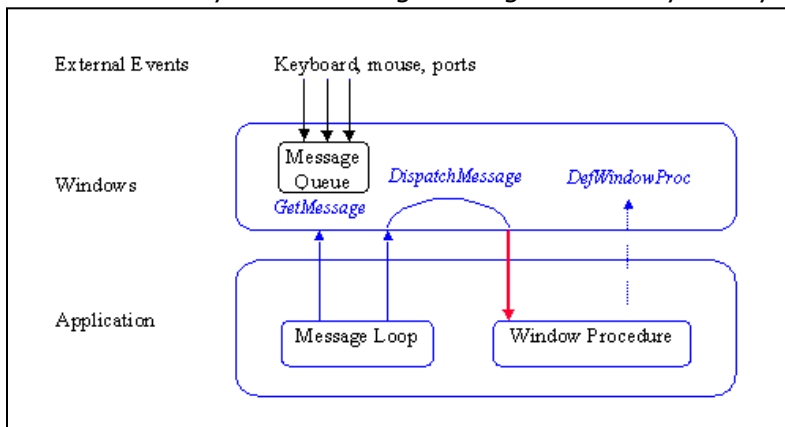


## Window Terms

- An **array dimension** can be **constant** or **dynamic**. In an array declaration, if its dimension is not specified, the array is dynamic. If array content is specified in the declaration, dimension is inferred from it and the array is not dynamic. A dynamic array is created with the **new** operator and deleted with the **[]delete** operator. **Dynamic variables** are also created with **new** and deleted w/ **delete**. Dynamic objects such as **Brushes**, **Fonts**, **Memory blocks** and **Device Contexts** are made with API functions `CreateSolidBrush()`, `CreatePen()`, `CreateFontIndirect()`, `GlobalAlloc()`, `CreateDC()`, etc and are deleted by delete functions `GlobalFree()`, `ReleaseDC()` & `DeleteObject()`. Dynamic objects are created at runtime (body of program) and not at compile time (variable declaration section of program) they are made in the Heap.
- If a function, in which a dynamic object has been created, exits without deleting it and without passing its address to the calling function, the object can never be deleted and also can never be accessed. The memory occupied by it is effectively cordoned off and constitutes a **memory leak**. If the address of the object is passed to the calling function, it can still be deleted and a **memory leak** can be avoided.
- A window is "bound" to a **thread**. This thread "owns" the window and is the only one that can execute the window procedure and process messages for the window. However another window of the same class, using the same window procedure, can be created in another thread with the two windows processing messages concurrently. Thus multiple threads can call the same functions.
- A window has **Input Focus** when it is ready for a keyboard input. Thus a **BUTTON** w/ Focus gets 'pressed' with the space bar, an **EDIT** box will take keyboard i/p or a menu item will get selected with a short cut key.
- **Disabled windows** don't receive mouse/keyboard i/p. **Enabled windows** receive all input. An enabled window becomes **active** (receives Input Focus) once it receives a mouse/keyboard i/p.
- There are two types of windows: modal and modeless. A **modal window** requires the user to close the window before allowing the application to continue. A **modeless window** allows the user to perform another task without closing the window.
- A **keyboard accelerator**, also known as a shortcut key, is a keystroke, or combination of keystrokes, that generates a `WM_COMMAND` message for the window procedure of a window. An ampersand '&' prefixed to an alphabet in a menu or button box caption causes the keyboard key of that alphabet to form an accelerator keystroke combination with the 'Alt' key eg "&Exit" appears as "Exit". The underlined letter is known as an "Underlined Accelerator". See `MakMenu()`.
- The words **thread** and **application**, used below, are synonymous.

## Windows Programming

Windows applications do not make explicit function calls to obtain input, they wait for the system to pass it to them (**event-driven**). The system passes all input (**messages**) for an application to the various windows in it. Each window has a function, called a window procedure, that the system calls whenever it has an input for the window. The window procedure processes the input and returns control to the system. Messages are generated by the system & applications. The system converts i/p



events like mouse/keyboard activity, ports, etc into messages & dispatches them to appropriate windows. eg keyboard messages go to the window currently having **input focus** (active window). Mouse messages are dispatched according to the position of the mouse cursor, usually go to the window that is directly under the cursor (unless some program *captured* the mouse). The system also generates messages in response to changes in the system brought about by an application, such as when an application resizes a window. Applications

can send messages to its own windows or to those in other applications. Messages end up in queues. Windows keeps a **message queue** for every application. Messages are retrieved one-by-one in a **message loop** and the window procedure for the window specified in the message is called.

