same as before.
2nd parameter- if it is TRUE then manual reset event object is created,if it is FALSE then autoreset event object is created.
3rd parameter- if it is TRUE initial state is signaled or else if FALSE, the initial state remain unsignaled.
4th paarmeter- it is the name of the event object if NULL, then event object is created without a name.

--We than can use WaitForSignalObject( ) to wait.
--To set the event object SetEvent( ) API is used. Prototype is…
BOOL SetEvent(HANDLE);
--to manually reset the event object ResetEvent( ) is used. Prototype is same as SetEvent( ).
--to open an existing event object OpenEvent( ) is used. Whose prototype is same as OpenSemaphore( ) and OpenMutex( ).

5. Waitable Timer ➔.

They are much like usualtimer objects but more flexible and convinient. They make easier the automation of background tasks.

This is the resent addition of windows os. Neither windows 3.1 nor windows 95 supports it. But they are added in withdows NT4.
Common APIs for waitable timers are…
1. CreateWaitableTimer( )
2. SetWaitableTimer( )
3. OpenWaitableTimer( )
4. CancleWaitableTimer( )
* SetWaitableTimer( ) uses a callback function called as TimerProc( ) or TimerFunc( ) [just a place holder].
* you can use MessageBeep( ) function to indicate the completion of time to the user.

➢ **Thread LocalStorage**

Global and local static variables are sharable to all threads in a program, local auto varianles on other side are local to respective thread and hence not sharable.

But suppose we want allthreads of a program to share a data which is local and not global or not static, the concept of thread local storage is used.

Win32 API has 4 functions for thread local storage.

1. declare a global structure which has your data.
2. primary thread will call TlsAlloc( ) to get index to it. Declare this index globally too.
3. then any thread which wants to set value of this data will allocate memory to GlobalAlloc( ) and will use above index and this memory in TlsSetValue( ).
4. now any other thread can get pointer to this memory (having data) will call TlsGetValue( ) to get the pointer to memory block. And then will use the data as necessary.
5. thread will free the memory GlobalFree( ) & finally main thread will free by TlsFree( ).s

Dynamic data Exchange

DDE (Dynamic Data Exchange)

Introduction

DDE is one of the method IPC (Inter process communication) it is less ambitious but it provided a gateway for great OLE/COM. DDE is based on windows messaging system in which two programs can talk with each other with the help of DDE in this context the program which needs some help from the other program is called as **client** and the program which provides this help to client is called as **server**

Programming in DDE can be done in 2 ways

1) By using bare Windows messages.
2) By using the library of these bare window messages . This library is called as DDEML(Dynamic Data Exchange Management Library) .

The aim of this DDEML is to make DDE programming easier than the first way.
Though DDE is now supposed to be outdated it is the Mother of the OLE & COM & hence to understand OLE COM it is better to study the ideas of DDE . It is not at all necessary that one client has one server and vice versa . Multiple clients may have one server and many servers can serve for one client.

Programmatically speaking "Client server talk" is called as **Conversation** . These two different programs communicate with each other by sending and receiving data, means server makes some calculations and keep the answer data at some global memory location and then tells client that it has kept the required data(may be in some header file along with application, topic and I item information's).

Now client picks up that answer data and tells the server that it has Received the data. For this conversation to work properly it is the duty of the server to assign the required amount of memory. Note that after reception of data Client does not destroy this memory allocated by the server rather in the context of the client this memory is "read only" and it just tells server that it received the data so that the server can deallocate the memory for current conversation . So in one line we can say that

"usually server only writes the data and client only reads the data"

BASIC CONCEPTS

When client requests a server for data the server must be able to identify the type of data that client wants.

To fulfill above purpose conventionally DDE decided to tackle data in the form of 3 strings (as character takes memory of only 1 byte & this is same throughout all machines ,thus for uniformity string is used .Integer is also send as string but is picked up at the other end as integer) which are called as

1) Application.
2) Topic.
3) Item.

1) **Application:** Name of the program .Usually server sends this to client.

2) **Topic:** This is the explanation of "what the data is about" .

3) **Item:** Which is the actual Data .

Note that Every server has only one Name it may have one or many topic but it must have atleast one topic and every topic must have atleast one item.

LOCATION OF THE SERVER

"Why DDE is less Ambitious " ?.
The server programs exe must be located in the clients exes directory or in that directory which is in the path variable.

Types of DDE conversations

There are 3 basic types of DDE conversations
1) Cold link.
2) Hot link.
3) Warm link.
These architectures of DDE are defined in dde.h header file which must be included in every DDE program.

1)**Cold link:**

This is the simplest conversation of all the 3.It begins when client broadcasts WM_DDE_INITIATE message. For every communication DDE uses a child window (i.e. child window control & not parent).In this message wparam is the senders

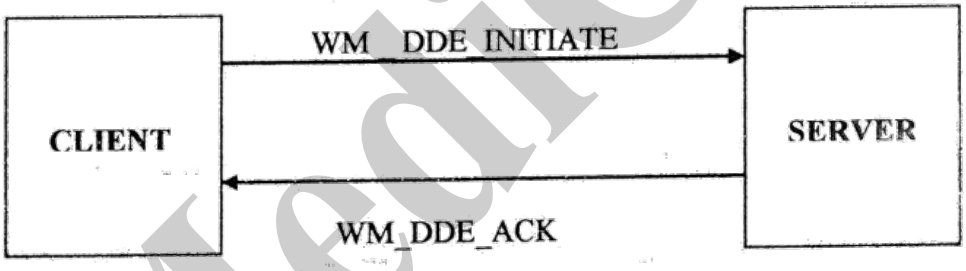
window handle (clients) and lparam is packed structure of application, topic and item

Note that

a) Client can send this message either to a specific server that it knows or
b) Client can send this message to all currently running applications on the Desktop and the one which is capable of responding to this becomes server of the client. Obviously here application name will be NULL(only in case b) or
c) In b we assumed that client needs a specific information (a specific topic) from any server which can respond to this .But in c there may be such clients which want to talk with any server and/or about any topic.

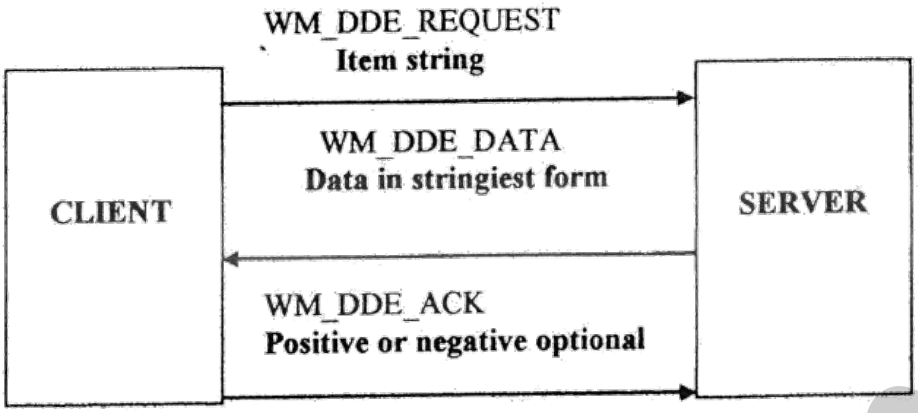
After broadcasting above message i.e. WM_DDE_INITIATE the server which supports the respective topic and responds the client by sending WM_DDE_ACK (ACK for Acknowledgement) message.

Initialization



After receiving WM_DDE_ACK message from the server client now requests the server about particular data (item) by sending WM_DDE_REQUEST . If the respective server is capable of giving this data it sends back WM_DDE_DATA along with the data. Obviously wparam of this is servers window handle and lparam is the actual data. When client receives the data through WM_DDE_DATA , client sends WM_DDE_ACK with positive response & if it doesn't receive data then it sends the same message but now with negative response. When conversation completes both send WM_DDE_TERMINATE to each other terminating the current DDE session . Usually WM_DDE_TERMINATE from the server is not as imp as from the client .The imp thing is that both must respond each other correctly to this WM_DDE_TERMINATE message by sending the same message.

Data Communication



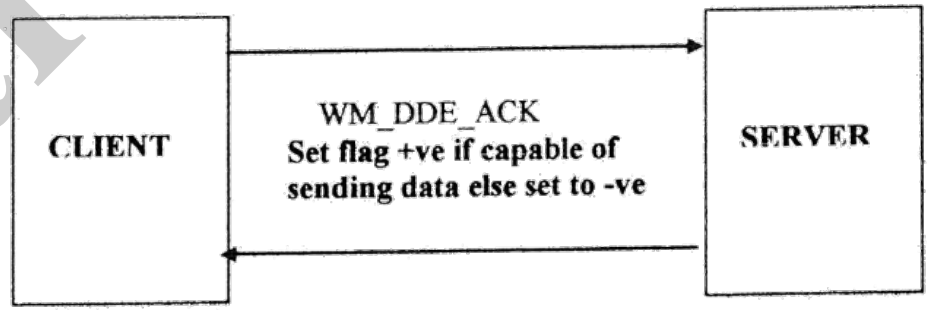
2) Hot link:

Cold link method is good for many cases but sometimes Data is of changing nature then it may happen that the rate of change in data is so fast that before completion of cold link conversation data might have been changed. For example census .Though server now knows that data is changed it can't send it to the client because client doesn't ask for it. So client has to be acquainted with such continuously changing data.

For example census is a continuously changing data.

Here again conversation starts as usual by WM_DDE_INITIATE and WM_DDE_ACK (see in cold link).

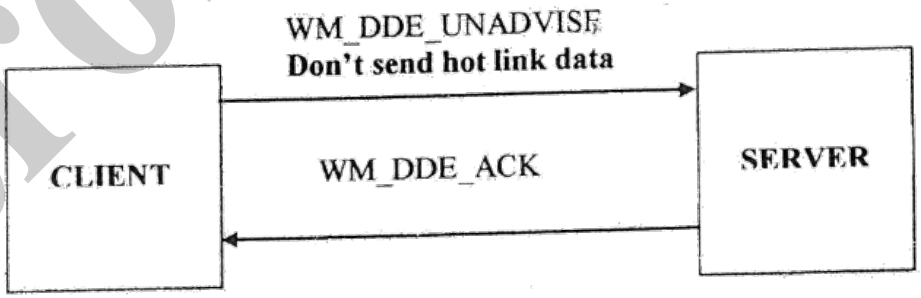
But the difference is that here instead of sending WM_DDE_REQUEST (which is specific for cold link) client sends WM_DDE_ADVISE. When server receives this message it replies by WM_DDE_ACK. Now the question is that whether server is capable of sending the requested data or not. For this purpose there is flag in WM_DDE_ACK. If server is capable of sending the requested data it sets this flag to positive (it is an integer flag) or else set this flag to negative.



This WM_DDE_ADVICE message forces the server to send the data periodically as soon as change in the data occurs . For example census.
Though method of sending data requested by client is different in cold link and hot link. The method of sending actual data is the same. Means here to the actual data sending message is WM_DDE_DATA through which server sends most recent data. After reception of the data client may or may not (optional) send acknowledgement WM_DDE_ACK.
The figure of WM_DDE_DATA and WM_DDE_ACK is same as in cold link. As like WM_DDE_REQUEST WM_DDE_ADVICE also has handle to window as wparam and packed data as lparam. When client or server are done with the job both can terminate DDE as usual as in cold link by WM_DDE_TERMINATE.

Note :
When the server is capable of both hot link DDE and cold link DDE then client can request for both type of data ,means can request one type of data with WM_DDE_ADVISE and then also can request other type of data by WM_DDE_REQUEST .
In such cases the duty of the client is to

1) Close the current session by WM_DDE_TERMINATE and then start the other session .But if the client and server are in hotlink session then before sending WM_DDE_TERMINATE client should send WM_DDE_UNADVISE (which is nothing but to say that "I don't want this type of data now onwards") message to start the next session of cold link with the same server.



wparam of WM_DDE_TERMINATE is handle of window. But lparam is 0 that is why the session is terminated.

2) **Warm link:**

It is combination of both hot and cold links. In hot link the basic idea is that server must send the requested data as soon as changes in data occurs, but in warm link client sets a flag in WM_DDE_ADVISE telling the server that don't send the data immediately when change in data occurs .Rather first notify me that change in data has occurred & then if I say send it .When server looks at this flag then server sends WM_DDE_DATA message . **But here data item is set to NULL** .When client looks at this NULL it knows that the data has been changed . Now it is up to the client whether to accept the data or not. If it wants the data it will send WM_DDE_REQUEST to get the changed data. As a response to WM_DDE_REQUEST server sends WM_DDE_DATA with actual data (Why client sends WM_DDE_REQUEST instead of WM_DDE_ADVISE again because changes have already been done on the data which is static & hence there is no need to dynamically change the data) If client doesn't needs the data it sends WM_DDE_UNADVISE & waits for next change in session and when the session completes both send WM_DDE_TERMINATE as usual.

Note:

Besides these 3 basic conversations there are 2 more conversations but they are very uncommon(see about them in MSDN)

1) WM_DDE_EXECUTE
2) WM_DDE_POKE

Data structures used in DDE

We know that large amount of data can be sent along with the message in packed form either in wparam and mostly in lparam e.g. LPCREATESTRUCT in WM_CREATE as its lparam. The same idea is used by DDE messages too. There are such 4 packed data structures which can be send as lparam

These 4 are

1) DDE_ACK
2) DDE_ADVISE
3) DDE_DATA
4) DDE_POKE

Out of these 1st 3 are important . All these are defined in DDE.H file ,therefore in every DDE program this file "DDE.H" must be included.

1) DDE_ACK

(This structure is used along with WM_DDE_ACK message)

```
typedef struct {
                UNSIGNED SHORT  bAppReturnCode;
                UNSIGNED SHORT reserved;
                UNSIGNED SHORT  fBusy;
                UNSIGNED SHORT  fAck }DDE_ACK;
```

bAppReturnCode : This is Application defined return code.

reserved : As stated it must be set to zero.

fBusy : This flag indicates whether the application which is sending WM_DDE_ACK is busy or not means a non zero value indicates that application is busy and thus unable to respond . When it is 0 it indicates that application is not busy & thus can respond.

fAck : This indicates whether the application has received the WM_DDE_ACK message or not. Non zero means received 0 means not received.

2) DDE_ADVISE :

This structure is used along with WM_DDE_ADVISE and WM_DDE_UNADVISE message.

```
typedef struct {
                UNSIGNED SHORT reserved;
                UNSIGNED SHORT fDeferUpd;
                UNSIGNED SHORT fAckReq;
                SHORT cfFormat } DDE_ADVISE;
```

reserved : As stated must be set to 0.

fDeferUpd : If this is non zero then the server which is receiving WM_DDE_ADVISE message must respond to the client by sending WM_DDE_DATA message along with NULL. Hence this method is used in warm link.

fAckReq : This flag indicates whether server should set fAckReq flag in WM_DDE_DATA (DDE_DATA structure also contains the same flag) or not . If non zero then server

must set fAckReq in DDE_DATA.

cfFormat : This specifies format of the client Requested data . The formats are same as clipboard formats means if the data is textual then this flag is CF_TEXT. If data is bitmap this flag is

1) CF_BITMAP for DDB.
2) CF_DIB for DIB.
3) If the data is metafile then CF_ENHMETAFILE (ENH for enhanced).
4) If the data is sound wave then CF_WAVE.
5) If the data is in Unicode form then CF_TIFF

3) DDE_DATA : This structure is used along with WM_DDE_ADVISE , WM_DDE_UNADVISE as well as WM_DDE_REQUEST when the server sends WM_DDE_DATA . So this is the core structure for DATA transmission.

```
typedef  struct {
                    UNSIGNED SHORT unUsed;
                    UNSIGNED SHORT fResponse;
                    UNSIGNED SHORT fRelease;
                    UNSIGNED SHORT fAckReq;
                                   SHORT cfFormat
                    BYTE value [1] }DDE_DATA;
```

1) unUsed : Not used.
2) fResponse : This flag indicates whether the data is going to be sent as a response to WM_DDE_REQUEST or WM_DDE_ADVISE. If 0 then WM_DDE_REQUEST .
3) fRelease : This is used along with DDE_POKE indicating memory release flag . For other application messages this flag is set to 0.
4) reserved : reserved set to 0.
5) fAckReq : Complementary meaning as that of fAckReq of DDE_ADVISE.
6) cfFormat : Same as before.
7) value : this is the actual data and depends upon cfFormat.

DDE_ColdClient.C

```
// headers
#include<windows.h>
#include<stdio.h>
#include<process.h>// for _spawnlp()
#include<dde.h>// for DDE
// macró
#define WM_USER_DDE_INITIATE (WM_USER+1)
// global function declarations
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
// global variable declarations
char AppName[]="DDE";
// WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                           LPSTR lpCmdLine,int nCmdShow)
{
      // variable declarations
      WNDCLASSEX wndclass;
      HWND hwnd;
    MSG msg;
      // code
      // main window class
      wndclass.cbSize=sizeof(wndclass);
      wndclass.style=CS_HREDRAW|CS_VREDRAW;
      wndclass.cbClsExtra=0;
      wndclass.cbWndExtra=0;
      wndclass.lpfnWndProc=WndProc;
      wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
      wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
      wndclass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
      wndclass.hInstance=hInstance;
      wndclass.lpszClassName=AppName;
      wndclass.lpszMenuName=NULL;
      wndclass.hIconSm=LoadIcon(NULL,IDI_APPLICATION);
    // register main window class
      RegisterClassEx(&wndclass);
      hwnd=CreateWindow(AppName,
                        "DDE Client - Cold Link",
                              WS_OVERLAPPEDWINDOW,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                              CW_USEDEFAULT,
                  NULL,
                  NULL,
                              hInstance,
                              NULL);
      ShowWindow(hwnd,nCmdShow);
      UpdateWindow(hwnd);
      // send user defined message to WndProc()
      SendMessage(hwnd,WM_USER_DDE_INITIATE,0,0L);
      // message loop
      while(GetMessage(&msg,NULL,0,0))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      return(msg.wParam);
```

(2)

```
}
// Main Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
{
      // function declarations
    void leadtrim(char[]);
    void trailtrim(char[]);
      // variable declarations
      ATOM aApplication,aTopic,aActualDataItem;
      static HWND hwndServer=NULL;
      UINT uiLow,uiHi;
      GLOBALHANDLE hDdeData;
      DDEDATA *pDdeData;
      DDEACK DdeAck;
      WORD wStatus;
      static char ServerAppName[]="DDE";
      static char szTopicName[]="SQUARE";
      int result,num,NegativeStatusFlag;
      long square;
      char DataString[255],AnswerString[255];
      // code
      switch(iMsg)
      {
      case WM_USER_DDE_INITIATE:
            // add application class name's string to global atom table &
            // return its unique id
            aApplication=GlobalAddAtom(ServerAppName);
            // add topic name's string to global atom table & return its
            // unique id
            aTopic=GlobalAddAtom(szTopicName);
            // send WM_DDE_INITIATE to all windows running on desktop
            SendMessage(HWND_BROADCAST,
                        WM_DDE_INITIATE,
                            (WPARAM)hwnd,
                            MAKELONG(aApplication,aTopic));
            // if no response then try by executing the server
            if(hwndServer==NULL)
            {
                  _spawnlp(_P_NOWAIT,"DDE_ColdServer.exe",
                           "DDE_ColdServer.exe",
                           NULL);
                  // send WM_DDE_INITIATE again to all windows running on
desktop
            SendMessage(HWND_BROADCAST,
                        WM_DDE_INITIATE,
                            (WPARAM)hwnd,
                            MAKELONG(aApplication,aTopic));
            }
            // delete the atoms
          GlobalDeleteAtom(aApplication);
          GlobalDeleteAtom(aTopic);
            // if still no response then display message box
            if(hwndServer==NULL)
            {
                  MessageBox(hwnd,"Server Can Not Be Started","Error",MB_OK);
                  break;
            }
```

```
            // if OK,then now post WM_DDE_REQUEST message to server
            num=25;
            sprintf(DataString,"%d",num);
            aActualDataItem=GlobalAddAtom(DataString);
            result=PostMessage(hwndServer,
                               WM_DDE_REQUEST,
                                  (WPARAM)hwnd,
                                  MAKELPARAM(CF_TEXT,aActualDataItem));
            if(result==0)
                  GlobalDeleteAtom(aActualDataItem);
            break;
      case WM_DDE_ACK:
            UnpackDDElParam(WM_DDE_ACK,lParam,&uiLow,&uiHi);
            FreeDDElParam(WM_DDE_ACK,lParam);
            hwndServer=(HWND)wParam;
            // get error message flag if any
            DdeAck=*((DDEACK *) &uiLow);
            NegativeStatusFlag=DdeAck.bAppReturnCode;
            if(NegativeStatusFlag==1 || NegativeStatusFlag==2 ||
NegativeStatusFlag==3)
                  MessageBox(hwnd,"The Send Data Was .....\nEither In Wrong
Format\nOr Beyond Limit\nOr Non-Integer\nOr Negative","Server Sent
Error",MB_ICONERROR|MB_OK);
            GlobalDeleteAtom((ATOM)uiLow);
            GlobalDeleteAtom((ATOM)uiHi);
            break;
      case WM_DDE_DATA:
            // receive the data
            UnpackDDElParam(WM_DDE_DATA,lParam,&uiLow,&uiHi);
            FreeDDElParam(WM_DDE_DATA,lParam);
            hDdeData=(GLOBALHANDLE)uiLow;
            pDdeData=(DDEDATA *)GlobalLock(hDdeData);
            aActualDataItem=(ATOM)uiHi;
            // check for invalid format
            if(pDdeData->cfFormat==CF_TEXT)
            {
                  // convert returned sended data item
                  GlobalGetAtomName(aActualDataItem,DataString,255);
                  // trim the received string
                  leadtrim(DataString);
                  trailtrim(DataString);
                  num=atoi(DataString);
                  // convert returned string to data
                  square=atol(pDdeData->Value);
                  sprintf(AnswerString,"Square Of %d = %ld",num,square);
                  MessageBox(hwnd,AnswerString,"Square Of Given Number",MB_OK);
            }
            // if acknowledgement is needed to be sent
            if(pDdeData->fAckReq==TRUE)
            {
                  DdeAck.bAppReturnCode=0;
                  DdeAck.fAck=FALSE;
                  DdeAck.fBusy=FALSE;
                  wStatus=*((WORD *) &DdeAck);
                  result=PostMessage((HWND)wParam,
                                     WM_DDE_ACK,
                                        (WPARAM)hwnd,
```

(4)

```
PackDDElParam(WM_DDE_ACK,wStatus,aActualDataItem));
                    if(result==0)
                    {
                          GlobalDeleteAtom(aActualDataItem);
                          GlobalUnlock(hDdeData);
                          GlobalFree(hDdeData);
                    }
              }
              else
              {
                    GlobalDeleteAtom(aActualDataItem);
              }
              if(pDdeData->fRelease==TRUE || DdeAck.fAck==FALSE)
              {
                    GlobalUnlock(hDdeData);
                    GlobalFree(hDdeData);
              }
              else
              {
                    GlobalUnlock(hDdeData);
              }
              break;
        case WM_DDE_TERMINATE:
              // if this message is received,you should also send the same
              PostMessage((HWND)wParam,WM_DDE_TERMINATE,(WPARAM)hwnd,0L);
              hwndServer=NULL;
              break;
        case WM_CLOSE:
              if(hwndServer==NULL)
                    break;
              PostMessage(hwndServer,WM_DDE_TERMINATE,(WPARAM)hwnd,0L);
              break;
        case WM_DESTROY:
              PostQuitMessage(0);
              break;
        }
        return(DefWindowProc(hwnd,iMsg,wParam,lParam));
}
void leadtrim(char s[])
{
    // variable declarations
    int i,j;
      char t[255];
    // code
    for(i=0;i<=(int)strlen(s)-1;i++)
    {
     if(s[i]!=' ')
          break;
    }
    for(j=0;i<=(int)strlen(s)-1;i++,j++)
    {
       t[j]=s[i];
    }
    t[j]='\0';
      strcpy(s,t);
}
```

```
void trailtrim(char s[])
{
    // variable declarations
    int i;
    // code
    for(i=strlen(s)-1;i>=0;i--)
    {
       if(s[i]!=' ')
             break;
    }
    s[i+1]='\0';
}
```