## Window Creation

- In `MakWin()`, a Window Class is first **registered** by filling a **WNDCLASS** structure & calling **RegisterClassEx()**. One element in this structure is the address of the **window procedure** for this window Class. When a window is created, the OS takes the address of the window procedure in the **WNDCLASS** structure and copies it to the new window's information structure. When a message is sent to the window, the OS calls the window procedure with its address in the window's information structure. Thus the window Class has an information structure and every window has an information structure inherited from the Class to which it belongs.
- **CreateWindowEx()** uses the Class registered to create the Window. When `read()`, for dialog boxes, or `MakMenu()`, for menus is called, `MakWin()` creates a main Window and Child Windows for dialog box controls. While the Window is visible, the Window procedure of the main Window and the default/subclassed Window procedures for child Windows manage all the messages.

Windows are pop-up with a title & thick border. The system does not automatically display it since the `WS_VISIBLE` style is not specified. The window is displayed by `ShowWindow()` in `MsgLoop()`. A Window handle to the main Window is returned for the application to manage the Window.

## Window Procedure

A window procedure is a function that receives and processes messages sent to the window. Every window class has a window procedure used for windows created with that class, to process messages.

The OS sends messages to a window procedure by passing message data as arguments. Unprocessed messages are sent by the procedure to the OS for default processing with a call to `DefWindowProc()`. If the window is sub/super classed, the default window procedure for the window class is executed 1st by `CallWindowProc()`. The window procedure returns the result of message processing.

A window procedure is shared by all windows of a class to process their messages. To identify the target window, a window procedure can examine the window handle passed with a message.

## Window Destruction

An application destroys open windows with `EnbWnd()` before terminating. The window procedure calls `EnbWnd()` in response to user input, such as clicking the **OK, CANCEL, PREV, NEXT, DEFAULT, APPEND, FIND** buttons in a dialog box, the **Exit** menu option or the 'X' mark on the upper RH corner of a window. **DestroyWindow()** invalidates the window handle which can't be used subsequently.

## Owner Window

Windows may have an owner window. When creating a window, the application sets the owner by specifying its window handle in `CreateWindowEx()`. The system uses the owner to determine the position of the window in the Z order so that the window is always positioned above its owner. Also, the system can send messages to the window procedure of the owner, notifying it of events in the window. Windows are automatically hidden, destroyed, enabled or disabled along with the owner. Thus the window procedure requires no special processing to detect changes to the state of the owner.

## Messages

The OS sends messages to window procedures with four parameters: *window handle*, *message identifier* and two *message parameters*. The *window handle* identifies the target window & window procedure. *Message identifier* is an integer constant identifying the purpose of a message. A window uses it to determine how to process the message. *Message parameters* specify data or its location for a window procedure to process the message. The meaning and value of these depend on the message.

## System-Defined Messages

The OS sends/posts *system-defined messages*. eg, A **WM\_PAINT** message tells a window procedure to repaint the window's client area. *Message identifiers* eg **WM\_PAINT** are symbolic constants in a specified range, defined in header files. The prefix of the identifier identifies the type of window or category to which the message belongs eg `CB_??`-Combo box, `BM_??`-Button box, `WM_??` General window etc. General window messages cover information and requests for mouse and keyboard, menu and dialog box, window creation and management and Dynamic Data Exchange(DDE).

## Application-Defined Messages

An application can create messages to be used by its own windows or to communicate with windows in other processes. The message identifiers of these should be within a specified range.

## Message Routing

The system uses two methods to route messages to a window procedure: posting messages to a first-in, first-out(FIFO) queue called a *message queue*, a system-defined memory object that temporarily stores messages, and sending messages directly to a window procedure. Messages posted to a message queue are called *queued messages*. They are primarily the result of user input entered through the mouse or keyboard, such as WM\_MOUSEMOVE, WM\_LBUTTONDOWN, WM\_KEYDOWN, and WM\_CHAR messages. Other queued messages include the timer, paint, and quit messages: WM\_TIMER, WM\_PAINT, and WM\_QUIT. Messages, sent directly to a window procedure, are called *nonqueued messages*.

## Posting - queued messages

The system can display any number of windows at a time. To route mouse, keyboard & port i/p to the appropriate window, the system uses message queues. A single system message queue and one thread-specific message queue are maintained. All threads are created initially without a message queue. A thread-specific message queue is created only when one is needed. Threads that don't use GUI don't get a queue. However if you send a message or peek for a message or create a window or do anything else that requires a message queue, a queue is created, except by PostThreadMessage().

Mouse/keyboard activity causes, a device driver to generate messages & place them in the system message queue. The system removes messages, one at a time, from this queue, examines them to determine the destination window and posts them to the message queue of the thread that created the window. An application can also post messages, to its own queue with **PostMessage()** or to the queue of another with **PostThreadMessage()**. Messages are placed at the end of a message queue(in an **MSG** structure) to ensure extraction in a first FIFO sequence.

Messages are extracted from the thread message queue by a **Message Loop** and directed to the system for dispatch to the appropriate window procedure for processing. Messages like WM\_QUIT do not invoke a window procedure.

**PostMessage()** places a message on the application message queue. If a window handle is not specified(NULL), the message applies to the entire application, instead of a specific window and must be processed in the message loop. An application can post a message to all top-level windows in the system with by specifying HWND\_TOPMOST for the *hwnd* parameter. A common error is to assume that a message is always posted, it isn't when the message queue is full. An application should examine the return value and repost if necessary.

**PostThreadMessage()** fails if the thread doesn't have a queue. The function operates on the queue of another thread that may not have a queue. Other queue functions operate on the queue of the thread making the call or on the queue of a thread that is known to have a queue. Thus, a thread controls whether or not it gets a message queue. If the function created a queue on demand, it would allow one process to create queues in other processes without them knowing.

A thread can use **WaitMessage()** to yield control to other threads when it has no messages in its message queue. The function suspends the thread and does not return until a new message is placed in the thread's message queue.

You can call **SetMessageExtraInfo()** to associate a value with the current thread's message queue. Then call **GetMessageExtraInfo()** to get the value associated with the last message retrieved by **GetMessage()** or **PeekMessage()**.

## Sending - nonqueued Messages

Nonqueued messages are sent immediately to the destination window procedure, by passing the message data as arguments to it, bypassing the system and thread message queues. The system typically sends them to notify a window of events that affect it. Eg, when the user activates a window, the system sends it a series of messages, including WM\_ACTIVATE, WM\_SETFOCUS and WM\_SETCURSOR to notify it of activation, of keyboard input being directed to it and of the mouse cursor having moved into its borders. Nonqueued messages can also result when an application calls certain system functions. Eg, the system sends the WM\_WINDOWPOSCHANGED message after an application uses the SetWindowPos function to move a window. An application uses **SendMessage()**, **BroadcastSystemMessageEx()**, **SendMessageCallback()**, **SendMessageTimeout()**, **SendNotifyMessage()** or **SendDlgItemMessage()** to send messages.

**SendMessage()** sends a message to the window procedure of a given window. It waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate this way. eg, a parent window that has an edit control as its child window can set the text of the control by sending a message to it. The control can notify the parent window of changes to the text that are carried out by the user by sending messages back to the parent.

The **SendMessageCallback** function also sends a message to the window procedure of a given window. However, this function returns immediately. After the window procedure processes the message, the system calls the specified callback function. For more information about the callback function, see **SendAsyncProc()**.

You may want to send a message to all top-level windows in the system. Eg, if the system time is changed and all top-level windows are to be notified a WM\_TIMECHANGE message can be sent by **SendMessage** with the *hwnd* parameter specified as HWND\_TOPMOST. You can also broadcast a message to all applications with **BroadcastSystemMessage**.

**InSendMessageEx()** can be used to determine if a window procedure is processing a message sent by another thread. This is useful when message processing depends on the origin of the message.

## Thread blocking. Deadlock w/ SendMessage

When sending a message to a window the sending function may need to wait for a *reply*. While it is waiting the thread which called the function is said to be *blocked*. This situation prevents the thread from handling any messages for other windows it "owns". When SendMessage sends a message to a window in another thread it waits for a reply which is issued when the receiving thread calls ReplyMessage. An explicit reply is usually not necessary because when a window procedure exits ReplyMessage is automatically called. An explicit ReplyMessage will be necessary to avoid *deadlock* if there is chain of unfinished SendMessage calls that involves more than one thread in a circular fashion. eg: The SendMessage chain: (window A thread 1)-->(B thread 2)--> ... -->(C thread n)-->(D thread 1) will deadlock when C (in thread n) sends a message to D if none of the window procedures have called ReplyMessage. If any of the invoked window procedures (other than D) call ReplyMessage before calling SendMessage the chain is broken. ReplyMessage causes the thread that called SendMessage to continue to run as though the thread receiving the message had returned control. The thread that calls ReplyMessage also continues to run.