DDE_ColdServer.c

```
// headers
#include<windows.h>
#include<stdio.h>
#include<dde.h>// for DDE
// macro
// global function declarations
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
LRESULT CALLBACK ServerWndProc(HWND,UINT,WPARAM,LPARAM);
BOOL CALLBACK MyEnumChildProc(HWND,LPARAM);
// global variable declarations
HINSTANCE hInst;
MSG msg;
char MainAppName[]="DDE";
char ServerAppName[]="SERVER";
// WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                           LPSTR lpCmdLine,int nCmdShow)
{
      // variable declarations
      WNDCLASSEX wndclass;
      HWND hwnd;
      // code
      // main window class
      wndclass.cbSize=sizeof(wndclass);
      wndclass.style=CS_HREDRAW|CS_VREDRAW;
      wndclass.cbClsExtra=0;
      wndclass.cbWndExtra=0;
      wndclass.lpfnWndProc=WndProc;
      wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
      wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
      wndclass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
      wndclass.hInstance=hInstance;
      wndclass.lpszClassName=MainAppName;
      wndclass.lpszMenuName=NULL;
      wndclass.hIconSm=LoadIcon(NULL,IDI_APPLICATION);
    // register main window class
      RegisterClassEx(&wndclass);
      // server window class
      wndclass.cbSize=sizeof(wndclass);
      wndclass.style=0;// style is not necessary
      wndclass.cbClsExtra=0;
      wndclass.cbWndExtra=0;
      wndclass.lpfnWndProc=ServerWndProc;
      wndclass.hIcon=NULL;// icon is not necessary
      wndclass.hCursor=NULL;// cursor is not necessary
      wndclass.hbrBackground=NULL;// background color is not necessary
      wndclass.hInstance=hInstance;
      wndclass.lpszClassName=ServerAppName;
      wndclass.lpszMenuName=NULL;
      wndclass.hIconSm=NULL;// iconSm is not necessary
    // register main window class
      RegisterClassEx(&wndclass);
      // here create main window only
      hwnd=CreateWindow(MainAppName,
                        "DDE Server - Cold Link",
                              WS_OVERLAPPEDWINDOW,
                              CW_USEDEFAULT,
```

```
                                        CW_USEDEFAULT,
                                        CW_USEDEFAULT,
                                        CW_USEDEFAULT,
                                NULL,
                                NULL,
                                        hInstance,
                                        NULL);
        // global instance handle
        hInst=hInstance;
        // proceed as usual
        ShowWindow(hwnd,SW_SHOWMINNOACTIVE);
        UpdateWindow(hwnd);
        // message loop
        while(GetMessage(&msg,NULL,0,0))
        {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
        }
        return(msg.wParam);
}
// Main Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
{
        // variable declarations
        HWND hwndServer;
        ATOM aApplication,aTopic;
        char szTopicName[]="SQUARE";
        // code
        switch(iMsg)
        {
        case WM_DDE_INITIATE:
              // add application class name's string to global atom table &
              // return its unique id
              aApplication=GlobalAddAtom(MainAppName);
              // add topic name's string to global atom table & return its
              // unique id
              aTopic=GlobalAddAtom(szTopicName);
              // check server's atoms with client specified atoms,
              // if matching create server's window (hidden) & send acknowledge
              // message to client
              if((LOWORD(lParam)==0 || LOWORD(lParam)==aApplication) &&
             (HIWORD(lParam)==0 || HIWORD(lParam)==aTopic))
              {
                    hwndServer=CreateWindow(ServerAppName,
                                            NULL,// title not necessary
                                                    WS_CHILD,// only child
style
                                                    0,// x co-ordinate is not
necessary
                                                    0,// y co-ordinate is not
necessary
                                                    0,// width is not
necessary
                                                    0,// height is not
necessary
                                                    hwnd,// parent window
handle
```

```
                                                    NULL,
                                                    hInst,
                                                    NULL);
                        // this is very important step.....
                        // sending server's window handle to client as 3 rd parameter
                        // ensures that now onwords whichever DDE message sent by the
                        // client,will be bypassed to server's window and not to main
                        // window ( because as a rule client will send DDE messages
                        // to that window which will send its own handle as wParam of
                        // WM_DDE_ACK message
                        SendMessage((HWND)wParam,// client's window handle
                                    WM_DDE_ACK,
                                        (WPARAM)hwndServer,
                                        MAKELPARAM(aApplication,aTopic));
                }
                else// if above fails,then no need of newly created atoms
                {
                        GlobalDeleteAtom(aApplication);
                        GlobalDeleteAtom(aTopic);
                }
                break;
        case WM_CLOSE:
                // Enumerate all child windows of this application and
                // notify all of them to get closed
                EnumChildWindows(hwnd,&MyEnumChildProc,0L);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return(DefWindowProc(hwnd,iMsg,wParam,lParam));
}
// EnumChildProc
BOOL CALLBACK MyEnumChildProc(HWND hwndChild,LPARAM lParam)
{
        // code
        SendMessage(hwndChild,WM_CLOSE,0,0L);
        return(TRUE);
}
// Server Window Procedure
LRESULT CALLBACK ServerWndProc(HWND hwndServer,UINT iMsg,WPARAM wParam,LPARAM
lParam)
{
        // function declarations
      void leadtrim(char[]);
      void trailtrim(char[]);
        // variable declarations
        ATOM aActualDataItem;
        WORD wDataFormat,wStatus;
        DDEDATA *pDdeData;
        DDEACK DdeAck;
        GLOBALHANDLE hDdeData;
        BOOL result;
        char AtomString[255],SquareString[255];
        int i,num,NegativeStatusFlag;
        long square;
        // code
```

```
switch(iMsg)
{
case WM_DDE_REQUEST:
      wDataFormat=LOWORD(lParam);
      aActualDataItem=HIWORD(lParam);
      // check for matching of client's sended data format and actual
      // global data format in atom,if no matching send negative
      // acknowledge message
      if(wDataFormat!=CF_TEXT)
      NegativeStatusFlag=1;// for invalid dataformat
      else// valid format
      {
            // get the client passed atom's string
            GlobalGetAtomName(aActualDataItem,AtomString,255);
            // trim the received string
            leadtrim(AtomString);
            trailtrim(AtomString);
            // check the received data
            if(AtomString[0]=='0' || AtomString[0]=='+')
                  i=1;
            else
                  i=0;
      for(;AtomString[i]!='\0';i++)
            {
                  if(AtomString[i]<48 || AtomString[i]>57)
                  {
                        NegativeStatusFlag=2;//alphabetic,float,negative
data
                        break;
                  }
            }
            if(atol(AtomString)>32767)
                  NegativeStatusFlag=3;// beyond limit
            if(NegativeStatusFlag==1 || NegativeStatusFlag==2 ||
              NegativeStatusFlag==3)
            {
                  DdeAck.bAppReturnCode=NegativeStatusFlag;
                  DdeAck.fAck=TRUE;
                  DdeAck.fBusy=FALSE;
                  wStatus=*((WORD *) &DdeAck);
                  PostMessage((HWND)wParam,
                              WM_DDE_ACK,
                                (WPARAM)hwndServer,

PackDDElParam(WM_DDE_ACK,wStatus,aActualDataItem));
            }
            // if OK then,convert it to a number
            num=atoi(AtomString);
            // make square of this number
      square=num*num;
            // convert this square to string
            sprintf(SquareString,"%ld",square);
            // now allocate memory for DDEDATA structure,when we pass
            // data to client,we will pass this structure too.So that
            // the client can get status of the data being passed.
            hDdeData=GlobalAlloc(GHND|GMEM_DDESHARE,
                                 sizeof(DDEDATA)+strlen(SquareString));
```

```
                    pDdeData=(DDEDATA *)GlobalLock(hDdeData);
                    // now assign values to DDEDATA structure members
                    pDdeData->cfFormat=CF_TEXT;
                    pDdeData->fAckReq=TRUE;
                    pDdeData->fRelease=TRUE;
                    pDdeData->fResponse=TRUE;// because data is sent in response
to WM_DDE_REQUEST
                    strcpy((PSTR)pDdeData->Value,SquareString);
                    // un-lock the memory handle so that others can use it
                    GlobalUnlock(hDdeData);
                  // now post WM_DDE_DATA message to the client
                    result=PostMessage((HWND)wParam,
                                       WM_DDE_DATA,
                                             (WPARAM)hwndServer,
                                             PackDDElParam(WM_DDE_DATA,
                                                           (UINT)hDdeData,

aActualDataItem));
                    if(result==0)
                    {
                          // free the global memory
                          if(hDdeData!=NULL)
                                GlobalFree(hDdeData);
                    }
                    // get acknowledgement message from client,if any it sends

      if(PeekMessage(&msg,hwndServer,WM_DDE_ACK,WM_DDE_ACK,PM_REMOVE)==TRUE)
                    {
                          wStatus=LOWORD(msg.lParam);
                          DdeAck=*((DDEACK *) &wStatus);
                        if(DdeAck.fAck==FALSE)// not received by client
                          {
                              if(hDdeData!=NULL)
                                  GlobalFree(hDdeData);
                          }
                    }
              }
              break;
      case WM_DDE_TERMINATE:
              // if this message is received,you should also send the same
              PostMessage((HWND)wParam,WM_DDE_TERMINATE,(WPARAM)hwndServer,0L);
              DestroyWindow(hwndServer);
              break;
      case WM_DESTROY:
              PostQuitMessage(0);
              break;
      }
      return(DefWindowProc(hwndServer,iMsg,wParam,lParam));
}
void leadtrim(char s[])
{
    // variable declarations
    int i,j;
      char t[255];
    // code
    for(i=0;i<=(int)strlen(s)-1;i++)
    {
```

```
        if(s[i]!=' ')
             break;
    }
    for(j=0;i<=(int)strlen(s)-1;i++,j++)
    {
       t[j]=s[i];
    }
    t[j]='\0';
      strcpy(s,t);
}
void trailtrim(char s[])
{
    // variable declarations
    int i;
    // code
    for(i=strlen(s)-1;i>=0;i--)
    {
       if(s[i]!=' ')
             break;
    }
    s[i+1]='\0';
}
```

* Common Dialog Box [Open And Print] Program Using Visual C.*

```
# include <windows.h>
# include <commdlg.h>
# include <stdio.h>
# include <stdlib.h>
# include <malloc.h>
# include "CommonDialog.h"
# define UNICODE

LRESULT CALLBACK WndProc(HWND , UINT , WPARAM , LPARAM);
BOOL CALLBACK AbortPrinterProc(HDC , int);
LRESULT CALLBACK KillPrint(HWND , UINT , WPARAM , LPARAM);

int PrintOk = 1;
HWND hNonModalDlg = NULL;

int WINAPI WinMain(HINSTANCE hInstance , HINSTANCE hPrevInstance , LPSTR lpCmdLine , int
nCmdShow)
{
WNDCLASSEX wndclass;
HWND hwnd;
MSG msg;
TCHAR szAppName[] = TEXT("MyCommandDialogBox");

wndclass.cbSize = sizeof(WNDCLASSEX);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(NULL , IDI_APPLICATION);
wndclass.hbrBackground = (HBRUSH) GetStockObject(BLACK_BRUSH);
wndclass.hCursor = LoadCursor(NULL , IDC_ARROW);
wndclass.lpszClassName = szAppName;
wndclass.lpszMenuName = TEXT("MyMenu");
wndclass.hIconSm = LoadIcon(NULL , IDI_APPLICATION);
RegisterClassEx(&wndclass);

hwnd = CreateWindow(szAppName , TEXT("My Common Dialog Box Pro.") ,
                    WS_OVERLAPPEDWINDOW , CW_USEDEFAULT ,
                    CW_USEDEFAULT , CW_USEDEFAULT ,
                    CW_USEDEFAULT , NULL , NULL, hInstance , NULL);

ShowWindow(hwnd , nCmdShow);
UpdateWindow(hwnd);

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

```
        return((int) msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
        long GetFileLength(FILE *);
        void GetFileData(FILE * , TCHAR *);
        static HWND hwndEdit = NULL;
        static HINSTANCE hInst = NULL;
        RECT rc;
        OPENFILENAME ofn;
        PRINTDLG pd;
        DOCINFO di;
        TEXTMETRIC tm;
        int i , j , y;
        long lNumLines;
        static FILE *file;
        static TCHAR szFileName[_MAX_PATH];
        TCHAR szFilter[] = TEXT("Text Files (*.TXT) \0*.TXT\0") \
        TEXT("All Files (*.*) \0*.*\0");
        TCHAR *szBuffer = NULL;
        TCHAR STR[255];

switch(iMsg)
{
        case WM_CREATE :
                hInst = (HINSTANCE) ((LPCREATESTRUCT)lParam)->hInstance;
        break;

        case WM_COMMAND :
                switch(LOWORD(wParam))
                {
                        case IDM_OPEN : szFileName[0] = '\0';
                        memset(&ofn , 0 , sizeof(OPENFILENAME));
                        ofn.lStructSize = sizeof(OPENFILENAME);
                        ofn.hwndOwner = hwnd;
                        ofn.hInstance = NULL;
                        ofn.lpstrFilter = szFilter;
                        ofn.lpstrCustomFilter = NULL;
                        ofn.nMaxCustFilter = 0;
                        ofn.nFilterIndex = 1;
                        ofn.lpstrFile = szFileName;
                        ofn.nMaxFile = _MAX_PATH;
                        ofn.lpstrFileTitle = NULL;
                        ofn.nMaxFileTitle = _MAX_FNAME + _MAX_EXT;
                        ofn.lpstrInitialDir = NULL;
                        ofn.lpstrTitle = TEXT("Open A Text File.....\0");
                        ofn.Flags = OFN_FILEMUSTEXIST | OFN_PATHMUSTEXIST | OFN_READONLY
                        OFN_EXPLORER;
                        ofn.nFileOffset = 0;
                        ofn.nFileExtension = 0;
                        ofn.lpstrDefExt = TEXT("*\0");
```

```
                    ofn.lCustData = 0L;
                    ofn.lpTemplateName = NULL;
                    if(GetOpenFileName((LPOPENFILENAME) &ofn) == FALSE)
                    {
                            if(CommDlgExtendedError() != 0)
                            {
                            MessageBox(hwnd , TEXT("Comman Dialog Box Error While Opening File.") ,
                            NULL , MB_OK);
                            DestroyWindow(hwnd);
                            }
                            break;
                    }
SetWindowText(hwnd , szFileName);
file = fopen(szFileName , TEXT("r"));
if(file == NULL)
{
        wsprintf(STR , TEXT("File Opening Error Occurs While Opening The File %s") , szFileName);
        MessageBox(hwnd , STR , NULL , MB_OK); break;
}
szBuffer = (TCHAR*) malloc(2 * GetFileLength(file));
if(szBuffer == NULL)
{
        MessageBox(hwnd , TEXT("Can't Allocate Memory.") , TEXT("Memory Error") , MB_OK);
        DestroyWindow(hwnd);
}
GetFileData(file , szBuffer);
GetClientRect(hwnd , &rc);
hwndEdit = CreateWindow(TEXT("EDIT") , NULL , WS_CHILD | WS_VISIBLE | ES_LEFT |
                    ES_MULTILINE | ES_NOHIDESEL | ES_AUTOHSCROLL
                    ES_AUTOVSCROLL , rc.left , rc.top , (rc.right - rc.left) , (rc.bottom - rc.top) ,
                    hwnd , NULL , hInst , NULL);
SendMessage(hwndEdit , WM_SETTEXT , (WPARAM) 0 , (LPARAM) szBuffer);
break;

case IDM_PRINT : if(szFileName[0] == '\0')
{
        MessageBox(hwnd , TEXT("No File For Printing. \n Open A File And Try Again.") , NULL ,
MB_OK);
        break;
}
memset(&pd , 0 , sizeof(PRINTDLG));
pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner = hwnd;
pd.hDevMode = NULL;
pd.hDC = NULL;
pd.Flags = PD_RETURNDC | PD_NOSELECTION | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE;
pd.nFromPage = 0;
pd.nToPage = 0;
pd.nMinPage = 0;
pd.nMaxPage = 0;
pd.nCopies = 1;
pd.hInstance = hInst;
```

```
pd.lCustData = 0;
pd.lpfnPrintHook = NULL;
pd.lpfnSetupHook = NULL;
pd.lpPrintTemplateName = NULL;
pd.lpSetupTemplateName = NULL;
pd.hPrintTemplate = NULL;
pd.hSetupTemplate = NULL;
if(PrintDlg(&pd) == FALSE)
{
if(CommDlgExtendedError() != 0)
        {
                MessageBox(hwnd , TEXT("Commod Dialog Box Error While Opening The Print Dialog Box")
                , NULL , MB_OK);
                DestroyWindow(hwnd);
        }
break;
}
memset(&di , 0 , sizeof(DOCINFO));
di.cbSize = sizeof(DOCINFO);
di.lpszDocName = TEXT("Printing Text File.");
di.lpszOutput = NULL;
di.lpszDatatype = NULL;
di.fwType = 0;

StartDoc(pd.hDC , &di);
GetTextMetrics(pd.hDC , &tm);
lNumLines = GetDeviceCaps(pd.hDC , VERTRES);
lNumLines /= (tm.tmHeight + tm.tmExternalLeading);
PrintOk = 1;
SetAbortProc(pd.hDC , AbortPrinterProc);

hNonModalDlg = CreateDialog(hInst , TEXT("PrintCancel") , hwnd , (DLGPROC) KillPrint);
rewind(file);
        for(i=0;i<pd.nCopies;i++)
        {
        StartPage(pd.hDC);
        y = 0; j = 0;
        do
        {
                if(feof(file)) break;
                if(fgets(STR , 80 , file) == NULL) break;
                STR[wcslen(STR) - 1] = 0;
                TextOut(pd.hDC , 0 , y , STR , wcslen(STR));
                y += tm.tmHeight + tm.tmExternalLeading;
                ++j;
                if(j == lNumLines)
                {
                        EndPage(pd.hDC);
                        j = 0; y = 0;
                        StartPage(pd.hDC);
                }
        }while(!feof(file));
```

```
EndPage(pd.hDC);
rewind(file);
}//for
if(PrintOk)
        {
                DestroyWindow(hNonModalDlg);
                EndDoc(pd.hDC);
        }
DeleteDC(pd.hDC);
fclose(file);
break;

case IDM_EXIT :
DestroyWindow(hwnd);  DestroyWindow(hwndEdit); free(szBuffer);
break;
}//switch WS_COMMAND
break;

case WM_DESTROY : if(file) fclose(file); PostQuitMessage(0); break;
}
return(DefWindowProc(hwnd , iMsg , wParam , lParam));
}

long GetFileLength(FILE *fp)
{
TCHAR ch;
long length;
rewind(fp);
length = 0;
while((ch = fgetc(fp)) != EOF) length++;
rewind(fp);
return(length);
}

void GetFileData(FILE *fp , TCHAR *szBuff)
{
TCHAR ch;
rewind(fp);
while((ch = (TCHAR) fgetc(fp)) != EOF)
{
if(ch == '\n')
{   *szBuff = '\r'; szBuff++; }
*szBuff  = ch;
szBuff++;
}
*szBuff = '\0';
rewind(fp);
}

BOOL CALLBACK AbortPrinterProc(HDC hdc , int error)
{
MSG msg;
```

```
while(PeekMessage(&msg , NULL , 0 , 0 , PM_REMOVE))
if(!IsDialogMessage(hNonModalDlg , &msg))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
return(TRUE);
}

LRESULT CALLBACK KillPrint(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
switch(iMsg)
{
case WM_COMMAND : switch(LOWORD(wParam))
{
case IDCANCEL : PrintOk = 0;
DestroyWindow(hNonModalDlg);
hNonModalDlg = NULL;
return(1);
}
break;
}
return(0);
}
```

*** Entries In ".h" File. ***

```
# define IDM_OPEN 100
# define IDM_PRINT 101
# define IDM_EXIT 102
```

*** Entries In ".rc" File. ***

```
# include <windows.h>
# include "CommanDialog.h"
PrintCancel DIALOG 10 , 10 , 100 , 40
CAPTION "Stop Printing(?)"
STYLE WS_CAPTION | WS_POPUP | WS_SYSMENU | WS_VISIBLE | WS_CHILD
BEGIN
        PUSHBUTTON "Cancel" , IDCANCEL , 35 , 12 , 30 , 14 , WS_TABSTOP
END

MyMenu Menu
BEGIN
        POPUP "&File"
                BEGIN
                        MENUITEM "&Open" , IDM_OPEN
                        MENUITEM "&Print" , IDM_PRINT
                        MENUITEM "&Exit" , IDM_EXIT
                END
END
```

FunctionDetails

OPENFILENAME :-
The OPENFILENAME structure contains information that the GetOpenFileName and GetSaveFileName functions use to initialize an Open or Save As dialog box. After the user closes the dialog box, the system returns information about the user's selection in this structure.

GetOpenFileName() :-
The GetOpenFileName function creates an Open dialog box that lets the user specify the drive, directory, and the name of a file orset of files to open.
BOOL GetOpenFileName(LPOPENFILENAME lpofn // initialization data);

SetWindowText() :-
The SetWindowText function changes the text of the specified window's title bar (if it has one). If the specified window is control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application.
BOOL SetWindowText(HWND hWnd, // handle to window or control
LPCTSTR lpString // title or text);

PRINTDLG :-
The PRINTDLG structure contains information that the PrintDlg function uses to initialize the Print dialog box. After the user closes the dialog box, the system uses this structure to return information about the user's selections.

PrintDlg() :-
The PrintDlg function displays a Print dialog box. The Print dialog box enables the user to specify the properties of a particular print job. Windows 2000 or later: The PrintDlg function has been superseded by the PrintDlgEx function. PrintDlgEx displays a Print property sheet, which has a General page containing controls similar to the Print dialog box.
BOOL PrintDlg(LPPRINTDLG lppd // initialization data);

DOCINFO :-
The DOCINFO structure contains the input and output file names and other information used by the StartDoc function.

StartDoc() :-
The StartDoc function starts a print job.
int StartDoc(HDC hdc, // handle to DC
CONST DOCINFO* lpdi // contains file names);

GetDeviceCaps() :-
The GetDeviceCaps function retrieves device-specific information for the specified device.
int GetDeviceCaps(HDC hdc, // handle to DC
int nIndex // index of capability);

SetAbortProc() :-
The SetAbortProc function sets the application-defined abort function that allows a print job to be canceled during spooling.
int SetAbortProc(HDC hdc, // handle to DC
ABORTPROC lpAbortProc // abort function);

StartPage() :-
The StartPage function prepares the printer driver to accept data.
int StartPage(HDC hDC // handle to DC);

EndPage() :-
The EndPage function notifies the device that the application has finished writing to a page. This function is typically used to direct
the device driver to advance to a new page.
int EndPage(HDC hdc // handle to DC);

EndDoc() :-
The EndDoc function ends a print job.
int EndDoc(HDC hdc // handle to DC);

THEORY

Program (Calling Start Page(), StartDoc(),EndDoc(),EndPage()).

↓

GDI call like (LineTo(),Rectangle(),TextOut())

↓

GDI Modules i.e GDI32.dll

↓

Spooler process i.e spooler exe || This cerate Job_Description_File

↓

Print Processor and Print Driver

↓

Calls gdiplaySpoolStream()

↓

Print router(routing from O.S to print provider)

↓

Print Provider

↓

Language monitorisation if necessary

↓

Port monitor (i.e.paralle port or USB port utilization)

↓

Printer

COMMON DIALOG BOX

There are 10 Common Dialog Boxes

1) Open File
2) Save As File
3) Find
4) Replace

5) Print
6) Print Setup
7) Print Property Sheet
8) Page Setup
9) Color
10) Font

Each has its associate 4 things
1) Structure 2) Display function
3) Hook Function 4)Windows Messages of Common Dialog Box

Name of Common Dialog Box	Structure	Display Function	Hook Function
File Open	OPENFILESTRUCTURE	GetOpenFileName()	OFNHookProc()
File Save As	OPENFILESTRUCTURE	GetSaveFileName()	OFNHookProc()
Find	FINDREPLACE	FindText()	FRHookProc()
Replace	FINDREPLACE	ReplaceText()	FRHookProc()
Print	PRINTDLG or PRINTDLGEX	PrintDlg() or PrintDlgEx()	PrintHookProc() or SetupHookProc()
Print Setup	PRINTDLG or PRINTDLGEX	PrintDlg() or PrintDlgEx()	PrintHookProc() or SetupHookProc()
Print Property Page	PRINTDLG or PRINTDLGEX	PrintDlgEx()	COM based
Page Setup	PAGESETUPDLG	PageSetUpDlg()	PageSetupHook() or PagePaintHookProc()
Color	CHOOSECOLOR	ChooseColor()	CCHookProc()
Font	CHOOSEFONT	ChooseFont()	CFHookProc()

How to program Common Dialog Box →

1>declare respective Common Dialog Box's Structure variable.
2> By using memset() function initialize all its members to 0
Prototype = void * memset(void * , int , size_t);
3> Now fill all the members like wndclass
4>Call associated function. All these function returns BOOL type valuse i.e. on success TRUE and on failure FALSE. They all take only one parameter, i.e address of above "filled" structure.
5> On return function fills one of its member with user selected value. Use that value as necessary
E.g. If we passed character array (empty) say szFileName in lpStrFile member of OPENFILENAME structure, then on return, this szFileName buffer will have the user selected file name with it's full path and and another member lpStrFileTital of same structure when given an empty buffer, on return fills it with file name only(not path).
6> As an advance programming practice You can specify your won hook.
Callback function in all above structure (all have lpfnHook member) and then in that function using common dialog box message (see MSDN) you can control the behaviour of common dialog box as per your need. Otherwise lpfnHook is kept NULL.

How to program Printer :
Import functions
1>PrintDlg() or PrintDlgEx().

2>StartDoc()
3> StartPage()
4>EndDoc()
5>EndPage()

1>As specified above fill PRINTDLG structure and call PrientDlg() function. On returns its member hdc will have printers Device's context handle.
2> Get text metrics for this hdc, (according to change made by user in print common dialog box)
3> By using GetDeviceCaps() and textmetric structure, get how many lines can fit on one page, as per the user seleted page type (e,g, letter A_4,A_5, B_5,etc)
4> Declare a variable of DOCINFO structure make its member 0 by using memset() and then initialize its member
5> Now call StartDoc() by specifying printer's hdc as first param and address of DOCINFO structure variables a second param.
6> By using SetAbortProc () specify abort procedure's name. The procedure must be declared globally as BOOL CALLBACK <name> (HDC,int).
7> Now setup a loop for nCopies member of PRINTDLG structure. Inside the Loop call StartDoc (), set page co-ordinate (as like you do in printing) Then read the data line by line, advance the co-ordinate, especially the co-ordinate until the number of lines on that page (calculated in step (3)) ends. If number of lines ends, call EndPage(). Again set the co-ordinates to initialize valuse for next page and again call the StartPage(). After en of data, exit the loop and again call EndPage().
8> After coming out of the loop call EndProc (), and then call the DeleteDC() to delete printers hdc.
9> Inside the AbortProc () declare the variable of MSG type and setup the Message loop for PeekMessage() using PM_REMOVE option. Inside the loop there are 2 usual calls for our WinMain's message loop

Inside the Message Loop you can use ISDialogMessage() method, if you specify the user to cancel printing "in between", by using a modeless dialog box.

10> If you give a change to user to cancel the printing in between, you have to create a modeless dialog box ..

a> In ".rc" file create a templet for dialog box, which has one "cancel"push button having default ID as IDCANCEL.
b> Specify its callback as usual dialog box's callback globally.
c> Declare one handle of HWND type and for its call CreateDialog () before the loop of step(5).The handle should be declare globally to use in AbourtProc () and callback.
d> Inside the callback, handle WM_COMMAND for "case IDCANCEL", in which using destroy window() destroy the modeless dialog box.

Process of Multiple Printing:-

Set the AbourtProc ()
↓
Call startDoc()
↓
Check for error
↓
If error cancel printing (i.,e.break)
↓
If success, call SartPage()
↓
Check for Error
↓
If error cancel printing(i.e. break)
↓

If success, call your GDI output functions for this page

↓

After end of page limit call EndPage()

↓

Here if user cancels printing (by using modeless dlg) go out of loop and call EndDoc()

↓

If user does not cancel printing, check whether document is complete (by nCopies member of PRINTDLG)

↓

If decument is completed. Call EndDoc()

↓

If document is not yet completed, go back to the step of startPage().

Spooler: - Printing involves 3 modules interaction (a) GDI32.dll library. (b)The pointer driver (.drv) and (c) The windows print spooler (spooler.exe). When PrintDlg() common dlg function is called the driver of printers get loaded in RAM. StartDoc () function inizalise this driver andEndDoc() de initializes it When StartPage() is Called, printer driver translate all GDI calls to printer specify code by using technique of "Banding" in which single page is sub deviled in to rectangular bands and translation is done for each band. The translated code is first stored into a temp file whose names begins with "~spl"and whose extetion is ". tmp" After complete transaction print spooler is called indicating that a new print job is ready. Now spooler takes this temp file (i.e. also called as job Descriptor File) and sends its to the printer for the printing. The process of printing takes place in separate thread, while application can proceeds for its next task. Spooler internally uses various port communication functions. After dong the job of printing this temp file is deleted.

Note :- Spooler always gets loaded at boot time.

* Sample Program Of ToolTip Display On The Window. *

```
# include <windows.h>
# include <commctrl.h> // Compulsory.
# define UNICODE
// Link "comctl32.lib" In /Project/Settings/Link Tab Compulsory.