Deadlocks with SendMessage() can also occur. If the receiving thread yields control while processing the message, the sending thread cannot continue executing, because it is waiting for SendMessage() to return. If the receiving thread is attached to the same queue as the sender (using journal hooks), it can cause an application deadlock to occur. Note that the receiving thread need not yield control explicitly. To cause a thread to yield control implicitly use: GetMessage(), MessageBox(), PeekMessage() or SendMessage(). A window procedure can determine whether a message it has received was sent by another thread by calling InSendMessageEx(). If this function returns TRUE, the window procedure must call ReplyMessage() before any function that causes the thread to yield control.

### Same-thread sends

SendMessage a thread blocking function is used quite often to send a message to a window in the **same** thread. Under normal rules this function would put the message on a queue and then force the thread to wait for itself to reply. Since it's waiting it can't reply--**deadlock**. In this special case, a same-thread send, Windows will directly call the proper window procedure as if it were a subroutine. In effect this is thread **preemption** for the purpose of responding to new messages. Since it doesn't require the message to be pulled out of a queue, messages can be processed without a message loop. eg CreateWindowEx can invoke your WM\_CREATE code and draw the window frame before your app enters its message loop. Also any window (created in the same thread) can be updated immediately by sending it the appropriate message. This immediate response occurs **only** when a message is sent (with SendMessage) to a window in the **same** thread.

### Forms of message transmission

SendMessage is a thread blocking function. It always waits for a reply. SendMessageTimeout blocks like SendMessage but it unblocks if the receiver takes too long to reply. You specify the timeout interval. SendNotifyMessage is a nonblocking function. It doesn't wait for a reply unless the destination window is in the same thread. SendMessageCallback is another nonblocking function. It also doesn't wait for a reply unless the destination window is in the same thread. You specify a callback routine which will be executed when the receiving thread replies. PostMessage is completely nonblocking. It never waits for a reply. You must target a window. The window determines which thread will receive the message. PostThreadMessage is nonblocking and can't target a window. You must target a thread. PostQuitMessage is nonblocking and posts a WM\_QUIT message to the current thread.

### Message Loop

A thread starts its message loop after creating its main window. A subsequent window of a thread may or may not have a message loop. If it doesn't, the message loop of the main window is used and **DispatchMessage()** calls the window procedure of the proper window because each message in the queue is an **MSG** structure containing the handle of the window to which the message belongs. However it should be noted that if subsequent windows use the message loop of the main window, a WM\_QUIT message posted by the window procedure of any window will cause the main loop to end.

If a dialog box window is created with read() from the WM\_COMMAND message processing code, in the window procedure of the main window, the absence of a message loop in read() would cause it to return, after creating and displaying the window. The window would remain on the screen and user inputs would be saved to the variables being edited but read() will not wait for the dialog box to be destroyed before returning. A message loop inserted in read() would hold the application at that point, taking temporary control of the thread's message queue. When the WM\_QUIT message is posted, the dialog box is destroyed, its message loop is terminated and variables being edited get final values before returning to the window procedure of the main window which then returns to DispatchMessage() in the message loop of the main window.

An application that uses accelerator keys must be able to translate keyboard messages into command messages. To do this, the application's message loop must include a call to **TranslateAccelerator()**. For more information, see Keyboard Accelerators. For dialog boxes, the message loop must include **IsDialogMessage()** so that the dialog box can process ALT and TAB keys.

A very important concept for windows programs is that your window procedure is not magically called by the system or by you directly, *you call indirectly* by calling DispatchMessage(). If we want our application window to process a specific command, or perform an action, we instruct our window to do so by sending it a message, using SendMessage().

1. The message loop starts with `Get/PeekMessage()` to extract a message from the queue into an **MSG** structure. `GetMessage()` waits for a message to appear in the queue, `PeekMessage()` doesn't wait. Both return **TRUE** indicating there is a message to be processed, and the **MSG** structure has been filled. `GetMessage()` returns **FALSE** if it extracts `WM_QUIT`, and a **negative** value if an error occurred. `PeekMessage` returns **FALSE** if it did not get a message. If `GetMessage()` returns **FALSE**, the loop ends. This accomplished with a call to `PostQuitMessage()` in the window procedure to post a `WM_QUIT` message in the queue. When `PeekMessage()` is used the termination of the loop depends on the value of memory variables and not on the `WM_QUIT` message.
2. The message (in the **MSG** variable) is passed to `TranslateMessage()`, This function is called if the thread is to receive character input from the keyboard. The system generates virtual-key messages (`WM_KEYDOWN` and `WM_KEYUP`) each time the user presses a key. A virtual-key message contains a virtual-key code that identifies which key was pressed, but not its character value. **TranslateMessage** translates the virtual-key message into a character message (`WM_CHAR`) and places it in the message queue to be removed in a subsequent iteration of the loop. The virtual-key message remains in the **MSG** structure of the current iteration.
3. `DispatchMessage()` examines the **MSG** structure of the message, extracts the handle of the target window, looks up the Window Procedure for the window and calls it with the window handle, the message identifier, and message parameter fields of the **MSG** structure and not its message time & mouse cursor position fields. An application can retrieve these with `GetMessageTime()` and `GetMessagePos()` in the window procedure. If the window handle is `HWND_TOPMOST`, it sends the message to the window procedures of all top-level windows in the system. If the window handle is `NULL`, it does nothing with the message.
4. In your window procedure you check the message and it's parameters, and do whatever you want with them! If you aren't handling the specific message, you almost always call `DefWindowProc()` which will perform the default actions for you (which often means it does nothing).
5. After processing the message, the windows procedure returns, `DispatchMessage()` returns, and we go back to the start of the loop.

### **Modal Window**

When creating a modal window, the system makes it the active window and it remains so until destroyed or a window in another application is activated. The owner window and its child windows are disabled when creating a modal window and remain so until the modal window is destroyed when they are re-enabled. While creating a modal window, a `WM_CANCELMODE` message is sent to the window (if any) currently capturing mouse input. A window that receives this message should release the mouse capture so that the user can move the mouse in the modal window. Mouse input is lost if the owner fails to release the mouse because the system disables the owner window.

### **Modless Window**

A modeless window doesn't disable the owner window. When creating the dialog box, the system makes it the active window, but the user or the application can change the active window at any time. If the modeless window does become inactive, it remains above the owner window in the Z order, even if the owner window is active.

### **Subclassing**

Substitution of the default window procedure(message handler) address in the information structure with another(subclass function) address is Subclassing. The subclass function then intercepts messages destined for the window's default window procedure. The subclass function can (1) pass the message to the default window procedure; (2) modify the message and pass it to the default window procedure; (3) not pass the message. The subclass function can process the message before, after, or both before and after passing the message to the default window procedure. The two types of subclassing are *instance subclassing* and *global subclassing*. An application cannot subclass a window or Class that belongs to another process(thread) and *the subclass procedure shouldn't destroy the original behavior of the window*.

## Why Subclass

Subclassing a window is an effective way to modify its behavior without redeveloping it eg The child windows(controls) of a dialog box (button, edit, list, combo, static, group and scroll bar boxes). For example, if a multiline edit control is included in a dialog box and the user presses the ENTER key, the dialog box closes. By subclassing the edit control, an application can have the edit control insert a carriage return and line feed into the text without exiting the dialog box. An edit control does not have to be developed specifically for the needs of the application.

## Global Subclassing

Global subclassing replaces the address of the window procedure in the Information Structure of a window Class. Subsequent windows created with the Class have the new window procedure's address.

## Instance Subclassing

Instance subclassing replaces the address of the default window procedure in the individual window's Information Structure with the address of a user defined function. Thus, only the messages of a particular window instance are sent to the new window procedure.

The **SetWindowLong** function is used to subclass an instance of a window. The application subclassing the window calls **SetWindowLong** with the handle to the window the application wants to subclass, the **GWL\_WNDPROC** option, and the address of the subclass function. **SetWindowLong** returns a **DWORD**, which is the address of the original window procedure for the window. The application must save this address to pass the intercepted messages to the original window procedure and to remove the subclass from the window. The original window procedure is called by calling **CallWindowProc** with the address of the procedure and the *hWnd*, *Message*, *wParam*, and *lParam* parameters it receives from Windows. The application removes the subclass from the window by calling **SetWindowLong** again with the address of the original window procedure, the **GWL\_WNDPROC** option and the handle to the subclassed window.

The following code subclasses and removes a subclass to an edit control:

```
FARPROC OldProc;
HWND hWnd;

hWnd = CreateWindowEx(); // Create window
OldProc=(FARPROC)SetWindowLong(hWnd,GWL_WNDPROC,(DWORD)SubClassFunc); // Subclass it
SetWindowLong(hWnd, GWL_WNDPROC, (DWORD) OldProc); // Remove the subclass
LONG FAR PASCAL SubClassFunc(HWND hWnd,WORD Message, WORD wParam, LONG lParam){ // subclass function
    if ( Message==WM_GETDLGCODE ) return DLGC_WANTALLKEYS; //When the focus is in an edit,
    return CallWindowProc(OldProc,hWnd,Message,wParam,lParam); //Disable default ENTER key action.
}
```

## Potential pitfalls

- 1>The subclass function should not break the original behavior of the window.
- 2>A subclass function should not use the extra window bytes or the Class bytes for the window.
- 3>Subclasses must be removed in reverse of the order in which they were performed.