LRESULT CALLBACK WndProc(HWND , UINT , WPARAM , LPARAM);

int WINAPI WinMain(HINSTANCE hInstance , HINSTANCE hPrevInstance , LPSTR lpCmdLine , int
nCmdShow)
{
WNDCLASSEX wndclass; MSG msg; HWND hwnd; TCHAR szAppName[] = TEXT("My Window");

INITCOMMONCONTROLSEX icc;
wndclass.cbSize = sizeof(WNDCLASSEX);
wndclass.style = CS_VREDRAW | CS_HREDRAW;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.lpfnWndProc = WndProc;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(NULL , IDI_APPLICATION);
wndclass.hbrBackground = (HBRUSH) GetStockObject(BLACK_BRUSH);
wndclass.hCursor = LoadCursor(NULL , IDC_ARROW);
wndclass.lpszClassName = szAppName;
wndclass.lpszMenuName = NULL;
wndclass.hIconSm = LoadIcon(NULL , IDI_APPLICATION);
RegisterClassEx(&wndclass);

hwnd = CreateWindow(szAppName , TEXT("My Window") , WS_OVERLAPPEDWINDOW ,
                    CW_USEDEFAULT , CW_USEDEFAULT , CW_USEDEFAULT ,
                    CW_USEDEFAULT , NULL , NULL , hInstance , NULL);

ShowWindow(hwnd , nCmdShow);
UpdateWindow(hwnd);

// Initialising Common Control Library.
icc.dwSize = sizeof(INITCOMMONCONTROLSEX);
icc.dwICC = ICC_BAR_CLASSES;
```

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


LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
static HWND hwndTT;
static HINSTANCE hInst;
TOOLINFO ti;

TCHAR *STR = TEXT("This Is My First ToolTip Hi Hi Hi.....");
/*
STR Must Be A Character Pointer Because The Variable To Whome We Are Goning To Assign This
Variable Is "lpsz" Type. i.e "TOOLINFO.ipszText" Is Long Pointer Type(lpsz) So We Can't Assign A
Character Array To It , Assignment Variable Must Be Of Pointer Type;
e.g. => ti.lpszText = STR Is Correct And wcscpy(ti.lpszText , STR) Is Wrong Where "ti" Is Of TOOLINFO
Structure's Variable And STR Is Character Buffer(Must Be A Character Pointer).
*/

PAINTSTRUCT ps;
RECT rc;
HDC hdc;
int scrHeight , scrWidth;
MSG msg;
switch(iMsg)
{
case WM_CREATE : hInst = ((LPCREATESTRUCT)lParam)->hInstance;
        hwndTT = CreateWindowEx(0L , TOOLTIPS_CLASS , NULL , TTS_ALWAYSTIP | WS_CHILD |
        WS_VISIBLE | WS_BORDER ,CW_USEDEFAULT ,CW_USEDEFAULT , CW_USEDEFAULT ,
        CW_USEDEFAULT , hwnd , NULL , hInst , NULL);
break;
```

```
/*
hwndTT = CreateWindow(TOOLTIPS_CLASS , NULL , TTS_ALWAYSTIP | WS_CHILD | WS_VISIBLE |
WS_BORDER , CW_USEDEFAULT ,CW_USEDEFAULT , CW_USEDEFAULT , CW_USEDEFAULT ,
hwnd , NULL , hInst , NULL);
*/

case WM_PAINT : hdc = BeginPaint(hwnd , &ps);
        GetClientRect(hwnd , &rc);
        SetTextColor(hdc , RGB(255 , 0 , 0));
        SetBkColor(hdc , RGB(0 , 0 ,0));
        DrawText(hdc , TEXT("ToolTip Program Testing.") , -1 , &rc , DT_SINGLELINE | DT_CENTER |
        DT_VCENTER);
        EndPaint(hwnd , &ps);

        ZeroMemory(&ti , sizeof(TOOLINFO));
        scrWidth = GetSystemMetrics(SM_CXSCREEN);
        scrHeight = GetSystemMetrics(SM_CYSCREEN);
        ti.cbSize = sizeof(TOOLINFO);
        ti.uFlags = TTF_CENTERTIP;
        ti.hwnd = hwnd;
        ti.rect.left = 0;
        ti.rect.top = 0;
        ti.rect.right = scrWidth;
        ti.rect.bottom = scrHeight;
        ti.lpszText  = STR;
        SendMessage(hwndTT , TTM_ADDTOOL , (WPARAM) 0 , (LPARAM) (LPTOOLINFO) &ti);
break;
case WM_MOUSEMOVE :
        msg.wParam = wParam;
        msg.lParam = lParam;
        msg.message = iMsg;
        msg.hwnd = hwnd;
        SendMessage(hwndTT , TTM_RELAYEVENT , (WPARAM) 0 , (LPARAM) (LPMSG) &msg);
break;
case WM_DESTROY : PostQuitMessage(0); DestroyWindow(hwndTT);
break;
}
return(DefWindowProc(hwnd , iMsg , wParam , lParam));
}
```

Keyboard Accelerator

When we use a Popup Menu in front of some Menu items, some keyboard combination is written. This keyboard communication is called Accelerator or Accelerator Key.

As its name suggests, this is such a keyboard key or keyboard combination invoking which directly executes the respective Menu item without even selecting a Menu or its term. This is because keyboard accelerators direct send **WM_COMMAND** message to the respective Menu item to which the accelerator key is assigned.

We may think that as keyboard accelerators are concerned with Menu items, one of them may be dependent of other, but in real sense this is not true. Menus & keyboard accelerators may be independent of each other; means there may be such a Menu which is not support keyboard accelerator. Also there might be such a keyboard accelerator which does not correspond to a Menu.

This keyboard accelerator are , because they reduce the time of first selecting the Menu & then selecting Menu item.

As like Menu, keyboard accelerator is also a resource.

f How to add keyboard accelerator to our program which has Menu ?

:

- Step 1 : Make modification to existing Menu template in '.rc' file.
- Step 2 : Write accelerator template '.rc' file.
- Step 3 : Loading of accelerator in **WinMain ( )**.
- Step 4 : Modification of **GetMessage ( )** loop accordingly.

I. Make modification to existing Menu template in '.rc' file.

: As an example, we will take Edit Menu. Suppose it has 4 Menu. Its Cut, Copy, Paste, Delete.

```
MMENU MENU
BEGIN
        POPUP "&File"
                BEGIN
                ……. // As seen before
                END
        POPUP "&Edit"
```

```
                BEGIN
                MENUITEM "&Cut \t Shift + Del", IDM_CUT
                MENUITEM "C&opy \t Ctrl + Ins", IDM_COPY
                MENUITEM "&Paste \t Shift + Ins", IDM_PASTE
                MENUITEM "&Delete \t Del", IDM_DELETE
                END
        MENUITEM "&Help", IDM_HELP
    END
```

The general method of adding accelerator key to string of Menu item is giving a '**\t**' after the Menu item string, so that there will be enough space between Menu item string and string of accelerator key combination.

After '**\t**' specify the accelerator key or key combination as needed. (**\t** gives one tab space)

II. Write accelerator template in '.rc' file ?

: The resource statement for keyboard accelerator is **ACCELERATORS**. Its syntax is :

<desired resource name> ACCELERATORS
BEGIN
….
….
END

There are 4 methods of assigning keyboard accelerator to a Menu item.

A) When a alphabet is going to be used as accelerator key.
< **"Alphabet" >, <ID of Menu item>, SHIFT**
e.g. If you want to assign O for Open Menu item in File Menu, then the statement is : **"O", IDM_OPEN, SHIFT**

B) When you want to use alphabet with control key.
< "^Alphabet" >, <ID of Menu item>
e.g. If you want to give **Ctrl + O** combination to Open Menu item of File Menu. Statement : **"^O", IDM_OPEN, SHIFT**
Shift takes care of case-sensitivity.

C) When you want to use ASCII value of alphabet instead of alphabet, then…
< ASCII value of alphabet >, ASCII, <ID of Menu item>, SHIFT e.g. If want to assign 'O' ASCII value to Open Menu item of File Menu.

D) When you want to assign virtual key to the Menu item, then…
<Virtual key code>, <ID of Menu item>, <VRTKEY>
e.g. If you want to assign virtual key of 'O' to Open Menu item of File Menu
VK_O, IDM_OPEN, VIRTKEY
Note : *If you want to use function key with virtual key code like VK_F9, then as like **B**, you cannot add '^'. Because in **B**, key was written in string form while virtual key is not specified with string form.*

So to add Control, Shift or Alt key with virtual key, add their respective resource statement, CONTROL, SHIFT, ALT respectively.
e.g. In **Turbo C** , Ctrl + F4 executes Run Menu item whose accelerator statement is **VK_F9, IDM_RUN, VIRTKEY, CONTROL**

Out of above 4 ways B & D are most commonly used. Using above syntax our Edit Menu accelerator resource statement will be :

```
MYACCEL ACCELERATORS
BEGIN
        VK_DELETE, IDM_CUT, VIRTKEY, SHIFT
        VK_INSERT, IDM_COPY, VIRTKEY, CONTROL
        VK_INSERT, IDM_PASTE, VIRTKEY, SHIFT
        VK_DELETE, IDM_DELETE, VIRTKEY
END
```

If 'O' key is to be assigned with Open Menu item of File Menu with case sensitivity of uppercase only, then statement will be :

a) "O", IDM_OPEN
b) "^O", IDM_OPEN
c) 79, ASCII, IDM_OPEN
d) VK_O, IDM_OPEN, VIRTKEY

- **Some important key codes are :**

VK_BACK : Backspace
VK_INSERT : Insert
VK_DELETE : Delete
VK_TAB : Tab
VK_ENTER : Enter
VK_F1 to VK_F12 : F1 to F12 function keys
VK_RETURN : Enter
VK_SHIFT : Shift
VK_ALT & VK_MENU : Alt
VK_SPACE : Spacebar
VK_PAUSE : Pause
VK_CAPITAL : Caps Lock
VK_SNAPSHOT : Print Screen
VK_NUMLOCK : Num Lock
VK_SCROLL : Scroll Lock

Some are already given under scrolling section in topic of 'Text'.

III. Loading of accelerator in WinMain ( )

: Declare handle of **HACCEL** as : **HACCEL hAccel ;**

Use **LoadAccelerators ( )** with first parameter as instance handle and second parameter is name of resource for **ACCELERATORS** in '.rc' file. This function returns valid **HACCEL** type handle. Hence assign return value to above **hAceel**.

hAccel = LoadAccelerators (hInstance, "MyAccel") ;

Instead of using string you use integer, then use **MAKEINTRESOURCE ( )** macro as explained before.

IV. Modification of GetMessage ( ) loop accordingly.

: while (**GetMessage (&msg, NULL, 0, 0))**

```
{
        if ( TranslateAccelerator ( hwnd, hAccel, &msg, ) == 0 )
        {
                TranslateMessage ( &msg ) ;
                DispatchMessage ( &msg ) ;
        }
}
```

We know that keyboard messages needed to be translated (as seen in first chapter), keyboard accelerators are also keyboard messages. Hence they also need to be translated. That is why **TranslateAccelerator ( )** is used inside the **GetMessage ( )** loop.

Note that : *Whenever a key is pressed, program enters in the loop.*
GetMessage ( ) retrieves the pressed key's message and passes it to **TranslateAccelerator ( )**.

Here, **TranslateAccelerator ( )** function looks for the message whether this is an accelerator key or an usual key. If it is an accelerator key, then it returns TRUE (non-zero) & sends **WM_COMMAND** directly to **WndProc ( )** with its **LOWORD (wParam)** as the ID of the Menu item to which the accelerator key was assigned.

But if the passed key message is of an usual key (non-accelerator key), then **TranslateAccelerator ( )** returns zero or FALSE.

Control is given back to our loop & **TranslateMessage ( )** and **DispatchMessage ( )** are executed as usual.

f Dynamic Working with Menu Items.

: Dynamic working with Menu items can be dynamically added, deleted, changed, enabled or disabled or checked or unchecked.

The function used for this purpose are : **GetMenu ( )**, **GetSubMenu ( )**, **( )**, **AppendMenu ( )**, **DeleteMenu InsertMenu ( )**, **EnableMenuItem ( )**, **CheckMenuItem ( )**, **AddMenu ( )**.

1) **GetMenu ( )** :

HMENU GetMenu (VOID) ;

This function returns handle of main Menu. e.g. **HMENU hMainMenu ;** This function assumes that, you had already a main Menu specified in **WNDCLASSEX** or specified in **CreateWindow ( )**. If your program does not have valid main Menu, then function returns NULL.

hMainMenu = **GetMenu ( )** ;

2) **GetSubMenu ( )** :

HMENU GetSubMenu (HMENU, int) ;

First parameter is handle of main Menu whose one of Popup Menu handle you want.

Second parameter is index of the Main Menu item which is Popup Menu. This index is zero based.

e.g. : Suppose we have three Menu items : File, Edit & Help. Out of them File, Edit are Popup Menu items while Help is a simple Menu item.

Now suppose we want handle of File Popup Menu. Then its index will be zero. Similarly, for Edit Popup Menu the index will be 1.

HMENU hMainMenu, **hFilePopupMenu** ;

hMainMenui = GetMenu ( ) ;

hFilePopupMenu = GetSubMenu (hMainMenu,

If your main Menu does not have any valid Popup Menu, then this function returns NULL.

3) **AppendMenu ( )** :

This function is used for appending Menu items at the bottom.

BOOL AppendMenu (HMENU, UINT, int, LPCTSTR) ;

First parameter is handle of the Menu in which you are going to append the Menu item. Second parameter is Menu flags prefixed by **MF_**. There are 12 such Menu flags.

1) MF_BITMAP : When the Menu item is a bitmap image.
2) MF_CHECKED : When the Menu item should be checked.
3) MF_DISABLED: When the Menu item is to be disabled.
4) MF_ENABLED : When the Menu item is to be enabled.
5) MF_GRAYED : When the Menu item is to be disabled & grayed.
6) MF_MENUBARBREAK : when a main Menu is to be displayed in two Menu bars or a Popup Menu is to be displayed in two columns.
7) MF_MENUBREAK : Same as above.
8) MF_OWNERDRAW : When the Menu item is owner drawn.

9) MF_POPUP : When the Menu item is going to display a new Popup Menu (Save Menu in our previous figure).
10) MF_SEPARATOR : To add a horizontal separation for between two Menu item.
11) MF_STRING : When you are going to give Menu item as string.
12) MF_UNCHECKED : This is default style in when Menu items are unchecked.

Third parameter is ID of Menu item. But if Menu item is of MF_POPUP, then this parameter will be handle of Popup Menu (e.g. Save)
Fourth parameter depends on second parameter. If 2nd parameter is MF_BITMAP, then 4th parameter is handle of bitmap of HBITMAP type. If 2nd parameter is MF_STRING, then 4th parameter will be actual string of Menu item. You can add "&" for hotkey as "&File" or even "**\t**" for tab space. Also the accelerator key combination within this string.

Now suppose to oursuppose to our example we want add new menu item to edit menu item will be-

AppendMENU (hEditMenu, MF_STRING, IDM_SELECTALL, "& Select all \t Ctrl + a");

if we want to append cut in edit menu
HMENU hMAINMENU ,hEDITPOPOP MENU ();
HMAIN MENU = GETMENU ()
hEDITPOPOPMENU= GETSUBMENU (hMAINMENU,1)
APPENDMEMU (hEDITPOPOPMENU , MFSTRING , IDM_CUT, " & CUT \t SHIFT + DEL ")

4] delete menu() : this function deletes the specified MENU or MENUITEM .

a) BOOL DELETE MENU (HMENU < UINT< UINT)
first parameter is handle of the menu or popop menu it the menu item of which u r going to delete
second parameter is either ID of menu item which u want to delete or zero based index of menu item which u want to delete.

this parameter will depend on second parameter . if second parameter is ID valve then third parameter will be MF_ BYCOMMAND
if second parameter is index valve then 3rd parameter will be MF_ BY POSITION

eg suppose we want to delete "CUT" menu item from edit popop menu then the call to the delete menu will be

by using id valve
delete menu (heditpopop menu, idm_ cot , mf_ bycommand)

by using index or position valve
delete menu (hEDITPOPOP MENU , o , mf_BYPOSITION)

5] INSERT MENU :this function is used to insert menu item in main menu or popop menu . note that append menu () can append menu item at the bottom but insert menu () can add menu items anywhere

a) BOOL INSERT MENU (HMENU , UNIT , UINT,UINT,LPCTSTR)
first parameter is handle of mainmenu or popop menu inside which u r going to insert new menu items .2nd parameter id of that menu before which you want to insert menu item either MF_BYCOMMAND or MF_CHECKED, MF_GRAYED, MF_SAPRATOR etc

if u want to insert () to work like appendmenu () then use MF_BYPOSITION followed to set 1
4th parameter is id of menu item that you want to insert .if u newly inserted menu item is going to be popop menu then add MFPOPOP style in 3rd parameter with other styles or oprator then specify o for this 4th parameter .5th parameter is the string of menu item with hot or accelerator key

eg suppose we want to insert copy menu between cut and paste menu item of edit menu

INSERTMENU (hEDITPOPOPMENU , IDM_PASTE , MF_BYCOMMAND , IDM_COPY, " & COPY \t ctrl + ins");

whenever u add or remove menu items from main menu or popop menu by using appendmenu () insertmenu () or deletemenu ()
then after complete all such oprations use DRAWMENUBAR () API so that system will redraw the menu DrawMenuBar (void)

CLIPBORD

In Windows programming, as it is a multitasking platform, there may be many programs running simultaneously on the desktop. There should be some way through which one program should be able to communicate with other program for give & take of data called Data Exchange . Also there may be some way by which one program should be able to communicate with other programs on the desktop.

The basic assumption is that, the two communicating programs running on the desktop. If this is the condition, then there are 5 ways of program communication called together as----

InterProcess Communication (or IPC) those are..

1) **Clipboard.**
2) **Dynamic Data Exchange (DDE).**
3) **Dynamic Link Library (DLL).**
4) **Process Based Multitasking.**
5) **OLE & COM.**

Out of these 5,we are going to discuss about Clipboard.

The Windows clipboard allows data to be transformed from one program to another provided both should use clipboard in their codes.

¾ The process of data transfer is as follows :-

The program that going to put data on clipboard (called as Clipboard Owner or Server or Clipboard Provider) puts desired data in clipboard.

Data is transferred to clipboard.

The program which is going to receive the data (called as clipboard Viewer or client) takes data from the clipboard.

¾ So What the Entity Clipboard Is?

Actually clipboard is a memory block, obviously global, means visible or accessible to all, and capable of receiving & transferring data when accessed by program. In Windows 95, in system tools, there is a "clipboard viewer" program, which shows current clipboard contents when clicked on it else it is just an empty window.

This does not mean any program, can receive data from clipboard. If a program wants to receive data from clipboard, then it must be in a format, which is supported by that program.

For Example :- If a program (server) puts a bitmap picture in clipboard, and some another program does not support bitmap format ,then it is not able to receive that clipboard bitmap data.

¾ The Standard Clipboard Data Formats :-

There are various clipboard data formats supported by windows. All these formats are expressed as identifiers in windows header files. The most common data formats are:-

1) CF_TEXT :- a null terminated ANSI string data.

2) **CF_BITMAP :- Device Dependent bitmap picture data.**
3) **CF_MATAFILEPICT :- a meta file picture data.**
4) **CF_ENHMETAFILE :- a enhanced meta file data.**
5) **CF_SYLK :- Microsoft's Symbolic Link format data.**
6) **CF_DIF :- Data Interchange format data.**
7) **CF_TIFF :- Tag Image File Format data.**
8) **CF_OEMTEXT :- similar to CF_TEXT but characters are in OEM character set.**
9) **CF_DIB :- Device Independent Bitmap picture data.**
10) **CF_PALETTE :- a color palette data used along with CF_DIB.**

The CF word in all above formats is a acronym for Clipboard Format.

- **As stated before clipboard resembles with a global memory block , hence the data provider (i.e. server or owner) when puts data in global memory ,then it must not use that memory in the same program anywhere else. Because as soon as global memory handle filled with desired data , is passed to clipboard ,the memory belongs to OS and program does not have rights to access that memory.**

* How a program transfer data to clipboard :-

This is the job of the clipboard owner or server or clipboard provider.
(All following variables are local to WndProc)
***String data(i.e. CF_TEXT) :- Assume that we have a string and "pstr" is the pointer to that string.**

STEPS :-

1) Allocate a "movable memory block" of "len" size.(here len is length of the string)

The memory block should be global.

```
HGLOBAL hGlobalMemory;
Char *pGlobalMemory;
//code
hGlobalMemory = GlobalAlloc(GMEM_MOVABLE|GMEM_ZEROINIT , len+1);
//if function fails hGlobalMemory is NULL hence here you should do memory allocation error
//checking.
PGlobalMemory = (char *) GlobalLock (hGlobalMemory);
//instead of using GMEM_MOVABLE| GMEM_ZEROINIT , you ca use GHND which is
//combination of 2.
```

HGLOBAL GlobalAlloc(UINT,SIZE_T); 1st is flag, 2nd –number of bytes to allocate. LPVOID GlobalLock(HGLOBAL);

Returns pointer to first byte of parameter's memory block and increments lock count on this memory.

Here we declare a global handle to hold memory and then using GlobalAlloc() API we get a memory block in this handle. By using GMEM_MOVABLE we make the memory block movable. By using GMEM_ZEROINTI we fill the entire block by zeros. we use len+1 because "len" is length of our string data & 1 is for the terminating null character.
Then we declare a character pointer pGlobalmemory and using GlobalLock() API we convert the memory handle into a pointer.our data is in string form, we cast the pointer to char *. (because GlobalLock returns LPVOID).

2) Now copy the desired string data into the global memory pointer.

```
for(i = 0 ; i < len ; i++)
{
    *pGlobalMemory = *pstr;
     pGlobalMemory++;
     pstr++;
}
```

OR

Simply copy the string data into global memory pointer. Strcpy(pGlobalMemory,pstr);

3) To allow the memory to move (as we use *GEME_MOVABLE*), there must be unlocking function for each instance at locking function *GlobalUnlock(hGlobalMemory);*
***//BOOL GlobalUnlock(HGLOBAL);* this has effect only on that handle which is created by *GMEM_MOVABLE* flag. it decrements lock count.**

4) Now we have our string data in pGLobalMemory pointer which represents hGLobalMemory handle. Thus indirectly we can say we have our data in hGlobalMemory returned memory block.

To allow it to get copied to clipboard ,first open the clipboard.
***OpenClipboard(hwnd);* //OpenClipboard(HWND);**

5) The clipboard may have any previous data so first empty it. *EmptyClipboard( );* //BOOL EmptyClipboard(void);

6) Now set the clipboard data to our string data (which is refernced by global memory handle. *HANDLE*
***SetClipboardData(UINT,HANDLE);*//SetClpiboardData(CF_TEXT,hGlobalMemory);**

7) Now Close the clipboard.
***CloseClipboard( );* //CloseClipboard(void);**

Usually all above steps are carried out in WM_CRAETE message case construct.
By displaying message box at completion of copying of data to clipboard in same above WM_CREATE, you can save unnecessary WM_PAINT.

Thus such owner sided program may have only 2 cases WM_CRAETE & WM_DESTROY.

But condition may vary according to program's needs.

How A Program Receives Data From Clipboard :-

This is the job of clipboard viewer or clipboard user.
String data (i.e.CF_TEXT) :- Assume that previous program copied data to clipboard. Now our program will receive that data in WM_CREATE and will display the data either in MessageBox in same above WM_CREATE or by using WM_PAINT.

STEPS :-

1) We should have a variable of handle type, to receive data from clipboard. then we should have a pointer of char type (as our data is string) to point to handle of clipboard.
HANDLE hClipboardMemory;

2) open the clipboard. OpenClipboard(hwnd);

3) Get the data from clipboard in handle
hClipboardMemory = GetClipboardData(CF_TEXT); if there is no data , or if data is not in correct format (i.e. not CF_TEXT), then error checking & close the opened clipboard.

```
if(hClipboardMemory = = NULL)
{
        //Display MessageBox of error message
        CloseClipboard( );
         DestroyWindow(hwnd);
         Exit(0);
```

4) Now lock the handle (obviously no error in previous step) to convert it into pointer.
PClipboardMemory=(char *)GlobalLock(hClipboardMemory);
Now our pClipboard data from the clipboard.