## Superclassing

Superclassing creates a new window Class based on a pre-registered window Class(base-Class). The superclass has its own window procedure to take the same actions as a subclass function and also handle window creation messages(WM\_NCCREATE, WM\_CREATE, etc) from Windows, which the subclass function can't(since subclassing occurs after creation). The superclass function can process these messages, but it must also pass them to the base-Class window procedure for it to initialize.

The application calls **GetClassInfo** to fill a **WNDCLASS** structure with the values from the base Class. **GetClassInfo** does not return the **lpzMenuName**, **lpzClassName**, or **hInstance** fields of the structure. The application sets the **hInstance** field of the **WNDCLASS** structure to the instance handle of the application, the **lpzClassName** field is set to the name of this superclass. If the base Class has a menu, a new menu must be supplied and its name string placed in the **lpzMenuName** field. If the superclass does not intend to process the WM\_COMMAND message, the new menu must have the same menu IDs as the base Class's menu. **GetClassInfo** fills the **lpfnWndProc** field with the original Class's window procedure. The application must save this address so that it can pass messages to the original window procedure with a call to **CallWindowProc** and replace it with the address of a user defined function. The application can modify any other fields in the **WNDCLASS** structure as required. The application can add to both the extra Class bytes and the extra window bytes because it is registering a new Class. The application must follow two rules when doing this: (1) the original extra bytes for both the Class and the window must not be touched by the superclass for the same reasons that an instance subclass or a global subclass should not touch these extra bytes; (2) if the application adds extra bytes to either the Class or the window instance for the application's own use, it must always reference these extra bytes relative to the number of extra bytes used by the base Class. Because the number of bytes used by the base Class may be different from one version of the base Class to the next, the starting offset for the superclass's own extra bytes is also different from one version of the base Class to the next. After the **WNDCLASS** structure is filled, the application calls **RegisterClass** to register the new window Class. Windows of this Class can now be created and used.

### Example

Create a window for a preregistered, "#32770" window Class, for a dialog box as follows:-  
hWnd=CreateWindowEx(dwStyleEx,"#32770","The dialog title",dwStyle,x,y,cx,cy,parentHandle,0,gInstance, 0);

This uses the default window procedure, over which we have no control. To gain control, subclass the window to our own window procedure as follows:-

```
SetWindowLong( hWnd, GWL_WNDPROC, (long)YourWndProc );
```

Even now, the default window procedure is used until CreateWindowEx() returns and messages like WM\_CREATE, fired by the window manager during this time are missed.

We therefore superclass the window Class by copying it, modifying the copy to point to our own window procedure and registering the copy w/ a new name. Finally CreateWindow() is called to use the new Class.

```
WNDCLASSEX wndclass = { 0 };
if (GetClassInfoEx( NULL, "#32770", &wndclass)) { //Make a copy of the system's dialog
Class(#32770)
    wndclass.lpfnWndProc = YourWndProc; //our own WndProc
    wndclass.lpszClassName= "MyModelessDialog"; //new Class name
    RegisterClassEx(&wndclass); //register the new Class
}
hWnd=CreateWindowEx(dwStyleEx,"MyModelessDialog","The dialog
title",dwStyle,x,y,cx,cy,parentHandle,0, gInstance,0 );
```

## Preregistered Window Classes

| Windows Class name | User Interface element type                             |                                      |                          |
|--------------------|---|--------------------------------------|--------------------------|
| ListBox            | List boxes  | SysHeader32                          | List view headers        |
| Button             | Push buttons, radio buttons, check buttons, group boxes | SysListView32                        | List view controls       |
| Static             | Labels  | SysTreeView32                        | Tree view controls       |
| Edit               | Text boxes  | SysDateTimePick32 (versions 5 and 6) | Date and/or time picker  |
| ComboBox           | Combo boxes, drop-down lists                            | SysIPAddress32                       | IP address controls      |
| ScrollBar          | Scroll bars   | tooltips_class32                     | ToolTips                 |
| #32768             | USER menus  | ToolbarWindow32                      | Toolbars                 |
| #32770             | USER dialog boxes                                       | RICHEDIT, RichEdit20A, RichEdit20W   | Text fields              |
| #32771             | Alt-tab window  | SysMonthCal32 (versions 5 and 6)     | Month calendar           |
| msctls_statusbar32 | Status bars   | SysAnimate32                         | Animation control        |
| msctls_progress32  | Progress bars   | SysTabControl32                      | Tab control              |
| msctls_hotkey32    | Hot key controls  | msctls_updown32                      | Up-down or spin controls |
| msctls_trackbar32  | Trackbars, sliders                                      |                                      |                          |

## Windows functions

### DispError

`void DispError(void);`

#### Return Value

None

#### Description

Before a window API returns it sets the thread's last-error code value using window API `SetLastError()`. The code is translated to a message and displayed in a pop-up message box. The function must be called immediately after a window API call, before it is overwritten by the next API call.

#### Parameters

None

### PutBmp

`LRESULT PutBmp(HWND hWnd, DWORD w, DWORD d, char *fnm);`

#### Return Value

0

#### Description

Reads the Bit Map file specified by *fnm* and inserts it at the centre of window *hWnd* after scaling to the size specified by *w,d*. The background color of the window and image must be matched manually. Uses window APIs `BeginPaint()`, `LoadImage()`, `CreateCompatibleDC()`, `SelectObject()`, `BitBlt()`, `DeleteDC()`, `DeleteObject()` and `EndPaint()`.

#### Parameters

*hWnd* Window Handle

*w,d* Width & depth, in pixels, to which image must be scaled

*fnm* name & path of .BMP file eg "D:\\Robocup IP\\icon.bmp"

### GetSize

`RECT GetWsize(DWORD x, DWORD y, DWORD cx, DWORD cy, DWORD cen);`

#### Return Value

In pixels, window top LH co-ordinates, width and depth in a predefined, structure variable, of type `RECT`, with members: `left` (window top LH x co-ordinate), `top` (window depth), `right`(window top LH y co-ordinate), `bottom`(window width).

#### Description

Return Window Co-ordinates in pixels, corrected to fit the display size & centered if required.

#### Parameters

*x,y* Co-ordinates of a windows top LH corner in pixels

*cx,cy* Width(*cx*) and depth(*cy*) of a window in pixels

*cen* The window is centered(1) or not(0)

#### Processing

Declare return variable 'rvl'. Obtain screen co-ordinates with window API `GetSystemMetrics()`. Input co-ordinates are centered if *cen*=1 and the result placed in 'rvl'. If the resulting co-ordinates are impossible to fit on the screen 'rvl' is maximized to fit the screen.

# EnbWnd

**void EnbWnd(HWND hWnd,DWORD mdl,DWORD dst)**

## Parameters

*mdl* Modal(1) or Modeless(0) window. Specifies whether windows opened prior to the last(active) window are to be enabled or disabled. If *mdl=0*, they are enabled and If *mdl=1*, disabled.

*dst* If the last(active) window is to be destroyed, *dst* must be made 1.

*hWnd* Handle of window last active window.

## Description

Enable or disable all windows except the last window active in the **List of Windows** for mouse & keyboard i/p.

## Processing

If a window in the **List of Windows** is being disabled, the system sends a WM\_CANCELMODE message to its window procedure. If the enabled/disabled state of a window is changing, the system then sends a WM\_ENABLE message. When the state of a window changes, the state of its child windows is implicitly changed, although they are not sent a WM\_ENABLE message. The messages are sent directly to the window procedure of the window, bypassing the message loop(nonqueued), by window API SendMessage().

A window must be enabled before it can be activated. For example, if an application is displaying a modal dialog box and has disabled its parent window, the application must enable(*mdl=0*) the parent window before destroying the dialog box(*dst=1*). Otherwise, another window will receive the keyboard focus and be activated instead of the parent window.

A window is destroyed with a call to window API DestroyWindow(*hWnd*), before which its parent window is re-painted with a call to window API InvalidateRect(pRnt,0,1). Where 'pRnt' is the handle of the parent obtained with a call to window API GetWindow(*hWnd,GW\_OWNER*)

A **List of Windows**, arranged according to the order of opening(in time and not visual position), is maintained by windows. A call to window API GetDesktopWindow() returns the handle of the desktop window to *hDsk*. The desktop window covers the entire screen(windows desktop) all icons and other windows are painted on this. Window API GetWindow(*hDsk,GW\_CHILD*) returns to *hW*, the handle of the 1st child window opened under *hDsk*. Subsequent calls to GetWindow(*hW,GW\_HWNDNEXT*), returns the next window opened under *hDsk*, to *hW*. The handle of each window, except the last, obtained in this way, is used to change its state, by setting a new window style, with a call to window API SetWindowLong().