5) To get data from clipboard memory pointer, we should have a character pointer say &pstr which must be equal to the memory block bytes returned by hClipboardMemory.
It seems that pClipboard character pointer contains our required string data and we can use it directly, without having pstr variable. But as pClipboardMemory belongs to clipboard, we should not use it directly (though possible) It is better to have its copy in pstr.

```
pstr = (char *) malloc(GlobalSize(hClipboardMemory));
```

usually we use sizeof operator in malloc( );but here we are dealing with global memory handle, hence instead of sizeof operator use API, GlobalSize( ); //SIZE_T GlobalSize(HGLOBAL);

6) Now we have 2 character pointer. One is pointing to the clipboard data handle i.e. pClipboardMemory and this has our required string's address. Other character pointer is pstr which is now empty and will going to be used to receive the string from pClipboardMemory. strcpy(pstr,pClipboardMemory);

7) As in previous program, unlock the clipboard handle GlobalUnlock(hClipboardMemory);

8) Close the Clipboard .CloseClipboard( );

9) In WM_CREATE , or where you used hClipboardMemory,after closing clipboard free the memory by GlobalFree(hClipboardMemory);

10) In WM_DESTROY ,or in WM_PAINT use Free(pstr);//for this you may declare pstr as static.
//if clipboard data belongs only to your program empty the clipboard before using it.
//also use GlobalFree( ) Free( ) fro memory deacllocation.

Now either use MessageBox( ) or WM_PAINT to display the pstr (i.e. our retrieved string data received from clipboard)

Use Of Clipboard For Bitmap Data :-

This is done by using CF_BITMAP format.

***. Suppose a server is converting its desktop to a bitmap and copying it to the clipboard and a client is receiving this clipboard data to visualize server's desktop on client's screen.**

***. How server will put its desktop on the clipboard?**

1) **Get hdc for entire screen by CreateDC(TEXT("DISPLAY"),NULL,NULL,NULL);say hScreenDC.**
2) **Get hdc for graphic device context by GetDC(hwnd); say hdc.**
3) **By using GetDeviceCaps( ) twice ,get horizontal and vertical resolution in maxX,maxY.**
4) **Now Create a bitmap handle say hBitmap by CreateCompatibleBitmap(hScreenDC,maxX,maxY);**
5) **CreateCompatible dc for hdc and save it in memDC by hMemDC=CreateCompatibleDC(hdc);**
6) **Select hBitmap into hMemDC by SlectObject(hMemDC,hBitmap);**
7) **Now cal BitBlt or StretchBlt , to copy hScreenDC to hMemDC. StretchBlt(hMemDC,0,0,maxX,maxy,hScreenDC,0,0,maxX,maxY,SRCCOPY);**
8) **Now open the clipboard by OpenClipboard(hwnd);**
9) **Clean previous contents of clipboard if any,by EmptyClipboard.**
10) **Now set clipboard's data to our bitmap by SetClipboardData(CF_BITMAP,hBitmap);**
11) **Close the clipboard by CloseClipboard( );**
12) **ReleaseDC hdc & delete hMemDC & hScreenDC.**

How Client will get server's desktop bitmap from clipboard : -

1) **Declare a HBITMAP handle,say hBitmap.**
2) **open the Clipboard ,OpenClipboard(hwnd).**
3) **Get clipboard's bitmap data in hBitmap by GetClipboardData( ). hBitmap = GetClipboardData(CF_BITMAP);**
4) **Close the clipboard by CloseClipboard( );**
5) **Now this hBitmap can be used by GetObject ( ) to get BITMAP structure information. then this info say bm can be further used to create BITMAPINFO structure, which say bmi can be used by GetDIBits( ) and then can be written to a new file by using CreateFile( ) & WriteFile( ) functions. Same strategy can work for device independent Bitmaps by using CF_DIB format instead of CF_BITMAP.**

Use Of Clipboard For MetaFile :-

BY using CF_MATAFILEPICT format and using CreateMetaFile( ) function to get hdc for metafile, say hdcmeta, then by using SetWindowExtEx( ) & SetWindowOrgEx( ) With hdcmeta,dimensions of meta file can be set. Further declaring a pointer of type LPMETAFIELPICT and allocating a global memory block to it ,its members are set for the handle of metafile. To get handle to metafile, call CloseMetaFile( ) as hmf=CloseMetaFile(hdcmeta) ,where hmf is handle to metafile.

Now select the LPMETAFILEPICT variable (already initialized) into a global memory handle, say hGmem. Then as usual open & empty the clipboard and call SetClipboardData(CF_METAFILEPICT,hGmem).and close the clipboard.

By this way *metafile is now on clipboard. Now client can use GetClipboardData( ) with CF_METAFILEPICT ,getreturn value in a global memory handle say hGmem ,then convert this memory to LPMETAFILEPICT variable by GlobalLock( ) and then play the metafile by PlayMetaFile( ) Function.

Use Multiple Data Items :-

When clipboard owner i.e server opens the clipboard tp put data into it , server must first clear the clipboard by EmptyClipboard( ) and then can use SetClipboardData( ) to put new data. This tells us very important fact that ,server can not put a new data into the clipboard when clipboard already has some previous data means clipboard can hold only one data item at a time.
But there is no such limit on number of calls to the SetClipboardData( ) between EmptyClipboard( ) & CloseClipboard( ).In other words ,server can use SetClipboardData( ) multiple times for multiple formats say first CF_TEXT , then CF_BITMAP ,then CF_METAFILEPICT and if wishes again CF_TEXT.
Now a client can call EnumClipboardFormats( ) in a loop by checking its return value.If return value is +ve ,there are yet more data. Then in loop use GetClipboardData to get the data. when return value is 0 means there are no more clipboard data, loop will exit. UINT EnumClipboardFormats(UINT);

```
Int iFormat=0;
While(iFormat = EnumCLipboardFormats(iFormat))
{
  //get clipboard data.
}
CloseClipboard( );
```

If you want how many dataformats right now there in clipboard ,you can use CountClipboardFormats( ) to get the required count as its integer return value.
int CountClipboardFormats(void);

Note that,when server uses multiple calls to SetClipboardData( ) between EmptyClipboard( ) & CloseClipboard( ) to put multiple data items in clipboard ,the next occurrence of EmptyClipboard( ) deletes all items in one single call.

Delayed Rendering :- **When server puts data item into the clipboard, it actually makes a memory copy of data. so if client never demands this data, memory is wasted if after long time , then too , memory remains unnecessarily occupied for long time.**

To avoid this problem, a better solution is that , tell OS that server wants to keep such data in clipboard and when some client demands it,O.S will notify server by sending message and then server will put valid data in clipboard by assigning necessary memory to it. This technique is called as Delayed Rendering.

To do this ,set 2nd parameter of SetClipboardData() to NULL, telling OS that ,this is invalid handle of data item. you can use this NULL for all calls to SetClipboardData() between EmptyClipboard() & CloseClipboard()when server wishes top keep multiple data items in the clipboard.

Now when a client needs data, OS Will send you respective messages that a client is there and it needs a data. Now a great question arises ---how OS recognizes that you are the server form so many apps? The answer is that when you call the function OpenClipboard( ),OS saves your window handle, but still you are not "clipboard owner or server" with respect to OS. when you actually call EmptyClipboard() then OS recognizes that you are the server and registers your previously saved window handle as the "Clipboard owner's handle".

So when client needs data, as a server Os sends you message of notification about the presence of such client. There are such 3 messages and every server, which needs such notification, must process these 3 messages by respective message handlers.
These 3 messages are :

1) **WM_RENDERALLFORMATS**
2) **WM_RENDERFORMAT**
3) **WM_DESTROYCLIPBOARD.**

WM_RENDERFORMAT :- **When a client calls GetClipboardData( ) OS sends this message to server,**

whose wParam is the client's requested data format. thus in its handler, server uses switch—case for wParam, obtains memory handle for requested data format and calls SetClipboardData( ).

WM_DESTRYOCLIPBOARD **:- Sometimes, when your server program is already dealing with clipboard, some another server program wants to own the clipboard as server, then that new program calls EmptyClipboard( ) OS sends this message to you ,that you leave the clipboard because some one wants to own it. so in its handler your duty is to call the CloseClipboard( ) and thus you are no longer the clipboard owner.**

WM_RENDERALLFORMATS **:- Sometimes, you are clipboard owner(means your application still has some calls to SetClipboardData( ) with 2nd parameter NULL) and your program is terminating without closing the clipboard then OS sends you this message saying that "render all your formats, close clipboard and then exit". Your duty in its handler, is to open the clipboard ,empty it, put data in memory blocks, call SteClipboardData( ) with valid memory block handless 2nd parameter and then close the clipboard .As your app in this handler calls EmptyClipboard( ),Os will send WM_DESTROYCLIPBOARD to you too, indicating that you had rendered all formats.(i.e. there are no NULL handles remaining in SetClipboardData( )).**

Suppose you want render a single data format say "text" then combine WM_RENDERALLFORMATS & WM_RENDERFORMAT like follow

```
case WM_RENDERALLFORMATS :  OpenClipboard(hwnd);
                            EmptyClipboard( );
                            //fall through
case WM_RENDERFORMAT:
//put text in valid global memory block and call SetClipboardData( )
                            if(iMsg = = WM_RENDERALLFROMATS)
                            CloseClipboard( );
break;
```

If your app uses several clipboard formats, then you don't need to process WM_DESTROYCLIPBOARD. Just process WM_RENDERFORMAT by switch—case to wParam and according to wParam, create valid global memory block for format and call SetClipboardData( ).

<u>Private Data Formats</u> : - Suppose your server wants to use clipboard for another instance of same your server (means you are the client too) and thus you don't want any other client program to use your clipboard data, then you can use Private Data Formats For Clipboard. There are 3 ways to do this:-

1) To make the clipboard data meaningful only to another instance of same your program, use CF_DSPTEXT inplace of CF_TEXT,use CF_DSPBITMAP in place of CF_BITMAP and so on. here "DSP" means "Display", in SetClipboardData( )[for server side] and in GetClipboardData( )[for client side].
But there is a catch! How your client side will know that these "DSP" data formats in clipboard are by your server side and not by any other server who is also using "DSP" type formats.

The way is to declare HWND type handle say hWndCB and call GetClipboardOwner( ) for it. hWndCB = GetClipboardOwner( );
then by using GetClassName( ) for this hWndCB get its class name in a string and compare this string with your class name. If both are equal, the client is another instance of same program.

2) Use SetClipboardData( ) with first Parma as CF_OWNERDISPLAY and second param as NULL. As client now will not be able to get the format, it is duty of server to point the client's window client area. Also 2nd param is NULL, server has to process "Delayed Rendering Message" too. In addition to this ,server has to process another 5 messages:-

1. **WM_ASKCBFORMATNAME ,**
2. **WM_SIZECLIPBOARD ,**
3. **WM_PAINTCLIPBOARD ,**
4. **WM_HSCROLLCLIPBOARD ,**
5. **WM_VSCROLLCLIPBOARD.**

1) WM_ASKCBFORMATNAME **:- Client sends this to server with lParam as the buffer pointer and wParam is max number of characters for this buffer. server must keep name of the format in this lParam.**

2) WM_SIZECLIPBOARD **:- Client sends this to server when client's client area size changes. wParam is handle of client's window and lParam is of LPRECT type which contains client's new client area size. if by fluke LPRECT's 4 members are 0,then this means that client is either destroyed or iconified(minimized).**

3) WM_PAINTCLIPBOARD **:- Client sends this to server to update its client area. wParam is client's window handle and lParam is handle to a memblock which contains client's PAINTSTRUCT. server must use GlobalLock( ) to lock this block and when done must use GlobalUnlock( ) before returning. As PAINTSTRUCT has hdc as one of its member ,server can use this hdc to paint on client's client area.**

4) WM_HSCROLLCLIPBOARD **:-**

5) WM_VSCROLLCLIPBOARD **:- This is send to server by client, when its client area is scrolled. as usual wParam is client's window handle and lParam's LOWORD is scrolling request (i.e. SB_MACROS) and HIWORD is thumb position if the LOWORD is SB_TUMBPOSITION.**

If the client is "clipboard viewer" & uses CF_OWENRDISPLAY, it must send 2,3,4,5 messages to the clipboard owner. Here viewer can obtain the handle of owner by HWND GetClipboarOwner(void).

3) This way is to create your own clipboard formats and register them to OS by using RegisterClipboardFromat( ) which takes name of your private clipboard data format as parameter and returns ID of your registered clipboard format which later you can use in SetClipboardData( ) as is first parameter. This ID is in between 0xc000 and 0xffff.

The Client can obtain this name by first looping with EnumClipboardFormats( ) and inside this loop calls the GetClipboardFormatName( ) by specifying 1st param as the return value of EnumClipboardFormats( ), 2nd param is an empty buffer to hold the format name and 3rd param is size of that buffer.

When the returned name does not match the standard CF formats, then it is a new format. After getting this name ,client can call GetClipboardData( ) by using the return value of

EnumClipboardFormats( ) for this name.

UINT RegisterClipboardFormat(LPCTSTR);
int GetCLipboardFormatName(UINT,LPSTR,int);
in this return value int is nothing but length of returned format name.

Clipboard Viewer :- A program which gets notified (by OS) by the changes in the contents of clipboards called as clipboard viewer. In other words it is a client program to visualize clipboard contents.

Clipboard Viewer Chain :- Any number of clipboard viewers can run in windows at same time ,and they all can be nitrified as well by changes in the clipboard. But from OS point of view ,three is only one clipboard viewer at a time, to which we can call "current clipboard viewer". This is because OS maintains only one clipboard viewer program's window handle ti identify the current clipboard viewer. Thus it sends nitrification only to this window about changes in the clipboard. It is the duty of this "current clipboard viewer" to be a part of "clipboard viewer chain" and pass the same message (that OS sent) to other viewers in the chain.

So when a program registers as "new clipboard viewer" to OS ,it becomes the current clipboard viewer and thus OS gives this program the window handle of previous current clipboard viewer. The new viewer saves this handle and when it receives a clipboard change notification form OS, it sends the same to WndProc( ) of the window of saved handle. Though form OS's point of view, the saved handled program was "previous viewer", form the point of view of new viewer, it (the saved one) is the "next viewer", to which this new one should send the message.

Clipboard Viewer Functions And Messages :- To become part of clipboard chain ,program calls SetClipboardViewer( ),usually in WM_CREATE. Parameter is its window handle and then return value is "window handle of previous clipboard viewer program" . The important duty of this program is to save this return value in static HWND type variable.

Static HWND hWndNext=NULL;
Case WM_CREATE:- hWndNext=SetClipboardViewer(hwnd);
break;

If this program is by fluke ,the first ,viewer then obviously hWndNext will be NULL given by SetClipboardViewer ( ).Now our program is part of clipboard viewer chain (though it might be first one in the chain). So whenever contents of clipboard get changed, OS sends WM_DRAWCLIPBOARD to you, because you are now the most current viewer.
Now what to do?

1) **If you are the only one in the chain, i.e. hWndNext is NULL, then in the message handler of WM_DRAWCLIPBOARD ,don't do any thing ,just "break" out retain 0.**
2) **But if hWndNext is not NULL, then you are not alone and thus you must do 2 things**
   a. **Send this Message to hWndNext by SenMessage(hWndNext,iMsg,wParam,lParam); and**
   b. **invalidate your client area by InvalidateRect(hwnd,NULL,TRUE);**

If your program wants to read the clipboard's currently changed data, then above InvalidateRect() will send WM_PAINT to you and you can then use OpenClipboard(),GetClipboardData() & CloseClipboard() as usual.

When you want to get removed from clipboard viewer chain, call ChangeClipboardChain( ) ,which has 2 HWND type parameters. First is of you and 2nd is of "Next".

***ChangeClipboardChain(hwnd,hndNext);* if you are alone in chain**
then *ChangeClipboardChain(hwnd,NULL);*

Now when you call this function ,OS sends you a message WM_CHANGECBCHAIN and thus you must have handler of this message.
The wParam of this message is your handle and lParam is the handle of "Next".
What to do in its handler? 1.Check whether by wParam is "handle of next", which actually should be of "you". If it is ,then set Next's handle to lParam. This makes sure that next WM _DRAWCLIPBOARD will not sent to you. 2.If wParam is correct, means "handle of you" and lParam is also not NULL(means of "next" handle),then simply send this message to hWndNext as it is.

```
Case WM_CHANGECBCHAIN:
If((HWND)wParam = =hWndNext) HWndNext =
(HWND) lParam; Elseif(hWndNext !=NULL)
SendMessage(hwWndNext,iMsg,wParam,lParam);
break;
```

3) **If wParam is correct i.e. of "you " and lParam is NULL and your hWndNext is also NULL ,then obviously you are alone in chain ,then no need of above work, just "break" out or return 0,in this message handler.**

Suppose you are terminating and you are still part of chain then its your duty to get out of chain before terminations call ChangeClipboardChain( ) in WM_DESTROY before calling PostQuitMessage( ).

If you are not alone in chain means hWndNext is not NULL and if you want to know who is first in chain(means whose hWndNext is NULL),you can use GetClipboadViewer( ) which has no paramaeter and returns handle of first viewer in chain. This function is usually not needed and can return NULL if there is no current clipboard viewer.

A Simulated Example Of Clipboard Viewer Chain :-

when windows boots ,there is no current clipboard viewer
so current clipboard viewer = NULL

If a program having its handle say hwnd1 calls SetClipboardViewer( ) to become part of chain. Obviously function will return null as there is no "next", so…. Current clipboard viewer = hwnd1
hwnd1's Next viewer = NULL.

Now send program having its handle say hwnd2 calls SetClipboardViewer( ) to become part of chain, then function Will return hwnd1 as its "Next Viewer". So current clipboard viewer = hwnd2 hwnd2's Next Viewer = hwnd1
hwnd1's Next Viewer = NULL

Now suppose a third program having Its handle say hand3 and then a fourth program having its handle say hwnd4 call SetClipboardViewer( ) then the chain will be current Clipboard Viewer = hwnd4 hwnd4's Next viewer = hwnd3
hwnd3's Next viewer = hwnd2
hwnd2's Next viewer = hwnd1
hwnd1's Next viewer = NULL

Now if contents of clipboard get changed. So OS will send WM_DRAWCLIPBOARD To current viewer, means to hwnd4.Then hwnd4 will sent to hwnd3,hwnd3 to hwnd2 hwnd2 to hwnd1,hwnd1 has no "Next viewer" i.e. NULL ,it will simply return.

Now suppose hwnd2 decides to remove From clipboard viewer chain by calling ChangeClipboardChain(hwnd2,hwnd1); Then OS will send WM_CHANGECBCHAIN To hwnd4(as it is current)with wParam as hwnd2 And lParam as hwnd1.

Now hwnd4 in its WM_CHANGECBCHAIN's Handler, checks its wParam ,it is "not of his next" i.e. not of hwnd3.so it bypasses "if statement" and executes "elseif" statement as its Next is not NULL and it is actually hwnd3. so in "elseif" block it sends this same message to hwnd3 as it is.
Now when it arrives to hwnd3,it checks wParam(which is hwnd2) And realizes that it is its Next's handle.

So "if block" gets executed and its Next now becomes the lParam i.e. hwnd1, so hwnd2 smoothly gets out of the chain and chain looks
current clipboard viewer =hwnd4
hwnd4's Next viewer = hwnd3
hwnd3's Next viewer =hwnd1
hwnd1's Next viewer=NULL.

SETTING AND GETTING A STRING USING CLIPBOARD

```
LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
static HGLOBAL hGlobalMem;
TCHAR *pGlobalMemory = NULL ,
*pClipBoardMemory; static TCHAR *ptr=NULL;
HANDLE hClipBoardMemory = NULL;
PAINTSTRUCT ps;
HDC hdc;

switch(iMsg)
```

```
{
case WM_CREATE : if((hGlobalMem = (GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT ,
                  4096)))== NULL)
                 {
                        MessageBox(hwnd , TEXT("Can't Allocate Memory.") , NULL
                                , MB_ICONERROR | MB_OK);
                        break;
                 }
                 pGlobalMemory = (TCHAR *) GlobalLock(hGlobalMem);
                 strcpy(pGlobalMemory,TEXT("Rasesh"));
                 GlobalUnlock(hGlobalMem);
                 OpenClipboard(hwnd);
                 EmptyClipboard();
                 if(SetClipboardData(CF_TEXT , hGlobalMem) == NULL)
                 {
                        MessageBox(hwnd , TEXT("Can't set desired data.") , NULL
                                , MB_ICONERROR | MB_OK);
                        CloseClipboard();
                        break;
                 }
                 CloseClipboard();

                 OpenClipboard(hwnd);
                 if((hClipBoardMemory = GetClipboardData(CF_TEXT)) == NULL)
                 {
                        MessageBox(hwnd , TEXT("Can't access desired data.") , NULL
                                , MB_ICONERROR | MB_OK);
                        CloseClipboard();
                        break;
                 }
                 pClipBoardMemory = (TCHAR *) GlobalLock(hClipBoardMemory);
                 ptr = (TCHAR *) malloc(GlobalSize(hClipBoardMemory));
                 strcpy(ptr,pClipBoardMemory);
                 GlobalFree(hGlobalMem);
                 CloseClipboard();
                 InvalidateRect(hwnd,NULL,TRUE);
break;
case WM_PAINT :  hdc = BeginPaint(hwnd , &ps);
                 SetTextColor(hdc , RGB(255,255,255));
                 SetBkColor(hdc , RGB(0,0,0));
                 TextOut(hdc , 0 , 0 , ptr , strlen(ptr));
                 EndPaint(hwnd , &ps);
break;
case WM_DESTROY : free(ptr);
                  PostQuitMessage(0);
break;
}
return(DefWindowProc(hwnd , iMsg , wParam , lParam));
}
```

SETTING AND GETTING A BITMAP USING CLIPBOARD

```
LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
PAINTSTRUCT ps;
HDC hdc,hMemdc;
HBITMAP hBitmap;
static HBITMAP hBitmapRec;

switch(iMsg)
{
case WM_CREATE : OpenClipboard(hwnd);
                 EmptyClipboard();
                 hBitmap = LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance ,
                                      TEXT("MyBitmap"));
                 if(SetClipboardData(CF_BITMAP , hBitmap) == NULL)
                 {
                        MessageBox(hwnd , TEXT("Can't set desired data.") , NULL
                                  , MB_ICONERROR | MB_OK);
                        CloseClipboard();
                        break;
                 }
                 CloseClipboard();

                 OpenClipboard(hwnd);
                 if((hBitmapRec = GetClipboardData(CF_BITMAP)) == NULL)
                 {
                        MessageBox(hwnd , TEXT("Can't access desired data.") , NULL
                                  , MB_ICONERROR | MB_OK);
                        CloseClipboard();
                        break;
                 }
                 CloseClipboard();
                 InvalidateRect(hwnd,NULL,TRUE);
break;
case WM_PAINT : hdc = BeginPaint(hwnd , &ps); hMemdc
                 = CreateCompatibleDC(hdc);
                 SelectObject(hMemdc , hBitmapRec);
                 StretchBlt(hdc , 0 , 0 , 1024, 768, hMemdc , 0 , 0 , 800 , 600 , SRCCOPY);
```

```
                    DeleteDC(hMemdc);
                    EndPaint(hwnd , &ps);
break;
case WM_DESTROY : free(ptr);
                        PostQuitMessage(0);
break;
}
return(DefWindowProc(hwnd , iMsg , wParam , lParam));
}
```

```
# include <windows.h>
# define UNICODE
# include "mdi.h"

LRESULT CALLBACK WndProc(HWND , UINT , WPARAM , LPARAM);
LRESULT CALLBACK HellowWindowWndProc(HWND , UINT , WPARAM , LPARAM);
LRESULT CALLBACK HellowMDIWndProc(HWND , UINT , WPARAM , LPARAM);
BOOL CALLBACK MyEnumChildProc(HWND , LPARAM);

HMENU hMainMenu , hHellowWindowMenu , hHellowMDIMenu;
HMENU hMainSubMenu , hHellowWindowSubMenu ,
hHellowMDISubMenu; TCHAR szAppName[] = TEXT("MDIClass");
TCHAR szChildOneAppName[] = TEXT("HellowWindowClass");
TCHAR szChildTwoAppName[] = TEXT("HellowMDIClass");

int WINAPI WinMain(HINSTANCE hInstance , HINSTANCE hPrevInstance , LPSTR lpCmdLine , int
nCmdShow)
{
WNDCLASSEX wndclass;
HWND hwnd;
MSG msg;
HACCEL hAccel;
HWND hwndClient;

wndclass.cbSize = sizeof(WNDCLASSEX);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(hInstance , TEXT("MainIcon"));
wndclass.hbrBackground = (HBRUSH)
GetStockObject(BLACK_BRUSH); wndclass.hCursor =
LoadCursor(NULL , IDC_ARROW); wndclass.lpszClassName =
szAppName; wndclass.lpszMenuName = NULL;
wndclass.hIconSm = LoadIcon(hInstance ,
TEXT("MainIcon")); if(RegisterClassEx(&wndclass) == 0) {

        MessageBox(HWND_DESKTOP , TEXT("System Error") , TEXT("ERROR
                RegisterClassEX()") , MB_OK | MB_ICONERROR);
         exit(-1);
}

wndclass.cbSize = sizeof(WNDCLASSEX);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.lpfnWndProc = HellowWindowWndProc;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(hInstance , TEXT("MDIIcon"));
```

```
wndclass.hbrBackground = (HBRUSH)
GetStockObject(BLACK_BRUSH); wndclass.hCursor =
LoadCursor(NULL , IDC_ARROW); wndclass.lpszClassName =
szChildOneAppName; wndclass.lpszMenuName = NULL;
wndclass.hIconSm = LoadIcon(hInstance , TEXT("MDIIcon"));
if(RegisterClassEx(&wndclass) == 0) {

        MessageBox(HWND_DESKTOP , TEXT("System Error") , TEXT("ERROR
                        RegisterClassEX()") , MB_OK | MB_ICONERROR);
        exit(-1);
}

wndclass.cbSize = sizeof(WNDCLASSEX);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.lpfnWndProc = HellowMDIWndProc;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(hInstance , TEXT("MDIIcon"));
wndclass.hbrBackground = (HBRUSH)
GetStockObject(BLACK_BRUSH); wndclass.hCursor =
LoadCursor(NULL , IDC_ARROW); wndclass.lpszClassName =
szChildTwoAppName; wndclass.lpszMenuName = NULL;
wndclass.hIconSm = LoadIcon(hInstance , TEXT("MDIIcon"));

if(RegisterClassEx(&wndclass) == 0)
{
        MessageBox(HWND_DESKTOP , TEXT("System Error") , TEXT("ERROR
                        RegisterClassEX()") , MB_OK | MB_ICONERROR);
        exit(-1);
}

hMainMenu = LoadMenu(hInstance , TEXT("MainMenu"));
hHellowWindowMenu = LoadMenu(hInstance , TEXT("HellowWindowMenu"));
hHellowMDIMenu = LoadMenu(hInstance , TEXT("HellowMDIMenu"));
hMainSubMenu = GetSubMenu(hMainMenu , 0); hHellowWindowSubMenu =
GetSubMenu(hHellowWindowMenu , 2); hHellowMDISubMenu =
GetSubMenu(hHellowMDIMenu , 2);

hwnd = CreateWindow(szAppName , TEXT("MDIDemo") , WS_OVERLAPPEDWINDOW |
                        WS_CLIPCHILDREN , CW_USEDEFAULT , CW_USEDEFAULT ,
                        CW_USEDEFAULT , CW_USEDEFAULT , NULL , NULL , hInstance
                        , NULL);

if(hwnd == NULL)
{
        MessageBox(HWND_DESKTOP , TEXT("System Error") , TEXT("ERROR
                        CreateWindow()") , MB_OK | MB_ICONERROR);
        exit(-1);
}

ShowWindow(hwnd , nCmdShow);
```

```
UpdateWindow(hwnd);
hAccel = LoadAccelerators(hInstance , TEXT("MyAccel"));
hwndClient = GetWindow(hwnd , GW_CHILD);

while(GetMessage(&msg , NULL , 0 , 0))
        {
                if((TranslateMDISysAccel(hwndClient , &msg) == 0) && (TranslateAccelerator(hwnd ,
                hAccel , &msg) == 0))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }

DestroyMenu(hMainMenu);
DestroyMenu(hHellowWindowMenu);
DestroyMenu(hHellowMDIMenu);
return((int) msg.wParam);
} // WinMain()

LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
static HINSTANCE hInst;
static HWND hwndClient;
HWND hwndChild;
CLIENTCREATESTRUCT ccs;
MDICREATESTRUCT mcs;
switch(iMsg)
{
case WM_CREATE : SetMenu(hwnd , hMainMenu);
                        hInst = ((LPCREATESTRUCT)lParam) ->
                        hInstance; ccs.hWindowMenu = hMainMenu;
                        ccs.idFirstChild = 1000;

                        hwndClient = CreateWindow(TEXT("MDICLIENT") , NULL , WS_CHILD |
                                                WS_VISIBLE | WS_CLIPCHILDREN , 0 , 0 , 0 , 0
                                                , hwnd , NULL , hInst , (PSTR) &ccs);
break;
case WM_COMMAND : switch(LOWORD(wParam))
        {
          case IDM_HELLOWWIN : mcs.szClass = szChildOneAppName;
                                mcs.szTitle = TEXT("Hellow Child Window");
                                mcs.hOwner = hInst;
                                mcs.style = 0;
                                mcs.lParam = 0;
                                mcs.x = CW_USEDEFAULT;
                                mcs.y = CW_USEDEFAULT;
                                mcs.cx = CW_USEDEFAULT;
                                mcs.cy = CW_USEDEFAULT;
                                hwndChild = (HWND) SendMessage(hwndClient ,
                                WM_MDICREATE , (WPARAM) 0 ,
                                (LPARAM)(LPMDICREATESTRUCT) &mcs);
```

```
        break;
        case IDM_HELLOWMDI : mcs.szClass = szChildTwoAppName;
                             mcs.szTitle = TEXT("Hellow MDI");
                             mcs.hOwner = hInst;
                             mcs.style = 0;
                             mcs.lParam = 0;
                             mcs.x = CW_USEDEFAULT;
                             mcs.y = CW_USEDEFAULT;
                             mcs.cx = CW_USEDEFAULT;
                             mcs.cy = CW_USEDEFAULT;
                            hwndChild = (HWND) SendMessage(hwndClient  ,
                             WM_MDICREATE , (WPARAM) 0 ,
                            (LPARAM)(LPMDICREATESTRUCT) &mcs);
        break;
        case IDM_HORZTILE : SendMessage(hwndClient , WM_MDITILE ,
                             (WPARAM) MDITILE_HORIZONTAL , 0L);
        break;
        case IDM_VERTTILE : SendMessage(hwndClient , WM_MDITILE ,
                             (WPARAM) MDITILE_VERTICAL , 0L);
        break;
        case IDM_CASCADE : SendMessage(hwndClient , WM_MDICASCADE ,
                             (WPARAM) 0 , 0L);
        break;
        case IDM_ICONARRANGE : SendMessage(hwndClient , WM_MDIICONARRANGE ,
                                           (WPARAM) 0 , 0L);
        break;
        case IDM_CLOSE : hwndChild = (HWND) SendMessage(hwndClient , WM_MDIGETACTIVE ,
                                                        (WPARAM) 0 , 0L);
                         SendMessage(hwndClient , WM_MDIDESTROY , (WPARAM)
                                     hwndChild , 0L);
        break;
        case IDM_CLOSEALL : EnumChildWindows(hwndClient , (WNDENUMPROC)
                                             (int) MyEnumChildProc , 0L);
        break;
        case IDM_EXIT : if(MessageBox(hwnd , TEXT("Do you really want to exit
                          ?") , TEXT("Exit Confirmation") , MB_YESNO |
                          MB_ICONQUESTION) == IDYES)
                        DestroyWindow(hwnd);
        break;
        default : hwndChild = (HWND) SendMessage(hwndClient , WM_MDIGETACTIVE ,
                                                 (WPARAM) 0 , 0L);
                  if(IsWindow(hwndChild))
                     SendMessage(hwndChild , WM_COMMAND , wParam , lParam);
        break;
}//switch
break; // WM_COMMAND
case WM_DESTROY : // first kill all MDI child windows
                  EnumChildWindows(hwndClient , (WNDENUMPROC) MyEnumChildProc ,
                  0L); PostQuitMessage(0);
break;
}//switch iMsg
return(DefFrameProc(hwnd , hwndClient , iMsg , wParam , lParam));
```

```
}

BOOL CALLBACK MyEnumChildProc(HWND hwndSendByOs , LPARAM
lParam) {
        // send kill message to all MDI child windows
        SendMessage(hwndSendByOs , WM_MDIDESTROY , (WPARAM) hwndSendByOs ,
        0L); return(TRUE);
}

LRESULT CALLBACK HellowWindowWndProc(HWND hwnd , UINT iMsg , WPARAM wParam ,
LPARAM lParam)
{
static HWND hwndClient , hwndMain;
HDC hdc;
RECT rc;
PAINTSTRUCT ps;
static int iRed = 255, iGreen = 255, iBlue = 255;
switch(iMsg)
{
        case WM_CREATE : hwndClient = GetParent(hwnd);
                                hwndMain = GetParent(hwndClient);
        break;
        case WM_MDIACTIVATE : if((LPARAM) hwnd == lParam)
                                SendMessage(hwndClient , WM_MDISETMENU , (WPARAM)
                                hHellowWindowMenu , (LPARAM) hHellowWindowSubMenu);
                                    else
                                SendMessage(hwndClient , WM_MDISETMENU , (WPARAM)
                                hMainMenu , (LPARAM) hMainSubMenu);
                                SendMessage(hwndClient , WM_MDIREFRESHMENU ,
                                (WPARAM) 0 , 0L);
                                DrawMenuBar(hwndMain);
        break;
case WM_COMMAND : switch(LOWORD(wParam))
{
        case IDM_RED : iRed = 255; iGreen = 0; iBlue = 0; break;
        case IDM_GREEN : iRed = 0; iGreen = 255; iBlue = 0; break;
        case IDM_BLUE : iRed = 0; iGreen = 0; iBlue = 255; break;
        case IDM_YELLOW : iRed = 255; iGreen = 255; iBlue = 0; break;
        case IDM_MAGENTA : iRed = 255; iGreen = 0; iBlue = 255; break;
        case IDM_CYAN : iRed = 0; iGreen = 255; iBlue = 255; break;
        default : break;
}
        InvalidateRect(hwnd , NULL , TRUE);
break;
case WM_PAINT : hdc = BeginPaint(hwnd , &ps);
                GetClientRect(hwnd , &rc);
                SetBkColor(hdc , RGB(0 , 0 , 0));
                SetTextColor(hdc , RGB(iRed , iGreen , iBlue));
                DrawText(hdc , TEXT("Hellow Window") , -1 , &rc , DT_SINGLELINE
                                | DT_CENTER | DT_VCENTER);
                EndPaint(hwnd , &ps);
break;
```

```
}return(DefMDIChildProc(hwnd , iMsg , wParam , lParam));
}

// window procedure of 2st MDI child window

LRESULT CALLBACK HellowMDIWndProc(HWND hwnd , UINT iMsg , WPARAM wParam ,
LPARAM lParam)
{
static HWND hwndClient , hwndMain;
HDC hdc; RECT rc; PAINTSTRUCT ps; static int iRed = 255, iGreen = 255, iBlue = 255; HBRUSH
hBrush;
switch(iMsg)
{
case WM_CREATE : hwndClient = GetParent(hwnd);
                 hwndMain = GetParent(hwndClient);
break;
case WM_MDIACTIVATE : if((LPARAM) hwnd == lParam)
                      SendMessage(hwndClient , WM_MDISETMENU , (WPARAM)
                      hHellowMDIMenu , (LPARAM) hHellowMDISubMenu);
                          else
                      SendMessage(hwndClient , WM_MDISETMENU , (WPARAM) hMainMenu
                      , (LPARAM) hMainSubMenu);
                      SendMessage(hwndClient , WM_MDIREFRESHMENU , (WPARAM) 0 ,
                      0L);
                      DrawMenuBar(hwndMain);
break;
case WM_COMMAND : switch(LOWORD(wParam))
{
        case IDM_RED : iRed = 255; iGreen = 0; iBlue = 0; break;
        case IDM_GREEN : iRed = 0; iGreen = 255; iBlue = 0; break;
        case IDM_BLUE : iRed = 0; iGreen = 0; iBlue = 255; break;
        case IDM_YELLOW : iRed = 255; iGreen = 255; iBlue = 0; break;
        case IDM_MAGENTA : iRed = 255; iGreen = 0; iBlue = 255; break;
        case IDM_CYAN : iRed = 0; iGreen = 255; iBlue = 255; break;
        default : break;
}
        InvalidateRect(hwnd , NULL , TRUE);
break;
case WM_PAINT : hdc = BeginPaint(hwnd , &ps);
                GetClientRect(hwnd , &rc);
                hBrush = CreateSolidBrush(RGB(iRed , iGreen , iBlue));
                SelectObject(hdc , hBrush);
                PatBlt(hdc , rc.left , rc.top , (rc.right - rc.left) , (rc.bottom – rc.top) , PATCOPY);
                SetBkColor(hdc , RGB(iRed , iGreen , iBlue));
                DrawText(hdc , TEXT("Hellow MDI") , -1 , &rc , DT_CENTER | DT_VCENTER
                        | DT_SINGLELINE);
                EndPaint(hwnd , &ps);
                DeleteObject(hBrush);
break;
}
return(DefMDIChildProc(hwnd , iMsg , wParam , lParam));
}
```

ENTRIES IN ".h" FILE [mdi.h]

```
# define IDM_HELLOWWIN 1
# define IDM_HELLOWMDI 2
# define IDM_CLOSE 3
# define IDM_CLOSEALL 4
# define IDM_EXIT 5
# define IDM_HORZTILE 6
# define IDM_VERTTILE 7
# define IDM_CASCADE 8
# define IDM_ICONARRANGE 9
# define IDM_RED 10
# define IDM_GREEN 11
# define IDM_BLUE 12
# define IDM_YELLOW 13
# define IDM_MAGENTA 14
# define IDM_CYAN 15
```

ENTRIES IN ".rc" FILE [mdi.rc]

```
# include <windows.h>
# include "mdi.h"

MDIIcon ICON music.ico
MainIcon ICON frog.ico

MainMenu MENU
BEGIN
        POPUP "&Create"
        BEGIN
                MENUITEM "New Hellow &World Window" , IDM_HELLOWWIN
                MENUITEM "New Hellow &MDI Window" , IDM_HELLOWMDI
                MENUITEM SEPARATOR
                MENUITEM "&Exit" , IDM_EXIT
        END
END

HellowWindowMenu Menu
BEGIN
        POPUP "&Create"
        BEGIN
                MENUITEM "New Hellow &World Window" , IDM_HELLOWWIN
                MENUITEM "New Hellow &MDI Window" , IDM_HELLOWMDI
                MENUITEM SEPARATOR
                MENUITEM "&Close" , IDM_CLOSE
        END
        POPUP "&Text Color"
        BEGIN
```

```
            MENUITEM "&Red" , IDM_RED
            MENUITEM "&Green" , IDM_GREEN
            MENUITEM "&Blue" , IDM_BLUE
            MENUITEM "&Yellow" , IDM_YELLOW
            MENUITEM "&Magenta" , IDM_MAGENTA
            MENUITEM "&Cyan" , IDM_CYAN
        END
        POPUP "&Window"
        BEGIN
            MENUITEM "&Tile &Horizontally" , IDM_HORZTILE
            MENUITEM "Tile &Vertically" , IDM_VERTTILE
            MENUITEM "&Cascade" , IDM_CASCADE
            MENUITEM "&Icon Arrange" , IDM_ICONARRANGE
            MENUITEM "Cl&ose" , IDM_CLOSE
            MENUITEM "Close Al&l" , IDM_CLOSEALL
        END
END

HellowMDIMenu MENU
BEGIN
        POPUP "&Create"
        BEGIN
            MENUITEM "New Hellow &World Window" , IDM_HELLOWWIN
            MENUITEM "New Hellow &MDI Window" , IDM_HELLOWMDI
            MENUITEM SEPARATOR
            MENUITEM "&Close" , IDM_CLOSE
        END
        POPUP "&Background Color"
        BEGIN
            MENUITEM "&Red" , IDM_RED
            MENUITEM "&Green" , IDM_GREEN
            MENUITEM "&Blue" , IDM_BLUE
            MENUITEM "&Yellow" , IDM_YELLOW
            MENUITEM "&Magenta" , IDM_MAGENTA
            MENUITEM "&Cyan" , IDM_CYAN
        END
        POPUP "&Window"
        BEGIN
            MENUITEM "&Tile &Horizontally" , IDM_HORZTILE
            MENUITEM "Tile &Vertically" , IDM_VERTTILE
            MENUITEM "&Cascade" , IDM_CASCADE
            MENUITEM "&Icon Arrange" , IDM_ICONARRANGE
            MENUITEM "Cl&ose" , IDM_CLOSE
            MENUITEM "Close Al&l" , IDM_CLOSEALL
        END
END

MyAccel ACCELERATORS
BEGIN
        "E" , IDM_EXIT
        "e" , IDM_EXIT
END
```

THE KEYBOARD

Windows 9X applications are dependent on the keyboard heavily for the sake of input. Though mouse is there as secondary mean of input, still keyboard is must. Knowledge of keyboard also requires knowledge of usual ASCII character set, extended ASCII character set, double – byte character set (DBCS) and UNICODE (Supported mainly by NT).

- **Basic Concepts: - As user presses or releases a key on the keyboard, the keyboard hardware drivers passes these keystrokes to OS, OS then saves these keystrokes in the form of *messages*, and then keeps these in the "*system message queue*". Then it transfers these messages, one at a time to the program's message queue (for which the messages are intended). The program then retrieves these messages (by *GetMessage()* function), then translates into actual key codes (by *TranslateMessage()* function) and then pass to the program's window procedure (by *DispatchMessage()* function).**

  Here a question may arise, why OS cannot directly pass keyboard messages to application's message queue? i.e. why the 2 queues are used? (One the *system message queue* and the other is *application message queue*).

  The answer is "*to make synchronization between the speed of user's typing and the capacity of application to receive these keyboard messages*". When a user types on the keyboard faster than the capacity of application's message retrieval, then OS keeps extra keystrokes (that application skipped) into its message queue. This is because user may be working with multiple window applications simultaneously by minimizing one and working with other. So one of user's keystrokes may be for some other application. Thus keeping keystrokes in system message queue, OS ensures the transfer of keystrokes correctly to that application only, which right now has "*input focus*".

- **Focus :- Focus is one of the most important term in window terminology. Because any message from the keyboard will go to that application which right now has input focus, means to the window which is right now "*active*".**

  There is another situation like MDI (Multiple Document Interface) in which there is one application having more that one window. In such cases a single keyboard must shared between all windows of single application. Thus it is better to say that "*a keyboard message will be received by that "Window" (rather that saying "Application") which is right now has input focus or say which is right now active.*"

  The active window (i.e. the window having input focus) can be identified by the highlighted title bar. If active window is a dialog box, then it can be identified by highlighted frame. If active window is minimized, then its activeness can be seen by its highlighted title in taskbar.

  Child window controls are themselves never active if a child window control has input focus, then the active window is parent of that child window control. The input focus on child window control can be seen by a dotted rectangle (around static text of button) or by a caret (edit box) or by a flashing cursor.

  Note that :: *when an active window is minimized, then no window has input focus. Then where do the keystrokes go? Keystrokes keep going to minimize but in different form, means not in that form when the window was not minimized.*

  We ourselves can determine, whether our application window has input focus or not. This is done, by trapping *WM_SETFOCUS* or *WM_KILLFOCUS* messages as seen in the topic of dialog box and child window control. The WM_SETFOCUS indicates that the application's window is receiving focus while the WM_KILLFOCUS indicates that the application's window is loosing focus. In both messages *wParam* gives the window handle of window that gaining focus (in WM_SETFOCUS) and of window that loosing focus (in WM_KILLFOCUS).

⌦ **Keyboard Messages: - Windows OS sends 8 different keyboard messages. But we need not to process every of them. Many of them can be ignored. Window OS handles many keyboard functions on its own.**

E.g. ***(a) All keystrokes that involves "Alt" key.***
(b) Dialog box's keyboard logic
(c) Child window control's keyboard logic etc.

OS handles all these.

Windows OS looks to the keyboard in 2 ways (1) a *key* (2) a *character*.

- **This is because every button on a keyboard is a "key" but this does not mean that every key is a character. E.g. *Alt, Shift, Ctrl, F1 - F12, Escape, Enter*, etc keys generate keystrokes but do not generate any character.**
- **On the other hand *A-Z* keys , *0-9* keys, *special* character keys, are keystrokes as well as characters too.**

So conceptually every button on a keyboard generates keystroke message and character keys generate both *keystrokes messages* and *character messages*.

Thus incase of "*non-character keys*", OS send only keystrokes messages and in case of "*character keys*", OS sends both keystrokes messages and character messages to the WndProc() of the application.

- **The 8 keyboard messages are.**

(1) WM_KEYDOWN
(2) WM_KEYUP } **Non - system keyboard messages.**
(3) WM_CHAR
(4) WM_DEADCHAR

prefixed by "SYS".

(5) WM_SYSKEYDOWN
(6) WM_SYSKEYUP } **System keyboard messages,**
(7) WM_SYSCHAR
(8) WM_SYSDEADCHAR

Usually WM_KEYDOWN is mostly used. Then WM_KEYUP can be ignored. Programmers usually ignore WM_SYSKEYDOWN and WM_SYSKEYUP, because they are mainly concerned with the operating system.

Thus whenever user presses a key on the keyboard OS sends either *WM_KEYDOWN* or *WM_SYSKEYDOWN* and when the pressed key is released, then OS sends either *WM_KEYUP* or *WM_SYSKEYUP*.

As with the logic of typing, usually "*down*" and "*up*" messages occur in pair. But if a key is hold for long time, then system sends multiple *WM_KEYDOWN* or *WM_SYSKEYDOWN* messages, and obviously one and only one *WM_KEYUP* or *WM_SYSKEYUP* when the key is released.

As all queue d messages arrives in application through application's message queue, all have MSG structure format with "*time of key press or release*" in time field of this structure. So the keystroke messages are "*time stamped*". If we need we can get the relative time of key press or key release by using *GetMessageTime()* function.

Within the given 8 keyboard messages the messages with "*SYS*" prefix are system keystroke messages which are important to windows OS than the application. And other messages (not prefixed by "*SYS*") are No-System keystroke messages, which are more important to application than to OS.

Keys pressed in combination with Alt key usually generate system keystroke messages. E.g. menu key.

Applications usually *do not process* these system keystroke messages and just pass them to *DefWindowProc()* for default processing. This is because as explained before windows OS takes care of all these messages.

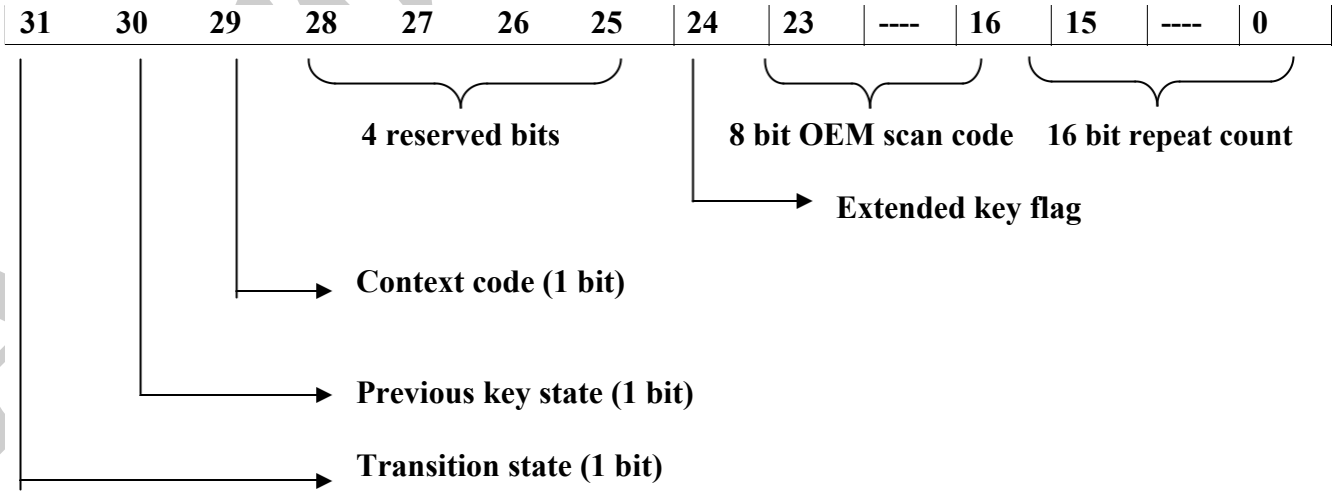
But if you deliberately want to process these messages, in your application, Note that, there "*case statement*" must have *break* and *not return* statement. Because "*break*" statement allows message to go to *DefWindowProc()* while "*return()*" statement directly returns. So by mistake you give return() statement for "*SYS*" message handlers, you may block default Alt key processing of OS *up till your program is running*.

On the other hand, for all those keys that are not used along with Alt, OS sends *WM_KEYDOWN* and *WM_KEYUP* messages when pressed and released respectively. Even it is not at all necessary that you must process these messages in your WndProc() you can ignore them safely, *DefWindowProc()* will take care about them. But if your program is intends to do something in concern with keyboard. E.g. Pressing space bar stops animation event on screen, then you have to process *WM_KEYDOWN* or *WM_KEYUP* in your WndProc().

- **wParam and lParam values of key stroke messages :- Particularly for 4 keystroke messages, *WM_KEYUP , WM_KEYDOWN , WM_SYSKEYUP* and *WM_SYSKEYDOWN*, their wParam is mush more important. The wParam for these messages is called as "*Virtual key code*". This is the identifier of the key that was pressed (for "*DOWN*" messages) or released (for "*UP*" messages). As these codes are defined in "*device independent*" manner, IBM PC may not generate some of them.**

  We already use some of these virtual key codes in scrolling program to give it keyboard support. All these virtual key codes begin with "*VK_* " prefix. For full list of codes see MSDN. Note that, though the full list of codes shows *VK_LBUTTON*, *VK_RBUTTON* and *VK_MBUTTON* codes corresponding to mouse buttons, one program never receives these messages for keyboards if we set *wParam* to these values while using *SendMessage()*. Mouse generates these similar messages on its own.

- **lParam :- For all above 4 keystrokes messages, the 32 bit *lParam* contains 6 fields.**
  (1) Repeat count – 16 bits.
  (2) OEM scan code – 8 bits.
  (3) Extended key flag – 1 bit.
  (4) Context code – 1 bit.
  (5) Previous key state – 1 bit and
  (6) Transition state – 1 bit.
  Remaining 4 bits are reserved.



(1) Repeat count :: It is a 16 bit field running from 0^{th} bit to 15^{th} bit. It indicates the number of keystrokes represented by the message. Most cases it is set to 1.

But if a key is held down for a long time and if WndProc() is not fast enough to process this key down at *Typematic Rate* (which is by default 10 characters/sec), then OS combines several *WM_KEYDOWN* or *WM_SYSKEYDOWN* (whatever you are using) messages into a single message and increases the repeat count's value accordingly.

Note that, this repeat count value is always 1 for *WM_KEYUP* and *WM_SYSKEYUP*.

As typematic rate and WndProc() are not in synchronous mostly with each other, repeat count field is ignored to avoid over scrolling effect.

(2) OEM scan code :: The *BIOS interrupt 16H* passes a value to a program in *AH* register. This value is called as OEM scan code which is given to you in this 8 bit field. This is actually a keyboard scan code generated by keyboard concerned hardware. This value is usually ignored by programmers because there are other better ways to decode keyboard information. Ranges from 16 – 23 bits.

(3) Extended key flag :: This is a one bit flag which is set to 0 or 1. This field is mainly created for IBM's enhanced keyboards. (The enhanced keyboard is that, which has function key and separate cursor pad and separate num pad). This is that 24^{th} bit.

This flag is set to 1 if the key press is concern with following keys…

(1) Ctrl and Alt keys at right side of keyboard.
(2) The cursor keys including Insert and Del.
(3) / and enter keys on numkey pad. and
(4) the num lock key.

Otherwise this flag is set to 0. Programmers usually ignore this flag.

(4) Context code :: This is also a 1 bit code and is the 29^{th} bit in the 32 bit position. (Because 25^{th} to 29^{th} bits are reserved). The value of this field is either 0 or 1. This is 1 if the Alt key is pressed.

This bit is always set to 1 for *WM_KEYDOWN* and *WM_SYSKEYDOWN* messages and is always set to 0 for *WM_KEYUP* and *WM_SYSKEYUP* message. But there are 2 exceptions….

(a) When an active window is minimized, it does not have keyboard focus and OS user "*SYS*" message to avoid the window to process non "*SYS*" keyboard messages. Thus when key is pressed or released, instead of *WM_KEYDOWN* and *WM_KEYUP*, the *WM_SYSKEYDOWN* and *WM_SYSKEYUP* are generated respectively. Hence if Alt key is not presses, then this field i.e. context code is set to 0. (Which should be 1 normally)

(b) On some foreign language keyboards, certain characters are generated by key combination along with Shift or Ctrl or Alt keys and the other key. In such cases, *WM_KEYUP* and *WM_SYSKEYUP* messages set this field to 1. (which should be 0 normally)

(5) Previous key state :: This is again a 1 bit field which occupies 30^{th} bit position in 32 bit lParam. This has value either 0 or 1. For *WM_KEYUP* and *WM_SYSKEYUP* it is always 1 but for *WM_KEYDOWN* or *WM_SYSKEYDOWN* it may be 0 or 1. 1 indicates that this message is second or subsequent message. In other words this indicates that previously the same message had already occurred. This is used while in typematic rate.

(6) Transition state :: This is also a 1 bit field occupies the left most i.e. 31^{st} bit in 32 bit lParam. This has 0 or 1 value. The transition state is 0 when a key is *being pressed* and set to 1 when a key is *being released*. For *WM_KEYDOWN* and *WM_SYSKEYDOWN*, this field is set to 0 and for *WM_KEYUP* and *WM_SYSKEYUP*, this field is set to 1.

- **Shift state :: The wParam and lParam of all four important keyboard messages do not tell anything about the state of shift keys. For this purpose *GetKeyState()* function is used. This is a function used to get the state of shift keys i.e. Shift , Ctrl , And Alt and of the toggle keys i.e. Caps Lock , Num Lock and Scroll Lock.**

e.g. If virtual key code of space bar VK_SPACE is used as parameter to GetKeyState() as ….

GetKeyState(VK_SPACE);

Then…

It will return a *negative value* if the Shift key is down (means if Shift space bar combination is pressed).

Similarly….

GetKeyState(VK_CAPITAL);

Will set the low bit value of return value if the caps lock is toggled ON.

But it is very important to note that, *GetKeyState()* will work only after the arrival of keyboard message from message queue into the WndProc(). Means you should use this function in respective keyboard message handler such as *WM_KEYDOWN*.

e.g. case WM_KEYDOWN :

GetKeyState(-----);

break;

In other words, the *GetKeyState()* function is not a real time keyboard check. You must use it along with keyboard messages. Thus saying….

```
while(GetKeyState(VK_F1) >= 0)
{
```



```
}
```

is totally wrong !!! Because your application must retrieve the keyboard message before *GetKeyState()* can retrieve the state of presses key. If you use above loop, it will jump again to its starting point and application's queue will not be able to retrieve next keyboard message, as your application is still in loop.

- **Character messages : The *TranslateMessage()* function in message loop works for translation of keyboard messages into the character messages.**

  Thus if a key is pressed along with Shift key and if this combination really generates a character, then *TranslateMessage()* first retrieves *WM_KEYDOWN* (or *WM_SYSKEYDOWN*) from *GetMessage()* and from this, *TranslateMessage()* itself generates character message and then keeps this character message in the queue. Obviously message loop immediately after the WM_KEYDOWN (or WM_SYSKEYDOWN) also receives the newly generated and kept character message from the queue.

  There are such 4 character messages…..

For non system characters	**(1) WM_CHAR** **(2) WM_DEADCHAR**	**Generated by *WM_KEYDOWN***
For system Character	**(3) WM_SYSCHAR** **(4) WM_SYSDEADCHAR**	**Generated by *WM_SYSKEYDOWN***

Out of these 4, *WM_CHAR* is mostly of programmer's interest. Whose *wParam* is ASCII code of the generated character and *lParam* is same as that of respective keyboard message's *lParam*. (i.e. *lParam* of *WM_KEYDOWN* or *WM_SYSKEYDOWN*).

***Above 4 character messages arrives to the WndProc() in sandwiched state between the keyboard messages*. Means first keyboard message will arrive followed by the character message, which is then again followed by next keyboard message. (Hence the word "*sandwich*" is used)**

E.g. (1) If caps lock is not toggled ON, means you want to type a small case letter and you press A key to type a, then the message will appear in following order….

WM_KEYDOWN	**:: As you pressed any key its *wParam* will be *VK_A* as *WM_KEYDOWN* produces a valid character, then function *TranslateMessage()* in the message loop, will translate *WM_KEYDOWN* to *WM_CHAR* and will put it in message queue and hence it arrives immediately after *WM_KEYDOWN* its *wParam* will be ASCII of 'a' .**
WM_CHAR (sandwich)	
WM_KEYUP	**As the key is released, this message arrives along with *VK_A* as its *wParam*.**

(2) Now consider following sequence of key pressing you want to type capital A without dealing with caps lock. Obviously you will use shift key combination. So to type capital A….

- **Shift key is pressed, then**
- **'A' key is pressed, then**
- **'A' is released and finally**
- **Shift is released**

Above sequence, first generates *WM_KEYDOWN* for shift key (with *wParam VK_SHIFT*), then will again generate *WM_KEYDOWN* for letter 'A' (with *wParam VK_A*), then will generate *WM_CHAR* (with *wParam* ASCII code of 'A'), which was in response to *WM_KEYDOWN* of 'A'. Then will generate *VK_KEYUP* for key 'A' as 'A' is released (with *wParam VK_A*) and finally again *WM_KEYUP* will be generated as Shift key is released. (with *wParam VK_SHIFT*)

- **In above scenario a question may arises, why there were no *WM_CHAR* messages for Shift keys? The answer is simple and important. *That is because Shift key does not produce any valid character, and hence there are no character messages for Shift key down and up.***

(3) If you hold down 'A' key for dynamic action, then there will be a long series of *WM_KEYDOWN* and *WM_CHAR* messages up till you release the key. When you release it *WM_KEYUP* will arrive as usual. As *lParam* of both *WM_KEYDOWN* and *WM_CHAR* are same, if *lParam* of *WM_KEYDOWN* contains the " *repeat count*" greater than 1, then obviously the *lParam* of *WM_CHAR* will also have the same "*repeat count*."

- **Handling *WM_CHAR* in WndProc() :: When we are interested in ASCII codes of pressed keys by the user, we will write message handler for *WM_CHAR* message, something like follows….**

```
case WM_CHAR :
    switch(wParam)
    {
        case '\b' : // for backspace
```

```
                        // do as need
                break;
                case '\t' : // for tab
                        // do as need
                break;
                case '\n' : // for new line i.e. linefeed
                        // do as need
                break;
                case '\r' : // for "enter" key (i.e. return key)
                        // do as need
                break;
                default : // for any other keys than above 4
                        // do as need
                break;
        }
break;
```

Above code snippet is similar to keyboard processing in 'C' on MS-DOS.

- **Dead character message :- Machine's by default expects "*US*" standard key boards. But different countries in the world may produce their own keyboard design with special meanings to some new keys to some standard US keys as per their nation wide languages.**
  For example, German language supports acute and grave accents ('&') whose keys are in the same position as the +/= key on US standard keyboard. Such character is called as *Diacritic*. Above 2 diacritics can be produced by processing +/= key as alone or along with shift key.
  Such keys are called as "*Dead keys*" and they produce *WM_DEADCHAR* or *WM_SYSDEADCHAR* messages. *Programmers usually do not have any concern with these messages usually do not have any concern with these messages and thus ignore them.*
  If any programmers is interested in these characters and thus in these 2 messages, then too, they not need to write handlers for them. Because *WM_KEYDOWN*, *WM_CHAR* in its *wParam* gives *both the diacritic character and the pressed character*.

- **The caret : Most programmers confuse between caret and cursor. When we type text in the editor, an underline mark or a box mark keeps on advancing as we type and it tells us about the position of the next type able character. *This is called as caret. While cursor is the bitmap image representing the mouse position.***
  In win32 APIs, there are 5 essential functions for caret related programming..

  (1) CreateCaret() :- Creates a caret associated with application's window.
  (2) SetCaretPos() :- Sets position of the caret on application's window.
  (3) ShowCaret() :- Shows the caret.
  (4) HideCaret() :- Hides the caret.
  (5) DestroyCaret() :- Destroying the caret.
  There are yet more 3 functions….
  (6) GetCaretPos() :- Get caret's current position.
  (7) GetCaretBlinkTime() :- Get the caret's blink time.
  (8) SetCaretBlinktime() :- Sets the caret's blink time.

When we deal with fixed size fonts e.g. in command prompt (or dos prompt), then

the caret is usually an underline mark(in Dos and Window) or a box mark (in Unix and its all flavors). This mark is of the size of the fixed font size and its size *cannot* be changed.

But when we deal with advanced window based multi-font editors, then the caret appears as a *vertical line*, which becomes smaller or taller according to the selected font's selected size.

Another very important thing about the caret is that, it is a "*system wide resource*". Means there is only one caret in the whole system. So the function *CreateCaret()* does not mean to create a new caret for your application, but it really means that you want to borrow the caret from the system. Similarly *DestroyCaret()* means giving the caret back to the system.

We might think as a limitation of the system. But it is not ! because as only one window is going to have focus at any time, only one application will require the caret and hence only one caret per system is enough.

That is why *CreateCaret()* is *not* called in *WM_CREATE* and *DestroyCaret()* is *not* called in *WM_DESTROY*.

Instead *CreateCaret()* is called in the *WM_SETFOCUS* message handler and *DestroyCaret()* is called in *WM_KILLFOCUS* message handler. Because WndProc() receives *WM_SETFOCUS* message when your application's window receives input focus. Similarly WndProc() receives *WM_KILLFOCUS* message when your application window looses input focus. Also not that *WM_KILLFOCUS* never arrives to your WndProc() unless it priory has *WM_SETFOCUS* and also remember that there always will be equal number at *WM_SETFOCUS* and *WM_KILLFOCUS* messages over the life time of your allocation's window.

In addition to above rule, we must also remember that *CreateCaret()* "*creates/borrows*" the caret but does not show it. To show it *ShowCaret()* must be called explicitly. Also the caret must be hidden by calling *HideCaret()* whenever we are drawing on our window outside the *WM_PAINT* (i.e. by calling GetDC() and related painting functionality). *Also remember that every HideCaret() call must accompanied by equal number of ShowCaret() calls.*

A SAMPLE CARET PROGRAM

```
LRESULT CALLBACK WndProc(HWND hwnd , UINT iMsg , WPARAM wParam , LPARAM lParam)
{
        static int cxChar , cxCaps , cyChar , cxClient , cyClient;
        TEXTMETRIC tm;
        HDC hdc;
        static HWND hCaret;
        PAINTSTRUCT ps;
        RECT rc;

        switch(iMsg)
        {
                case WM_CREATE : hdc = GetDC(hwnd);
                                 GetTextMetrics(hdc , &tm);
                                 cxChar = tm.tmAveCharWidth;
                                 cyChar = tm.tmHeight + tm.tmExternalLeading;
                                 if(tm.tmPitchAndFamily & 1) cxCaps = (3 * cxChar) / 2;
                                 else cxCaps = cxChar;
                                 ReleaseDC(hwnd , hdc);
```

```
            break;
            case WM_SETFOCUS : CreateCaret(hwnd , NULL , cxCaps , cyChar);
                               SendMessage(hwnd , WM_KEYDOWN , VK_HOME , 0L);
            break;
            case WM_KILLFOCUS : HideCaret(hCaret);
                                DestroyCaret();
            break;
            case WM_PAINT : hdc = BeginPaint(hwnd , &ps);
                            GetClientRect(hwnd , &rc);
                            SetBkColor(hdc , RGB(0,0,0));
                            SetTextColor(hdc , RGB(255,255,255));
                        DrawText(hdc , TEXT("Rasesh") , -1 , &rc , DT_CENTER |
                                        DT_VCENTER |
                            DT_SINGLELINE); EndPaint(hwnd , &ps);
            break;
            case WM_SIZE : cxClient = LOWORD(lParam);
                          cyClient = HIWORD(lParam);
            break;
            case WM_KEYDOWN : switch(LOWORD(wParam))
            {
                    int j;
                    case VK_HOME :  j = ((cxClient / 2) - (cxCaps * (strlen("Rasesh")/2)));
                                    SetCaretPos(j, cyClient / 2);
                                    ShowCaret(hCaret);
                    break;
                    case VK_END : j = ((cxClient / 2) + ( cxCaps * (strlen("Rasesh")/2)));
                                  SetCaretPos(j , cyClient / 2);
                                  ShowCaret(hCaret);
                    break;
            }
            break;
            case WM_DESTROY : PostQuitMessage(0);
            break;
        }
        return(DefWindowProc(hwnd , iMsg , wParam , lParam));
}
```

THE MOUSE

Introduction: - the mouse is a pointing device used by OS as an easy. User friendly way of input. The mouse on the laptop is called as Track Ball. It is an optional device. Means it is not compulsory as keyboard. In other words programmers must provide a keyboard logic for all those things which are done by mouse.(with an exception to mouse dragging as in scrollbar – SB_THUMBTRACK).

Basic: - window can support three types of mice…

a) **one button mouse**
b) **two button mouse**
c) **three button mouse.**

Out of these 3 the one button mouse functionality also can be achieved by joystick or light pen.

Now a day, a pc should have at least 2 buttons mouse as standard. Because the 2nd button is now a day becomes common way of invoking "Context Menu" (or say floating menu) as the standard.

- **if you want to determine whether mouse is parent or not, then you can use GetSystemMetric( ) function with SM_MOUSEPRESENT parameter. The function will return TRUE (or 1) if mouse is present or it will return FALSE (or 0) if mouse is absent.**
- **If you ward to determine how many buttons the mouse has, then you can use same above function with the SM_CMOUSEBUTTONS parameter. It will return number of mouse buttons or 0 if mouse is not installed.**
- **The button pressed by index finger is considered as left mouse button by default. An user may switch the button if it is left-handed. You can programmatically determine whether the button is switched, by calling same above function with SM_SWAPBUTTON parameter.**
- **When a mouse is moved, a small bitmapped image moves accordingly which is called as "mouse cursor". This cursor has a single pixel in it, which has a special meaning and called as Hot Spot. Means if the cursor is of arrow shape, then the top of the arrow is a hot spot. Hot spot points to the exact location on the display.**
- **As stated before, there are many stock cursors (which can be used by GetStockObject ()) and programmer can design its own cursor. The default cursor for the application is provided by the programmer by setting hCursor member of WNDCLASSEX structure by using LoadCursor() function as seen before.**
- **Clicking of mouse button means pressing & releasing a button.**
- **Double-Clicking of mouse button means pressing & releasing a button twice in quick succession.**
- **Dragging of mouse means moving the mouse by holding down a button in pressed state.**

Mouse Message **:- the mouse buttons are identified by APIs as constants declared in header files with LBUTTON , MBUTTON & RBUTTON identifiers with meanings left, middle & right mouse button respectively.**

The wndProc ( ) of an application receives mouse messages when a mouse is either clicked or moved along the window at the application. Important thing to note that, unlike keyboard, mouse messages can be received by application's window even if the specified window does not have focus (i.e. not active).

There are 21 mouse messages. They are primarily divided into 2 groups, 10 each. The 9 messages of each group (out of 10) correspond to 3 buttons i.e. 3 messages for each button and 1 is for mouse movement. The messages are….

A) Client Area Messages.

1. **WM_LBUTTONDOWN**
2. **WM_LBUTTONUP**
3. **WM_LBUTTONDBLCLK**
4. **WM_MBUTTONDOWN**
5. **WM_MBUTTONUP**
6. **WM_MBUTTONDBLCLK**
7. **WM_RBUTTONDOWN**
8. **WM_RBUTTONUP**
9. **WM_RBUTTONDBLCLK**
10. **WM_MOUSEMOVE**

B) Non – Client Area Messages.

1. **WM_NCLBUTTONDOWN**
2. **WM_NCLBUTTONUP**
3. **WM_NCRBUTTONDBLCLK**
4. **WM_NCLBUTTONDOWN**
5. **WM_NCLBUTTONUP**
6. **WM_NCRBUTTONDBLCLK**
7. **WM_NCLBUTTONDOWN**
8. **WM_NCLBUTTONUP**
9. **WM_NCRBUTTONDBLCLK**
10. WM_NCMOUSEMOVE

C) The only remaining 21st message is WM_NCHITTEST.

If you saw above constants, you will realize that all 10 -client area messages with NC prefix (NC- stands for "Non Client"). All these message use LBUTTON, MBUTTON & RBUTTON identifiers in them. The word "DOWN" is for "pressing a button ", word "UP" is for "releasing a button and "DBLCLK" is for "Double Clicking a Button".

The Non Client mouse messages are usually ignored by programmers. They arrive to WndProc ( ) whenever they occur by users respective actions(pressed, released, double clicked or moved) on non-client areas of window such as title bar, menu or scrollbars. By this way they reasonable with "sys" type keyboard messages and are usually handled by DefWindowProc( ). But as "SYS" messages are important to system, these "NC" messages are also important to the system.

A) Client Area Mouse Messages : - These messages arrive when mouse actions are made in the client area of applications window. Note that "MBUTTON" type mouse messages will olive only if you mouse has middle button else not.

Also remember that DBLCLK i.e. double click messages will arrive only when application supports " Double Clicking ". How to make an application Double Click Awarded ? This is done by specifying the window class style CS-DBLCLKS in "style" member of WNDCLASSEX structure in WinMain ( ). This style is used by "Or"ing with CS-HREDRAW | CS _VREDRAW. If you do not include this class style then a double click will arrive you WndProc ( ) as a pair of WM_LBUTTONDOWN & WM_LBUTTONUP twice. As described on earlier page , the double click should arrive in quick succession. Because system has predefined this time and if more time is specified by user during double click, then it will not be valid double click. The system's predefined time of double click is called as " Double-Click Time".

From the conceptual point of view, the double click should perform the same task of single click. Means if we consider the double clicking on a file name in right hand panel of windows explorer, then its first click selects the item (blue colored selection appears) and the next click executes it . but due to the short double click time, user visualizes execution action directly.

*** For all 10 client area mouse messages, lParam gives the value of x & y co-ordinates of mouse cursor position. Where x co-ordinate is obtained by LOWORD macro and y co-ordinate is obtained by HIWORD macro wParam of all these messages indicates the "state of mouse button" and the state of shift keys (i.e. shift & ctrl) pressed along with the mouse button. All these values are predefined and prefixed by MK where MK means " Mouse Key".**

e.g. MK_LBUTTON -> Left mouse button is down.
MK_MBUTTON -> Middle mouse button is down.
MK_RBUTTON -> Right mouse button is down.
MK_SHIFT -> Shift key is down.
MK_CONTROL -> Control key is down.

To use these values you have to use bit masking with wParam.

Eg. –

```
Case WM_MOUSEMOVE:
    If(wParam & MK_LBUTTON)
    {
        // This is checking of whether WM_MOUSEMOVE is generated by pressing left moue
        button as in dragging.
    }
    break;
```

*** One thing is very important to keep in mind about WM_MOUSEMOVE is that , windows does not generate it for every pixel on client area. This number of WM_MOSEMOVE messages depends upon mouse hardware & speed of WndProc ( ).**

*** When an user clicks left mouse button on or in the inactive window, he/she sees that window becomes active. So for him/ her only this single action is felt. But for this system actually does 2 actions, it first actives the window (i.e. sends WM_SETFOCUS) and then sends WM_LBUTTONDOWN.**

In keyboard we saw that, system never sends WM_KEYUP unless the window already has WM_KEYDOWN message. Because keyboard messages are delivered only to that window which is currently active (i.e. which currently has focus). And the reason behind this is quite logical and obvious. But this is not true for mouse. In mouse logic, a window can receive the WM_<> BTTONUP message even though previous message was not WM_<> BUTTONDOWN. This can happen when a mouse button is pressed on one window, then mouse is moved to your window (keeping the button pressed) and then button is real eased on your window. Here your window will receive WM_<> BUTTONUP message without first receiving WM_<> BUTTONDOWN message.

There are 2 exceptions to above rule.

1) **A window (i.e. an application) can capture the mouse so that it will continue to receiving mouse messages even when the mouse is outside the client area of application window.**
2) **A system modal dialog box or message box is on the desktop, then no other program / window can receive the mouse messages except this dialog or message box. This is because, system modal**

dialog & message boxes prevent switching to another window program while the box is active. (the best live example of this is the "shut down " dialog box)

Another important thing to note that, mouse is not like the keyboard. System treats every keystroke as if it is very important to you. While pressing or releasing of mouse button when program is busy (i.e. denoted by hourglass cursor) is safely discarded.

***Processing of Shift Keys : - As described on earlier page using bitwise AND operator with wParam you can get the state of shift or ctrl keys to determine whether a mouse button is pressed along with pressing shift or ctrl key.**

```
        Case WM_<>BUTTONDOWN:
        If(MK_SHIFT & wParam)
        {
            if(MK_CONTROL & wParam)
            {
                /* this 1 & 2 together checks can be used when both shift & ctrl keys are pressed along
                with a mouse button */
            }
            else
            {
                /*this involves only 1 and hence can be used only for shift key is pressed along with a
                mouse button */
            }
        }
elseif( MK_CONTROL & wParam)
{
    /* this can be used when only ctrl key is down along with a mouse button */
}
else
{
    /* neither shift nor ctrl key is down */
}
break;
```

suppose, by any reason, your right mouse button is not working and you want to evaluate left mouse button + shift = Right mouse button , then the message handler will be like follows…

```
case WM_LBUTTONDOWN:
    if(!MK_SHIFT & wParam)
    {
        /* means left mouse button is pressed, without pressing shift key obviously this is left button
        logic */
    }
```

… there is no "break" statement i.e. fall through

```
case WM_RBUTTONDOWN:
/* when left mouse button is pressed along with shift key, it bypasses. Above "if block " and falls through
to message handle of WM_RBTTONDOWN to allow you to emulate right button click */ break;
```

… here is "break" statement , which stops the fall through

You also can use, GetKeyState ( ) function to obtain the state of mouse button or shift or control keys, because virtual key codes also defines virtual keys for corresponding mouse button as VK_LBUTTON, VK_RBUTTON AND VK_MBUTTON. As you know there is VK_SHIFT for shift key And VK_CONTROL for ctrl key. The button or key is said to be "down", when GetKeyState ( ) returns negative value. But again remember, as stated in keybord chapter on earlier pages , GetKeyState ( ) can work only after the message is received by WndProc ( ) from message loop. Hence it most be used in respective message handler. You can not use it in the loop as stated on these pages.

B) NonClient – Area mouse messages :- As stated before , programmers usually ignore these messages and allow them to pass to DefWindowProc ( ). The meaning of these messages are same as those of client area mouse messages with the defference that, they ocure whine mouse is pressed, released or moved on non client areas of window like caption bar, menu and scroll bars.

WParam & lParam of these messages are different than those of client area mouse message. WParam indicates that non-client area where mouse is pressed, released or moved. The identifiers of wParam start with HT_ prefix(means Hit Test). While the lParam gives you x & y co-ordinates of non-client area when used with LOWORD & HIWORD macros respectively. But the most important thing to note that, these co-ordinate are in screen co-ordinate form and not in client co-ordinate form.

Remember that, screen co-ordinate system, assumes its (0,0) at left-top corner of the screen where x co-ordinate increases towords right side and y co-ordinate increases towards down side. While client co-ordinate system assumes its(0,0) at left- top corner of the client area. In default MM_TEXT mapping mode its x & y increase similar to that of screen co-ordinate system. But in other mapping modes location of (0,0) and x & y increment directions differ significantly. Means client area co-ordinate system depends upon Mapping Mode.(we will see about the mapping modes in graphics).

If you want to convert screen co-ordinates to client co-ordinates,you can use ScreenToClient ( ) function and for client co-ordinates to screen co-ordinates conversion, you can use ClientToScreen ( ) both these functions have same prototypes….

ScreenToClient (HWND, LPPOINT);
ClientToScreen(HWND,LPPOINT);

Where first parameter is handle of window and second parameter is address of variable of POINT structure.

POINT structure has 2 members x & y. you specify your convertible co-ordinate of one system in these 2 members and pass the structure variable's address.
Such as….

```
POINT pt;
Pt.x = <some number>;
Pt.y = <some number>;
ScreenToClient(hwnd,&pt);
Or
ClientToScreen(hwnd,&pt);
```

Here it should be noted that, when function returns, it overwrites the passed values by converted values. It is also important that if you specify such a point to ClientToScreen ( ) which is above the windows client area (means point is not in window), then y will be negative on return. Similarly if you specify such a point to ClientToScreen ( ), which is left to the windows client area (means again the point is not in window), then x will be negative on return.

*** The Hit Test Message :- The last remaining mouse message(out of 21) is WM_NCHITTEST , which means "Non-Client HIT TEST". Note that, though we are learning it at the last, actually it is the first mouse message sent by the system before any other "client" or "Non Client" mouse message.**

As programmers ignore this message, this is passed to DefWindowProc ( ). Then DefWindowProc ( ) receives this message and processes it. After completing the processing of WM_NCHITTEST, DefWindowProc ( ) returns and its return value (recall that return value of the DefWindowProc ( ) is of LRESULT type) becomes the wPraram of either "client" or "Non Client" mouse message. So this return value is important and can be any of the following….

HTCLIENT -> Client area.
HTNOWHERE -> Not on any window.
HTTRANSPARENT -> A window covered by another window.
HTERROR -> leads DefWindowProc ( ) to produce beep.
HTSYSMENU -> System menu icon.

Means suppose user presses a left mouse button over client area of a window, then we know that system sends WM_LBUTTONDOWN message to WndProc( ) of window. But this is just the surface analysis , UNDER THE HOOD , actually, when user clicks left mouse button, system first sends WM_NCHITTEST to WndProc ( ) ,WndProc ( ) ignores it and thus it goes to DefWindowProc ( ) , DefWindowProc ( ) processes it and as mouse button was pressed over client area, DefWindowProc ( ) returns with the value HTCLINT, to the system. Now system knows that mouse hit was done on a client area (due to HTCLIENT), so it coverts the mouse co-ordinates (which it gets as lParam of WM_NCHITTEST) which are in screen co-ordinates to client co-ordinates. Then it converts HTCLIENT to MK_<respective button>,here MK_LBUTTON, to wParam and converted mouse co-ordinates to lParam and sends to WndProc ( ) as WM_LBUTTONDOWN message and its message parameters.

In short, we can say that, system uses WM_NCHITTEST message to generate all other 20 mouse message.

Now as R & D but a dangerous test can be made to check the importance at WM_NCHTTEST. We know that WM_NCHITEST is used by the system to generate all other 20 mouse messages. Obviously to allow the system to do this a program must send it back to OS, means to DefWindowProc ( ) Thus a programmer should ignore it by " not Writing its message handler " or " by Writing its message handler but the handler must have break statement so that message will pass to DefWindowProc ( ) ."

If instead of writing " break" statement if we write " return" statement in the message handler of WM_NCHITTEST message, the message will not go to DefWindowProc ( ) , system can not receive and our application will not receive any mouse message (20).

```
Case WM_NCHITTEST:
    Return((LRESULT)HTNOWHERE);
```

By specifying HTNOWHERE in return statement, we are telling that, though mouse is clicked on our window (either on client area or on non-client area), it should be considered as it was clicked " nowhere".

Such type of WM_NCHITTEST handler will disable all client & non-client area mouse messages including one window system menu, sizing buttons, close button & title bar.
This is similar to blocking all keyboard by returning (instead of break) in WM_SYSTEM message handler , as explained on earlier page.

About wParam & lParam of WM_NCHITTEST As it is a system message, system reserves its wParam and it is not used. While lParam is as usual and gives mouse co-ordinates but as like other NC messages they are in screen co-ordinates.

- **How "Double Clicking" on any window's system menu icon closes the window? Double click on window's system menu icon**

WM_NCHITTEST is generated

Passed to defwindowProc ( )

DefWindowPorc ( ) processes it and returns HTSYSMENU as its return value, to OS.

OS takes HTSYSMENU, generates left muse button Double click message WM_NCLEFTBUTTON DBLCLK(as system menu is non- client area), Packs HTSYSMENU as wParam of this message.

As WndProc ( ) ignonres WM_NCLBTTONDBLCLK, It is also passed to DefWindowProc ( ).

OS gets WM_NCLBUTTONDBLCLK from DefWindowProc ( ) and looks at its wParam. As wParam is HTSYSMENU , it generates WM_SYSCOMMAND and puts SC_CLOSE value in its wParam. (recall that, when user selects "close" menu item of system menu , OS generates WM_SYSCOMMAND see earlier page). It puts WM_SYSCOMMADN in message queue.

This WM_SYSCOMMAD is also usually ignored by WndProc ( ) and thus passed to DefWindowProc ( ).

DefWindowProc ( ) Processes WM_SYSCOMMAD and by looking at its wParam, which SC_CLOSE, it sends WM_CLOSE to Window.

When our window receives WM_CLOSE in WndProc ( ) it ignore it (because we usually have message handler of WM_DESTROY and not WM_CLOSE) and passes it to DefWindowProc()

In DefWindowProc ( ), the message handler of WM_CLOSE has a call to DestroyWindow ( ) function with the parameter as our window handle. So call to DestroyWindow ( ) sends WM_DESTROY message to our window.

As our WndProc ( ) always has WM_DESTROY message handler, it appears in WndProc ( ), then PostQuitMessage ( ) is called and our application window terminates as usual.

This is the best example of " A message gives birth to another message".

*** what is Hit Testing : - From Prior point's discussion we know how WM_NCHITTEST can generate other messages. This is because DefWindowProc ( ) itself has message handlers of WM_NCHITTEST, in which it determines the location & button of mouse. This is called as Hit Testing. Our program usually does not deal with such testing. But if we do so, we should have our own WM_NCHITTEST message**

handler and logic of converting screen co-ordinates (given by lParam of WM_NCHITTEST) to client co-ordinates if our hit test deals with client area of our window.

Best example is "Windows Explorer Program Explorer.exe", which determines, which area of client area is clicked, whether the location of click was on a file or directory, if it is then process it as required.

If we need to write our own explorer, then too, we don't need this because SDK gives OS readymade explore control to use directly in our program.

But suppose if we have a window (assuming no menu bar, tool bar) with client area having several columns of alphabetically sorted names of files. As client area has no menu bar and tool bar, first file name begins at 0, 0 of client area. Also assume that the longest file name gives max width of column as cxcolmnwidth. Now if cxclient & cyclient are width & hight of our client area, then….

INumber of files one column = cyclient / cychar;
Where cycha is height of character.

Now suppose , we got left button click and we receive mouse co-ordinates in lParam and we convert them to cxMouse & cyMouse respectively, then we can calculate in which column this is given by

IColumn = cxMouse / cxColumnWidth;

And also we can calculate the relative position from top, i.e. on which file name, he is clicking, by

IformTop = cyMouse / cyChar;

Now the last remaining thing to get the actual file name. Suppose all file names in the client area are stored in an array pszFileNames[], then to get the exact file name, we just need to calculate the index, which can be done by…

Iindex = (Icolumn * iNumber of files InOneColumn) + iFrom Top;

So pszFileName[iIndex] will give the user selected file name.

All above example assumes that all file names are in fixed pitch single font.

The hittesting becomes difficult if we deal with variable size fonts as in offices' s WINWORD or in WORDPAD.

*** Capturing The Mouse : - Sometimes, to avoid the drawing to be drawn out of our widow, as when user clicks a button and alongs mouse out of the windows client area to another window, then as we know, our window can not receive BUTTONUP message as it goes to some other window.**

To avoid this, we need such a solution which will keep the mouse (logically) concerned with our window (though it goes out of bound of our window) and BUTTONUP will be received by our window, even if button was released outside our window. The solution is capturing the mouse.

The win32 API SetCapture (HWND) captures the mouse, and ReleaseCapture ( ) release it.
Now if we Want WM_LBUTTONUP to be received by our WndProc ( ) , even it was release outside the window, we should use SetCapture ( ) in WM_LBUTTONDOWN and should use ReleaseCapture ( ) in WM_LBUTTONUP.

ODBC

- Open Database Connectivity

Contents …

Document Name	**Open Database Connectivity**
Date of Creation	**Sep 25, 2004**
Composed by	**Amol Pawar**

Note: The contents of the document are recompiled and reframed version of MSDN library Oct 2004.

Assumptions:

This session will be conducted with the consideration that we will be working on Windows NT platform.

Prerequisites:

The prerequisite of the sessions are…

- **Comfortable with the usage of Structured Query Language (SQL) with minimal knowledge of insert, delete, update, and select queries.**
- **Acquainted with static & dynamic linking concept (.lib and .dll)**

Introduction

Open Database Connectivity (ODBC) is a widely accepted application-programming interface (API) for database access. It is based on the Call-Level Interface (CLI) specifications from X/Open and ISO/IEC for database APIs and uses Structured Query Language (SQL) as its database access language.

ODBC is designed for maximum *interoperability*—that is, the ability of a single application to access different database management systems (DBMSs) with the same source code. Database applications call functions in the ODBC interface, which are implemented in database-specific modules called *drivers*. The use of drivers isolates applications from database-specific calls in the same way that printer drivers isolate word processing programs from printer-specific commands. Because drivers are loaded at run time, a user only has to add a new driver to access a new DBMS; it is not necessary to recompile or relink the application.

What Is ODBC?

Many misconceptions about ODBC exist in the computing world. To the end user, it is an icon in the Microsoft® Windows® Control Panel. To the application programmer, it is a library containing data access routines. To many others, it is the answer to all database access problems ever imagined.

First and foremost, ODBC is a specification for a database API. This API is independent of any one DBMS or operating system; although this manual uses C, the ODBC API is language-independent. The ODBC API is based on the CLI specifications from X/Open and ISO/IEC.

The functions in the ODBC API are implemented by developers of DBMS-specific drivers. Applications call the functions in these drivers to access data in a DBMS-independent manner. A Driver Manager manages communication between applications and drivers.

Applications that use ODBC are responsible for any cross-database functionality. For example, ODBC is not a heterogeneous join engine, nor is it a distributed transaction processor. However, because it is DBMS-independent, it can be used to build such cross-database tools.

Processing an SQL Statement

To process an SQL statement, a DBMS performs the following five steps:

1. **PARSE Statement - The DBMS first parses the SQL statement. It breaks the statement up into individual words, called tokens, makes sure that the statement has a valid verb and valid clauses, and so on. Syntax errors and misspellings can be detected in this step.**

2. **VALIDATE Statement - The DBMS validates the statement. It checks the statement against the system catalog. Do all the tables named in the statement exist in the database? Do all of the columns exist and are the column names unambiguous? Does the user have the required privileges to execute the statement? Certain semantic errors can be detected in this step.**

3. **GENERATE Access Plan - The DBMS generates an access plan for the statement. The access plan is a binary representation of the steps that are required to carry out the statement; it is the DBMS equivalent of executable code.**

4. **OPTIMIZE Access Plan - The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.**

5. **EXECUTE Access Plan - The DBMS executes the statement by running the access plan.**

Using SQL in host program

There are 3 techniques for using SQL in the host program.

- **Embedded SQL**
  - **Static SQL**
  - **Dynamic SQL**
- **SQL Module**
- **Call Level Interface**

The ODBC Solution

The question, then, is how does ODBC standardize database access? There are two architectural requirements:

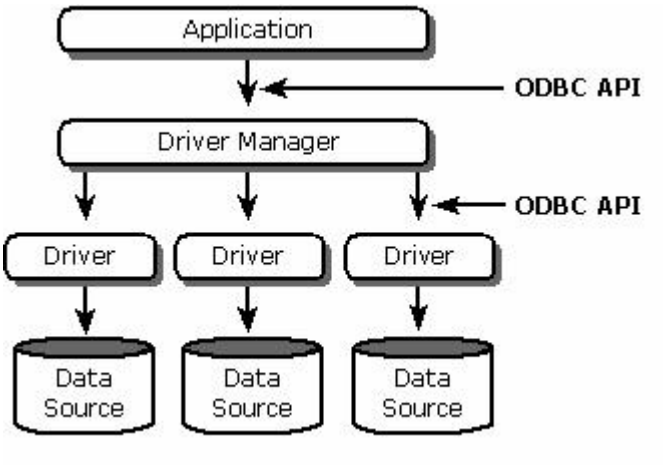
- **Applications must be able to access multiple DBMSs using the same source code without recompiling or relinking.**
- **Applications must be able to access multiple DBMSs simultaneously.**

And there is one more question, due to marketplace reality:
Which DBMS features should ODBC expose? Only features that are common to all DBMSs or any feature that is available in any DBMS?
ODBC solves these problems in the following manner:

- **ODBC is a call-level interface.**
- **ODBC defines a standard SQL grammar.**
- **ODBC provides a Driver Manager to manage simultaneous access to multiple DBMSs.**
- **ODBC exposes a significant number of DBMS features but does not require drivers to support all of them.**

ODBC Architecture

The ODBC architecture has four components:



Application: Performs processing and calls ODBC functions to submit SQL statements and retrieve results.

Driver Manager: Loads and unloads drivers on behalf of an application. Processes ODBC function calls or passes them to a driver.

Driver: Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.

Data source. Consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.

The following illustration shows the relationship between these four components.

The interface between the Driver Manager and the drivers is sometimes referred to as the *service provider interface,* or *SPI*.

The Driver Manager

The *Driver Manager* is a library that manages communication between applications and drivers. For example, on Microsoft® Windows® platforms, the Driver Manager is a dynamic-link library (DLL) that is written by Microsoft and can be redistributed by users of the Microsoft Data Access Software Development Kit (SDK).The Driver Manager exists mainly as a convenience to application writers and solves a number of problems common to all applications. These include determining

- **which driver to load based on a data source name**
- **loading and unloading drivers**
- **calling functions in drivers.**

To see why the latter is a problem, consider what would happen if the application called functions in the driver directly. Unless the application was linked directly to a particular driver, it would have to build a table of pointers to the functions in that driver and call those functions by pointer. Using the same code for more than one driver at a time would add yet another level of complexity. The application would first have to set a function pointer to point to the correct function in the correct driver, and then call the function through that pointer.

The Driver Manager solves this problem by providing a single place to call each function. The application is linked to the Driver Manager and calls ODBC functions in the Driver Manager, not the driver. The application identifies the target driver and data source with a *connection handle*. When it loads a driver, the Driver Manager builds a table of pointers to the functions in that

driver. It uses the connection handle passed by the application to find the address of the function in the target driver and calls that function by address.

For the most part, the Driver Manager just passes function calls from the application to the correct driver. However, it also implements some functions (SQLDataSources, SQLDrivers, and SQLGetFunctions) and performs basic error checking. For example, the Driver Manager checks that handles are not null pointers, that functions are called in the correct order, and that certain function arguments are valid.

The final major role of the Driver Manager is loading and unloading drivers. The application loads and unloads only the Driver Manager. When it wants to use a particular driver, it calls a connection function (SQLConnect, SQLDriverConnect, or SQLBrowseConnect) in the Driver Manager and specifies the name of a particular data source or driver, such as "Accounting" or "SQL Server." Using this name, the Driver Manager searches the data source information for the driver's file name, such as Sqlsrvr.dll. It then loads the driver (assuming it is not already loaded), stores the address of each function in the driver, and calls the connection function in the driver, which then initializes itself and connects to the data source.

When the application is done using the driver, it calls SQLDisconnect in the Driver Manager. The Driver Manager calls this function in the driver, which disconnects from the data source. However, the Driver Manager keeps the driver in memory in case the application reconnects to it. It unloads the driver only when the application frees the connection used by the driver or uses the connection for a different driver, and no other connections use the driver.

Drivers

Drivers **are libraries that implement the functions in the ODBC API. Each is specific to a particular DBMS; for example, a driver for Oracle cannot directly access data in an Informix DBMS. Drivers expose the capabilities of the underlying DBMSs; they are not required to implement capabilities not supported by the DBMS.**

Driver Tasks

Specific tasks performed by drivers include:

- **Connecting to and disconnecting from the data source.**
- **Checking for function errors not checked by the Driver Manager.**
- **Initiating transactions; this is transparent to the application.**
- **Submitting SQL statements to the data source for execution. The driver must modify ODBC SQL to DBMS-specific SQL; this is often limited to replacing escape clauses defined by ODBC with DBMS-specific SQL.**
- **Sending data to and retrieving data from the data source, including converting data types as specified by the application.**
- **Mapping DBMS-specific errors to ODBC SQLSTATEs.**

Driver Architecture

Driver architecture falls into two categories, depending on which software processes SQL statements:

- **File-based drivers.**
- **DBMS-based drivers.**

Data Sources

A *data source* is simply the source of the data. The purpose of a data source is to gather all of the technical information needed to access the data into a single place and hide it from the user. This information is

- **the driver name**
- **network address**
- **network software**

The user should be able to look at a list that includes Payroll, Inventory, and Personnel, choose Payroll from the list, and have the application connect to the payroll data, all without knowing where the payroll data resides or how the application got to it.

Types of Data Sources

There are two types of data sources:

- **machine data sources**
- **file data sources.**

Although both contain similar information about the source of the data, they differ in the way this information is stored.

Using Data Sources

Data sources usually are created by the end user or a technician with a program called the *ODBC Administrator*. The ODBC Administrator prompts the user for the driver to use and then calls that driver. The driver displays a dialog box that requests the information it needs to connect to the data source. After the user enters the information, the driver stores it on the system.

Later, the application calls the Driver Manager and passes it the name of a machine data source or the path of a file containing a file data source. When passed a machine data source name, the Driver Manager searches the system to find the driver used by the data source. It then loads the driver and passes the data source name to it. The driver uses the data source name to find the information it needs to connect to the data source. Finally, it connects to the data source, typically prompting the user for a user ID and password, which generally are not stored.

When passed a file data source, the Driver Manager opens the file and loads the specified driver. If the file also contains a connection string, it passes this to the driver. Using the information in the connection string, the driver connects to the data source. If no connection string was passed, the driver generally prompts the user for the necessary information.

Data Source Example

On computers running Microsoft® Windows NT® Server/Windows 2000 Server, Microsoft Windows NT Workstation/Windows 2000 Professional, or Microsoft Windows® 95/98, machine data source information is stored in the registry. Depending on which registry key the information is stored under, the data source is known as a *user data source* or a *system data source*. User data sources are stored under the HKEY_CURRENT_USER key and are available only to the current user. System data sources are stored under the HKEY_LOCAL_MACHINE key and can be used by more than one user on one machine. They can also be used by systemwide services, which can then gain access to the data source even if no user is logged on to the machine.

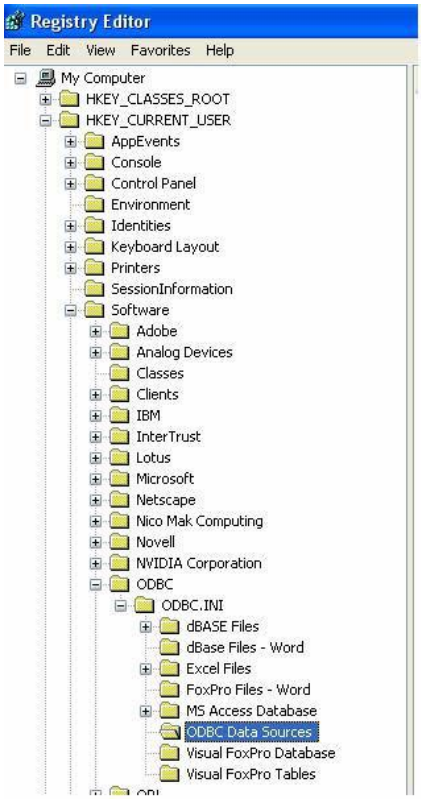
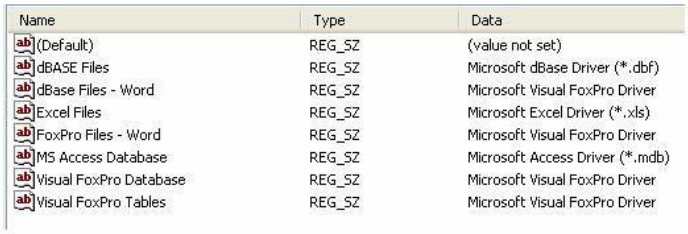
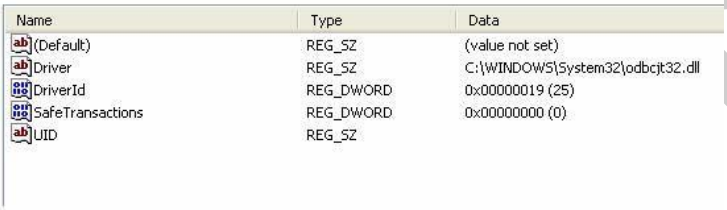


Fig. Dsn1



Name	Type	Data
(Default)	REG_SZ	(value not set)
dBASE Files	REG_SZ	Microsoft dBase Driver (*.dbf)
dBase Files - Word	REG_SZ	Microsoft Visual FoxPro Driver
Excel Files	REG_SZ	Microsoft Excel Driver (*.xls)
FoxPro Files - Word	REG_SZ	Microsoft Visual FoxPro Driver
MS Access Database	REG_SZ	Microsoft Access Driver (*.mdb)
Visual FoxPro Database	REG_SZ	Microsoft Visual FoxPro Driver
Visual FoxPro Tables	REG_SZ	Microsoft Visual FoxPro Driver

Fig. Dsn2



Name	Type	Data
(Default)	REG_SZ	(value not set)
Driver	REG_SZ	C:\WINDOWS\System32\odbcjt32.dll
DriverId	REG_DWORD	0x00000019 (25)
SafeTransactions	REG_DWORD	0x00000000 (0)
UID	REG_SZ	

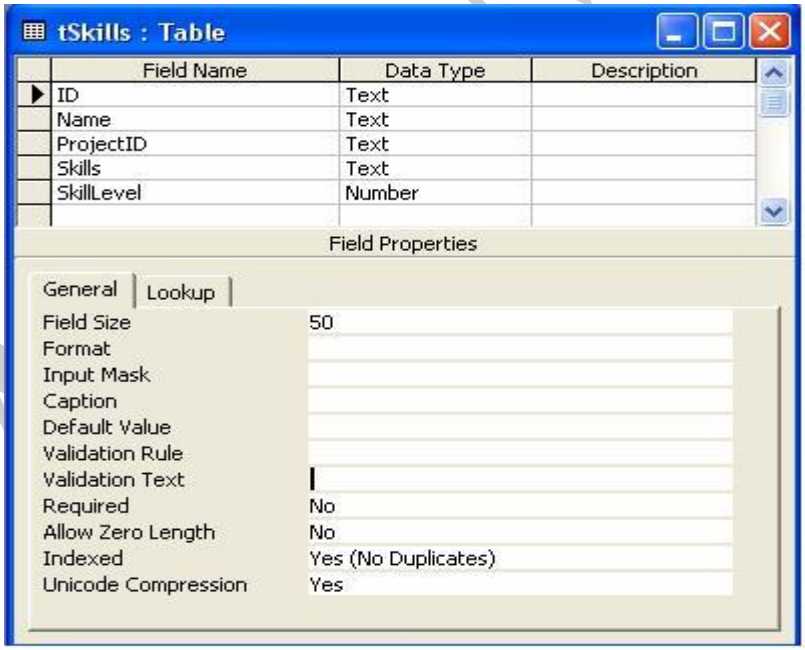
Fig. Dsn3

Creating Access Database

Create an access database with the followings

Database File save Location	**<YourPath>\DbSkills.mdb**
Table Name	**TSkills**

Create a table with structure shown below …



Creating DSN

For our demo, we shall create a machine DSN manually as follows …

- **Go to Control Panel -> Administrative Tools ->Data sources(ODBC)**
- **Now a window "ODBC Data source Administrator" will pop up. You can see different tabs including User DSN.**
- **Click on Add. "Creating New Data Source" window will pop up. Select Microsoft access driver(*.mdb) , and say finish.**
- **Another window "ODBC Microsoft Access Setup" will pop up. Fill the following fields as described below …**

Field	Value	Purpose of the Field
Data Source Name	SkillsDSN	This value will be used in our ODBC program.
Description	Test ODBC program	You can ignore the field. Just the information about this DSN.
Select	<YourPath>\DbSkills.mdb	Path of the Access Database.

- **Finish all windows with all Oks. Our DSN is created.**

ODBC Program – C Code

```
        Application     : ODBC Demo
        Functionality   : Using ODBC C APIs for
                                a. Connecting to Data source
                                b. Executing SQL queries(Insert & select)

                                c. Closing the connection.                              */

#include<stdio.h>
#include<windows.h>
#include<sqlext.h>

#define RetOK 0
#define RetER 1

// global vars
SQLHENV    g_HENV       = SQL_NULL_HENV;
SQLHDBC    g_HDBC       = SQL_NULL_HDBC;
SQLHSTMT g_HSTMT     = SQL_NULL_HSTMT;
SQLRETURN  g_SqlRet;
SQLCHAR * g_DSN                 =(SQLCHAR*)"SkillsDSN";
SQLCHAR * g_UserName =(SQLCHAR*)"";
SQLCHAR * g_Password   =(SQLCHAR*)"";

// function declarations
void fnTraceFormat(const char *, int);
int fnSimpleODBC(void);
```

```
void fnCloseODBC(void);

/* ---------------------------------------------------------------------------------------
        Function        : main()
        Input           : ---
        Return                  : ---
        Flow            : a. Call SimpleODBC() for All ODBC ops
                          b. Call CloseODBC() for closing connection.
 --------------------------------------------------------------------------------------- */
int main(void)
{
        int iRet;

        fnTraceFormat("Starting ODBC Demo Application",RetOK);
        iRet=fnSimpleODBC();
        if(iRet==RetER)
                fnTraceFormat("Program is closing due to the ERROR",RetER);
        else if(iRet==RetOK)
                fnTraceFormat("Program executed successfully",RetOK);
        fnCloseODBC();
        fnTraceFormat("Closing ODBC Demo Application",RetOK);
        printf("\n\n");
        return 0;
}

/* ---------------------------------------------------------------------------
        Function        : fnTraceFormat()
        Input           : a. string to display
                          b. Flag for [ok] or [error] display
        Return                  : ---
        Flow            : a. Print the inputed string                   */
void----------fnTraceFormat(const----------char*iszTraceMsg,intiintFlag)

{
        printf("\n %s",iszTraceMsg);
        if(iintFlag==RetOK)         printf("\n                                [ OK ]");
        else if(iintFlag==RetER)    printf("\n                                [ ERROR ]");
}

/* ---------------------------------------------------------------------------
        Function        : fnSimpleODBC()
        Input           : ---
        Return                  : Suceess or Error Value
        Flow            : a. SQLAllocHandle() for ENV
                          b. SQLSetEnvAttr()
                          c. SQLAllocHandle() for DBC
                          d. SQLConnect()
                          e. SQLAllocHandle() for STMT
                          f. INSERT QUERY
```

```
                                i.  SQLPrepare()
                                ii. SQLExecute()
                            g. SELECT QUERY
                                i.  SQLPrepare()
                                ii. SQLExecute()
                                iii.SQLBind()
                                iv. SQLFetch()
 ------------------------------------------------------------------------------------ */

int fnSimpleODBC(void)
{
        char szID[5]            = "003";
        char szName[50]         = "Anjali";
        char szProjID[10]       = "Streaming";
        char szSkills[50]       = "Network Programming";
        int     iSkillLevel     = 7;
        char szQuery[255];
        SQLINTEGER iNameLen,iSkillsLen,iSkillLevelLen;

// --- Allocate environment handle
        g_SqlRet=SQLAllocHandle(SQL_HANDLE_ENV , SQL_NULL_HANDLE , &g_HENV);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLAllocHandle - ENV", RetER);
                return RetER;
        }
        fnTraceFormat("SQLAllocHandle - ENV", RetOK);

// --- set attributes of environment
        g_SqlRet=SQLSetEnvAttr( g_HENV , SQL_ATTR_ODBC_VERSION ,
                                        (SQLPOINTER)SQL_OV_ODBC3 , SQL_IS_INTEGER);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLSetEnvAttr", RetER);
                return RetER;
        }
        fnTraceFormat("SQLSetEnvAttr", RetOK);

// --- Allocate ODBC Connection Handle
        g_SqlRet=SQLAllocHandle(SQL_HANDLE_DBC , g_HENV, &g_HDBC);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLAllocHandle - HDBC", RetER);
                return RetER;
        }
        fnTraceFormat("SQLAllocHandle - HDBC", RetOK);

// --- establish a coonection with a driver & data source
        g_SqlRet=SQLConnect(g_HDBC , g_DSN , SQL_NTS , g_UserName , SQL_NTS,
```

```
                                g_Password , SQL_NTS );
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet != SQL_SUCCESS))
        {
                fnTraceFormat("SQLConnect", RetER);
                return RetER;
        }
        fnTraceFormat("SQLConnect", RetOK);
// --- allocate statement handle
        g_SqlRet=SQLAllocHandle(SQL_HANDLE_STMT , g_HDBC, &g_HSTMT);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet != SQL_SUCCESS))
        {
                fnTraceFormat("SQLAllocHandle - STMT",
                RetER); return RetER;
        }
        fnTraceFormat("SQLAllocHandle - STMT", RetOK);

// --- Prepare SQL stmt for insert opreation
        sprintf(szQuery,"INSERT INTO tSkills
values('%s','%s','%s','%s',%d)",szID,szName,szProjID,szSkills,iSkillLevel);
        g_SqlRet=SQLPrepare(g_HSTMT, (SQLCHAR *)szQuery, SQL_NTS);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet != SQL_SUCCESS))
        {
                fnTraceFormat("SQLPrepare - Preparing for INSERT Query",
                RetER); return RetER;
        }
        fnTraceFormat("SQLPrepare - Preparing for INSERT Query", RetOK);

// --- SQLExecute
        g_SqlRet=SQLExecute(g_HSTMT);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLExecute - Executing INSERT Query", RetER);
                return RetER;
        }
        fnTraceFormat("SQLExecute - Executing INSERT Query", RetOK);

// --- SQLPrpare for SELECT query
        sprintf(szQuery,"SELECT Name, Skills, SkillLevel FROM tSkills");
        g_SqlRet=SQLPrepare(g_HSTMT , (SQLCHAR *)szQuery , SQL_NTS);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet != SQL_SUCCESS))
        {
                fnTraceFormat("SQLPrepare - Preparing for INSERT Query",
                RetER); return RetER;
        }
        fnTraceFormat("SQLPrepare - Preparing for INSERT Query", RetOK);

// --- SQLExecute for SELECT
        g_SqlRet=SQLExecute(g_HSTMT);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet != SQL_SUCCESS))
```

```
        {
                fnTraceFormat("SQLExecute - Executing SELECT Query",
                RetER); return RetER;
        }
        fnTraceFormat("SQLExecute - Executing SELECT Query", RetOK);

// ---SQLBind for bidning the local vars with the table columns
        // Col : Name
        g_SqlRet=SQLBindCol(g_HSTMT , 1 , SQL_C_CHAR ,
                                (SQLPOINTER)szName, 50 , &iNameLen);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLBind - COL 1", RetER);
                return RetER;
        }
        fnTraceFormat("SQLBind - COL 1", RetOK);

        // Col : Skills
        g_SqlRet=SQLBindCol(g_HSTMT , 2 , SQL_C_CHAR , (SQLPOINTER)szSkills , 50
                                , &iSkillsLen);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLBind - COL 2", RetER);
                return RetER;
        }
        fnTraceFormat("SQLBind - COL 2", RetOK);

        // Col : SkillLevel
        g_SqlRet=SQLBindCol(g_HSTMT , 3 , SQL_C_USHORT , (SQLPOINTER)&iSkillLevel
                                , 0 , &iSkillLevelLen);
        if( (g_SqlRet != SQL_SUCCESS_WITH_INFO) && (g_SqlRet !=
        SQL_SUCCESS)) {
                fnTraceFormat("SQLBind - COL 3", RetER);
                return RetER;
        }
        fnTraceFormat("SQLBind - COL 3", RetOK);

// --- start fetching
        g_SqlRet=SQLFetch(g_HSTMT);
        printf("\n\n\t Name    \t\t Skills    \t\t SkillLevel");
        printf("\n\t ----- \t\t\t ------- \t\t ----------\n");
        while( (g_SqlRet == SQL_SUCCESS_WITH_INFO) || (g_SqlRet ==
        SQL_SUCCESS)) {
                printf("\n\t %s   \t\t %s        \t %d",szName,szSkills,iSkillLevel);
                g_SqlRet=SQLFetch(g_HSTMT);
        }
        printf("\n\t ----------------------------------------------------\n\n");

        fnTraceFormat("SQLFetch - Query", RetOK);
```

```
        return RetOK;
}

/* ---------------------------------------------------------------------------
        Function        : fnCloseODBC()
        Input           : ---
        Return                  : ---
        Flow            : a. SQLFreeHandle() for STMT
                          b. SQLDisconnect() for DBC
                          c. SQLFreeHandle() for DBC
                          d. SQLFreeHandle() for ENV
 --------------------------------------------------------------------------- */
void fnCloseODBC(void)
{
        if(g_HSTMT)
                SQLFreeHandle(SQL_HANDLE_STMT,g_HSTMT);
        if(g_HDBC)  {
                SQLDisconnect(g_HDBC);
                SQLFreeHandle(SQL_HANDLE_DBC,g_HDBC);
        }
        if(g_HENV)
                SQLFreeHandle(SQL_HANDLE_ENV,g_HENV);
}

/* ----------------------------------------------end of application---------------------------------------- */
```

References:

- MSDN Library Ver. October 2001

THE TIMER

- **Introduction: - Using timer in Win32 applications, allows the system to notify the application about completion of certain time interval and the application can take specification after that notification.**

  In other words, if an application sets the timer for 10sec, then it is just like telling the system that, "wake me up after 10 sec." So after 10 sec system sends WM_TIMER message to your application and application can take appropriate action after this message notification.

  So timer is another way of input to the system. It is a very important aspect for the system programmers, service applications and server administration. We can do following things using timer, i.e.

  1. **Animation.**
  2. **Multitasking & Multithreading.**
  3. **Periodic updates of certain records & status, such as automatic saving of user's work, like an edited file, if user forgets to save. This is called as "Autosave".**
  4. **Screensavers – saving monitor's phosphorous burn out.**
  5. **User prompts about updates, task scheduled tasks etc.**
  6. **Automatic starting or stopping certain services.**
  7. **Multimedia refreshing.**
  8. **Automatic shutting hard-disks, monitors on idle machines.**
  9. **System Hibernation.**
  10. **Using system clock in custom applications.**

- Basics :- **A program can tell the OS about setting time interval for it, by using SetTimer( ) function.**

  Its prototype is UINT SetTimer(HWND, UINT UINT ,TIMERPROC) ;

  1ST parameter **:- Handle at window to which we wish the system to send WM_TIMER message.**

  2ND parameter **:- Application defined ID for this timer event. Usually this is #defined macro.**

  3Rd parameter **:- Interval in millisecs i.e. after which system will send your app's window (i.e. 1St parameter) the WM_TIMER message.**

  4 TH parameter **:- Address of the callback function if any. If this is kept NULL, OS will send WM_TIMER to WndProc( ) which will be handled by application by WM_TIMER message handler in WndProc. If a valid function address is given, then instead at sending WM_TIMER to WndProc, OS will call this function.**

  Return value **:- If function succeeds, it returns the same id that you set as 2ND parameter, indicating that timer is successfully set for this id. If function fails it returns value is 0.When program is done with the timer, it should call KillTimer( ) function to give the timer control back to OS. The prototype is BOOL KillTimer(HWND, UINT); where the meanings of parameters is same as that of 1St & 2nd parameters of SetTimer ( ).**

  Actually OS builds its timer functionality by using built in logic OS ROM BIOS. ROM BIOS initializes the timer chip on motherboard, which in turn generates hardware interrupt. This interrupt is called as "Clock tick" or "timer tick" interrupt. This interrupt occurs after every 54.925 milliseconds. When a windows application calls SetTimer( ) , OS takes the 3 rd parameter(i.e. the times interval in milliseconds) and the starts decrementing it by 1. When it

reaches 0, the "time is up" and OS sends WM_TIMER to application, and resets the 3rd parameter back to its original value and starts decrementing it for next interval.

That is why, if WndProc( ) has message handler for WM_TIMER and the tasks (which app wants to do after receiving WM_ TIMER) are lengthy than the possible interval, it is better to kill the timer immediately(i.e. as 1st statement in WM_TIMER message handler), then do the tasks and again call function SetTimer( ) as the last statement in message handler. This avoids arrival of WM_TIMER again, in between the ongoing process of previous WM_TIMER handler logic.

Note that, when specified "time is up", OS keeps WM_TIMER in message queue which then goes to WndProc. So there is no fear of asynchronous, unexpected timer events. Actually timer hardware chip generates interrupts asynchronously but OS translates them to orderly & serialized messages. The interval of each clock tick interrupts i.e. 54.925 ms or 18.2times/sec is an important for two reasons.

1. **A windows application cannot receive WM_TIMER with the faster rate than this using a single timer.**
2. **The value given in the 3rd parameter of SetTimer( ) is always rounded to the nearest integer. Eg: - if this value is 1000, then OS calculates 1000/54.925 which is 18.207 and thus takes 18, and thus WM_TIMER will arrive after 18 clock tics, which is really speaking, actually 989 ms and not the 1000ms as specified.**

The Windows OS's Timer Message are not Asynchronous actually when any hardware interrupt occurs (such as the timer chip's clock interrupt –"clock tick"), the currently running application (any) is interrupted and suspended (i.e. stops its running). The interrupt is handled by interrupt handler functions in kernel of OS and when this function terminates, then the suspended application is restarted.

So the running application , from its perspective ,can say that "I was running smoothly and timer interrupted me in between". So from this view, the timer events are synchronous for that application. But as timer interrupts are occurring after regular interval, we should call them isochronous than asynchronous.

From windows point of view, though timer events are asynchronous (or say isochronous), it sends WM_TIMER synchronously by keeping them in application's message queue. And as WM_TIMER is similar to WM_PAINT, they both are at low priority and thus will go to WndProc( ) only when there are no other messages in message queue.

This tells us another important fact that, if message queue already has some messages and if WM_TIMER arrives now, OS will first send all other messages by FIFO manner and thus application can not gaurantedly say that I am receiving WM_TIMER exactly after the time as I specified in 3rd parameter of SetTimer ( ). So if the 3rd parameter is 1000ms, application expects it either in 1000ms (i.e.) or in 989ms, but this is not guaranteed too. Means if app's is busy in doing something (with respect to some other message handler) for more than 1sec, then it wont receive WM_TIMER during that time.

There is yet another similarity between the WM_TIMER & WM_PAINT, As we know, if many WM_PAINT arrives and loads the message queue, OS does not send them all at once. Rather it combines them and send. Similarly if many WM_TIMER arrives and loaded the message queue (as application was busy in doing something else), then don't expect that when message queue has no messages other than WM_TIMER, they will arrive "all at once". OS will combine multiple WM_TIMER into a single and will send to you. So an application cannot determine the "missing" WM_TIMER events.

3) Methods of Using The Timer:-

Method 1 **:- Set the timer in WinMain( ) or in the WM_CREATE of WndProc( ). Give 2nd parameter a valid id and then use that id switch ---case statements in WM_TIMER message handler. Because OS sends WM_TIMER with its wParam as the timer's id.**

```
#define MY_TIMER 101
        |
        |
case WM_TIMER:
                        switch(wParam)
                        {
                                case MY_TIMER: ---
                                                        break;
                        }
                        break;
```

Then you can call KillTIMER( ) in WM_DESTROY message handler.

- No Timer Available **:- Sometimes SetTimer( ) may return 0 i.e. NULL telling that no timer is currently available. Now if an application is concerned heavily with clock display, it can't do anything without timer. In such cases instead of calling. SetTimer ( ) in Wm_Create, it is better to call it in WinMain( ) before message loop and after CreateWindow ( ). So that if SetTimer( ) fails, application will exit smoothly without entering in message loop and WndProc( ). You can do this by displaying error message box & then using exit. But the message boxes are modal and thus user can not do anything in that application without answering the message box. If user knows that some other application is using timer heavily and hence this application cannot get its timer, then user should be able to close that "timer loaded" application and should come back to this application for retrying again for timer.**
  - **So instead at exiting on failure at SetTimer( ) you can code like follows…**

```
While(SetTimer(….) = = NULL)    //use NULL as last parameter
{
        if(MessageBox(hwnd,TEXT("Toomanyclocks"),TEXT("Error"),MB_ICONEXCLAM
        ATION | MB_RETRYCANCEL) = = IDCANCEL) {

                return(0);   // exit(0);
        }
}
```

here user has opportunity to close "timer loaded" another application and coming back to this application, it can press on "retry" button to see whether OS can give timer for this time or not. This is more elegant error handling in WinMain( ). Here if "Cancel" is pressed, application will exit and if "Retry" is pressed, SetTimer( ) will be called again (due to loop) and system by chance, may give a valid timer to application.

- Method 2 **:- Prior method uses NULL as the last parameter, telling that WM_TIMER will be added in application's message queue and thus will arrive WndProc().**

  But there is another way, declare a callback function globally (along with WndProc) of the type TimerProc(), whose prototype is….

```
Void CALLBACK TimerProc(HWND,UINT,UINT,DWORD);
```

Here the name "TimerProc" is just a place holder, you can, if you wish use another name, such as MyTimerProc().

Its body will be…

```
Void CALLBACK MyTimerProc(HWND hwnd, UINT iMsg, UINT iTimerId,
DWORD dwTime)
{
        //whatever you had done in WM_TIMER
        //message handler in previous example.
        //just use "switch(iTimerId)" in place of switch(wParam);
}
```

The call to SetTimer() in WM_CREATE will now look as follows….

SetTimer(….,….,….,(TimerProc)MyTimerProc);

Specifying a valid callback function name as 4th parameter, forces OS to call this callback function when time is up and thus WM_TIMER will not be kept in message queue & will not be received by WndProc().

- Method 3 **:- This is similar to method 2, just the difference is that, use NULL as hwnd param of SetTimer().**

  If you use this method, your specified 2nd param, i.e. timer id is ignored and system will return it created timer id to you as the return value of SetTimer(). So you can specify 0 as 2nd parameter.

  iTimerID = SetTimer(NULL,0,…,…); obviously when above method is used, the first parameter of KillTimer() also must be NULL.

  And obviously, if your 4th parameter is valid callback function's name, then when OS will call it, it will also send NULL as hwnd parameter of your callback function.

  This method is rarely used.

- How To Use Settings In WIN.INI File ?

The Win.ini file is created during installation of windows OS. This initialization file contains different sections enclosed in […] . one of the section is [intl] which is for International Settings. This section has its subentries concerned with date, time, currency and numbers. The date can be obtained from this section in 3 formats…a. mm-dd-yyyy b. yyyy-mm-dd & c. dd-mm-yyyy. The separator here, we used is " dash", but if we wish, we can use separator like slash, period or any other character we wish. The timer is either in 12hr format or 24hr format. Colon or period is used as separator for hh-mm-ss. We don't need to use File I/O functions to read or to write the Win.ini file. Instead win32 API provides us 5 functions to manipulate the Win.ini file.

1. **GetProfileInt( ):- Gets int value from Win.ini.**

2.

Prototype:- UINT GetProfileInt(LPCTSTR,SPCTSTR,int);

1st para:- section name in win.ini.
2nd para:- key name under above section at win.ini
3rd para:- the default value, if key is not found. You can compare this with the actual return value of UINT type.

Return value: - if function succeeds it returns your required integer value. If fails, then it returns the value of 3rd parameter (as you specified) to indicate you that function is failed.

3. **GetProfileSection() :- Gets all keys & values from your specified section.**

Prototype:-

DWORD GetProfileSection(LPCTSTR,LPCTSTR,DWORD)

1st para:- section name in win.ini.

2nd para:- pointer to a buffer that receives the contents of section.

3rd para:- size of buffer.

Return value:- If functions succeeds it returns number of copied char to your buffer(not including terminating NULL) and if fails due to lack of adequate size of buffer it returns the size of 3rd parameter -2.

4. **GetProfileString():- Gets a string value from win.ini.**

Prototype:

DWORD GetProfileString(LPCTSTR,LPCTSTR,LPCTSTRLPCTSTRDWORD);

1st para:- section name in win.ini. if NULL all sections are considered

2nd para:- key name under above section at win.ini if NULL all keys

3rd para:- default string value you want to get returned , if fails.

4th para:- pointer to a buffer which will receive single or multiple strings according to 1st &2nd parameters.

5th para:- size of the buffer,

Return value:- If functions succeeds it returns number of copied char to your buffer(not including terminating NULL) and if fails due to lack of adequate size of buffer it returns the size of 3rd parameter -2.

5. **WriteProfileSection():- Stores an empty section in win.ini file with all specified keys and their values.**

Prototype:- BOOL WriteProfilesection(LPCTSTR,LPCTSTR);

1st para:- string that contains the section name.

2nd para:- pointer to a buffer that is already filled with valid keynames & their value. It should be in the form key = value. The last string must have an extra NULL character.

Return value:- If succeeds return TRUE, else return FALSE.

6. **WriteProfileString():- Stores a string of key = value pair under specified section.**

Prototype:-

BOOL WriteProfileString(LPCTSTR,LPCTSTR,LPCTSTR);

1st para:- section name in win.ini.

2nd para:- key name under above section, whose value is to be written.

3rd para:- the actual string value. I

Return value:- If succeeds return TRUE, else return FALSE.

By using above functions, we can manipulate win.ini file to fulfill our needs.

Eg: - 1. Suppose our section name is "intl" which is in szSection variable. iDate & iTime are our int variable, then…

```
iDate = GetProfileInt(szSection, TEXT("iDate"),0);
iTime = GetProfileInt(szSection, TEXT("iTime),0);
```

…..will retrieve date & time.

Note:- the stings "iDate & iTime" used as 2nd parameter are not our variables, but are really keys in win.ini file under "intl" section. The letter "i" in iDate & iTime stands for "international" and not for int datatype. While iDate & iTime variables used on left side of = operator are our integer variables.

3. **Under "intl" section of win.ini file, there are keys "sDate" which stands for system Date, "sTime" which stands for sysTime, "51159" which stands for 12hr and "52359" stands for 24hr time.**

   To retrieve values of above 4 keys, we use GetProfileString() as follows…

```
GetProfileString(szSection,TEXT("sDate"),TEXT("/"),szDate,2);
GetProfileString(szSection,TEXT("sTime"),TEXT("/"),szTime,2);
GetProfileString(szSection,TEXT("51159"),TEXT("/"),szAM,5);
GetProfileString(szSection,TEXT("52359"),TEXT("/"),szPM,5);
```

   Where the declared strings are …
   TCHAR szDate [2], szTime[2], szAM[5], szPM[5];
   The last parameters are sizes of respective sting buffer arrays and "/", ":", "AM", "PM" are delimeters for time & date as specified in win.ini file.

- **<u>Windows Standard Time</u> :-**

1. **The function DWORD GetTickCount(void) returns the number of "clock ticks" elapsed since windows session was started. The return value is tick count..**
2. **The function DWORD GetMessageTime(void) returns the time that a message was kept in the message queue. This is the value Kept in Time member of the MSG structure. This time is in millisecs elapsed since windows session was started. In other words, this is time in millisecs since windows was started when the message was sent.**
3. **The function DWORD GetCurrentTime(void) returns the "windows time" or number of clock ticks elapsed since windows session was started. So values returned by 1st & 2nd are equal.**

ATOM AND CHARACTER STRINGS

We already discussed that both server and client identify the data in 3 strings i.e. "***application***", "***topic***" & "***item***". But while write application these are not used as strings but used as ATOMs. ATOM is a WORD type value, which is used to refer a string in case insinuative form (means case is not important). So ATOM is used for character strings particularly when we want to put string data in programs data space.

The declaration of ATOM is done by…

ATOM aAtom;

Now if we have a string pointer like pstr; then it can be added to above declared atom as….

aAtom = AddAtom (pstr);

Here AddAtom adds pstr string to the Atom table and returns a unique identifier in aAtom variable. So prototype of AddAtom() is…..

ATOM AddAtom (LPCSTR);

Note that, each ATOM has its own reference count when a string is added to ATOM table, the reference count is initiated to 1. Now whenever same string is added to Atom by calling AddAtom(), the count goes on incrementing by 1 each time. The Function …

DeleteAtom(aAtom);

Decrement the reference count and when count becomes Zero (0), both the ATOM & string are removed from Atom table. ***(Prototype: ATOM DeleteAtom(ATOM);)***

The Function

aAtom = FindAtom (pstr);

Will return the Atom associated whit the string in a Atom variable, if and only if the string is exists. ***(Prototype: ATOM FindAtom(LPCSTR);)*** OR, otherwise this function returns zero (0) in a Atom variable. This function has no effect on reference count.

Suppose you have an ATOM and you want to see which string it holds, then you should use GetAtomName() function. ***(Prototype: UINT GetAtomName(ATOM, LPCSTR,int);***

Where the first parameter is ATOM variable, string in which you want to find. Second parameter is the string buffer in which the function will return the actual string and the third parameter is the size of the string buffer you had given. Returns numbers of bytes in string

E.g. ATOM aAtom;
Char MyString [255];

// Code
aAtom = AddAtom ("VIJAY");
GetAtomName (aAtom, MyString, 255);

This string will return string "VIJAY" in MyString array that can be checked by MessageBox(). Above 4 functions i.e. ***AddAtom(), DeleteAtom(), FindAtom() & GetAtomName()*** are well suited for a program dealing with string for itself only.

In DDE, these functions cannot be used. Because both server & client are .exe programs and neither of them can have access to each other's data space. (Recall every .exe has it's own 4 GB virtual memory space)

In DDE, these functions cannot be used. Because both server & client are .exe programs and neither of them can have access to each others data space. (Recall every .exe has it's own 4 GB virtual memory space)

So for DDE, we should use the global functions of these 4 functions as .... GlobalAddAtom( ), GlobalDeleteAtom( ), GlobalFindAtom( ), and the GlobalGetAtomName( ). The prototype ,meaning of parameters and returns value are same as that of above four function.....

```
ATOM GlobalAddAtom(LPCSTR);
ATOM GlobalDeleteAtom(ATOM);
ATOM GlobalFindAtom(LPCSTR);
UINT GlobalGetAtomName (ATOM, LPCSTR, int);
```

Note that , as server and client in DDE are both .exe programs, data should be kept at such a memory location (out of both programs 4GB)which is global and thus accessible to both (not only both, but accessible to all windows program that are currently running), So ATOM table created by "Global" function, is stored in shared memory in a dll library within Windows OS. So if server use GlobalAddAtom to add a string in global Atom table, then client can use GlobalGetAtomName( ) to retrieve that string. This is the method by which DDE application knows above there 3 things "application", "topic", and "item".

Rules For using ATOMs in DDE :

1> As ATOMs for DDE are in Global Atom table, it is not good if an atom still required by one program is auidently deleted by another program.

2> It is also not good to keep strings in ATOM when they are not needed.

3> The data & data related other information in DDE is passed using the DDE structure (just describe before). These structure when use must be allocated by proper memory amount either by using global memory functions such as GlobalAlloc( ) or by using memory mapped files.

4> Out of above 2 memory allocation methods, using memory mapped files is dangerous, because in DDE when server keeps data in memory location in one of the DDE structure, client can overwrite the values in data when memory mapped files are used. So avoid usage of memory mapped files.

5> Instead of memory mapped files , use of global memory function GlobalAlloc ( ) is better. Because when server keeps data in memory using this function, the data is "read only" and thus client can read it but cant overwrite it (even flag of GlbalAlloc( ) is set to GMEM_DDESHARE, then too functions ignores it). The client can see the data because OS makes copy of it. So client can see the data, not the actual memory block and thus can not overwrite it.

6> In DDE, usually the program (server or client) that allocates a memory, is also responsible for free the memory. If one program needs that other programs should free the memory, then it explicitly tells other program by using WM_DDE_POKE message. This is the reason that WM_DDE_POKE is rarely used to keep responsibilities to self.

ATOM's And Character Strings :-

We already discussed that both server and client identify the data in 3 strings i.e. "application" , "topic" & "item". But while write application these are not used as strings but used as ATOMs.

ATOM is a WORD type value which is used to refer a string in case insinuative form (means case is not important). So ATOM is used for character strings particularly when we want to put string data in programs data space.

The declaration of ATOM is done by…..

```
ATOM   aAtom;
```

Now if we have a string pointer like pstr; then it can be added to above declared atom as….

```
aAtom = AddAtom (pstr);
```

Here AddAtom adds pstr string to the Atom table and returns a unique identifier in aAtom variable. So prototype of AddAtom ( ) is…..

```
ATOM AddAtom (LPCSTR);
```

Note that, each ATOM has its own reference count when a string is added to ATOM table, the reference count is initiated to 1. Now whenever same string is added to Atom by calling AddAtom ( ), the count goes on incrementing by 1 each time.

The Function …

```
DeleteAtom(aAtom);
```

Decrement the reference count and when count becomes Zero (0), both the ATOM & string are removed from Atom table.(prototype: ATOM DeleteAtom (ATOM);

The Function

aAtom = FindAtom (pstr);

Will return the Atom associated whit the string in a Atom variable, if and only if the string is exists. (Prototype : ATOM FindAtom (LPCSTR): OR ,otherwise this function returns zero (0) in a Atom variable. This function has no effect on reference count.

Suppose you have an ATOM and you want to see which string it holds, then you should use GetAtomName ( ) function. (prototype : UINT GetAtomName(ATOM, LPCSTR,int);

Where the first parameter is ATOM variable , string in which you want to find. Second parameter is the string buffer in which the function will return the actual string and the third parameter is the size of the string buffer you had given. Returns numbers of bytes in string

E.g. ATOM aAtom;

Char MyString [255];

```
// Code
aAtom = AddAtom ("VIJAY");
GetAtomName (aAtom, MyString,255);
```

This string will return string "VIJAY" in MyString array that can be checked by MessageBox ( ).

Above 4 functions i.e. AddAtom ( ), DeleteAtom ( ), FindAtom( ) & GetAtomName( ) are well suited for a program dealing with string for itself only.