# MakWin

**HWND MakWin(HWND *prnt*,WNDPROC \**proc*,char \**titl*,DWORD *colr*,DWORD *cen*,void \*\**hMem*,  
DWORD *x*,DWORD *y*,DWORD *cx*,DWORD *cy*, DWORD *mdl*, DWORD *lpw*,  
DBOX \**hgb*);**

## Return Value

When successful, handle to the Window created otherwise NULL

## Parameters

*prnt* Handle to Parent window(not reqd since Window is Pop-up type)  
*proc* Pointer to Window Procedure(Message Handler) for the Window.  
*titl* Pointer to string to be displayed as a title for the Window.  
*colr* Window background color passed as an integer returned by RGB(r,g,b). Where r,g & b are intensities of Red, Blue and Green components in the color, in the range 0 to 255  
*cen* Indicates that the window is to be centered in the display(1) or not(0). When *cen*=1, *x* & *y* inputs are ignored.  
*hMem* Pointer to memory block, type casted to a pointer to an array of pointers to memory variables declared in the calling program, requiring access in the window procedure(*proc*). A memory block is created and *hMem* assigned with the use of window API GlobalAlloc() as follows:-  
*hMem*=(void\*\*)GlobalAlloc(GMEM\_ZEROINIT,N\*sizeof(LPVOID));  
'N' is the number of memory variables+1. The extra(+1) element is for the 0th element, which is assigned the handle of the window created(the return value).  
*x,y* Top LH corner co-ordinates of the window in pixels. Unused when *cen*=1  
*cx,cy* Width & depth of the window *resp* in pixels.  
*mdl* Create Modal(1),Modeless(0) window  
*lpw* Element count of *hgb*[ ] array  
*hgb* Array of child window details in the parent window

## Description

Create Parent and Child Windows in memory

## Processing

### Main Window-Creation

- Register a new window Class with name: *titl*, background color:*colr* and Window Procedure:*proc* and display a message in case of error.
- Assign 'style' & 'stylex' inputs of CreateWindowEx(). The WS\_POPUPWINDOW bit in 'style', specifies that *x,y,cx* & *cy* inputs or internally calculated co-ordinates are wrt the physical display screen and not the parent window *prnt*)
- Determine window co-ordinates in pixels depending on *center* with a call to UDF GetWsize().
- Window API CreateWindowEx() creates a Window in memory & returns its handle. The window is invisible since the WS\_VISIBLE is not set in 'style'. It is made visible in a subsequent call to window API ShowWindow() in UDF MsgLoop(). CreateWindowEx() sends a number of non-queued messages including WM\_CREATE to *proc*(the window procedure of the window) and WM\_PARENTNOTIFY to the window procedure of the parent window.

### Main Window-Data for

- Assign the window handle to *hMem*[0]
- *hMem* is passed in the *lpParam* parameter(last) of CreateWindowEx(). *lpParam* is available as the *lpCreateParams* member of a CREATESTRUCT structure, a pointer to which is sent in the *lpParam* parameter of the window procedure(*proc*), under the WM\_CREATE message. This permits access to the data in the memory block in WM\_CREATE message of the window procedure when it is called from CreateWindowEx(), before *hMem* can be inserted in the Property List by window API SetProp().
- Call window API SetProp() to insert *hMem* into the Property List of the window.

### Main Window-Modal/Modeless

- Call EnbWnd() with window handle and the *mdl* input. If *mdl*=1 all other windows are disabled. EnbWnd() sends WM\_CANCELMODE, WM\_KILLFOCUS & WM\_ENABLE non-queued messages to the window proc of the parent window.

### Child Window-Creation

- Create *lpw* child windows with a call to window API `CreateWindowEx()` for each of the *lpw* elements of *hgbll[]*. `CreateWindowEx()` is called with child window details in members of an *hgbll[]* element. A number of non-queued messages including `WM_CREATE` are sent to the default window proc of the child box and a `WM_PARENTNOTIFY` to *proc*, the window procedure of the main window.

### Child Window-Sub Classing

- The window procedure, to handle messages for a child window can be custom or default. Custom window procedures (Sub Class functions) are selected by Instance Sub Classing by window API `SetWindowLong()`. Pointers to the Sub Class and Default window procedures, returned by `SetWindowLong()`, for each child window are stored in *hgbll[]*. The Sub Class function in *hgbll[].proc[0]* and default in *hgbll[].proc[1]*. Sub Classing is performed soon after `CreateWindowEx()` to trap all messages thereafter.

### Child Window-Data for

- As with the parent, *hMem* is inserted into the property list of each child window.

### Child Window-Font

- The font of each child is set with a call to window API `SendMessage()` to send a non-queued `WM_SETFONT` message to the subclassed window procedure for the child.

### **Passing data to Window Procedures and Sub-Classed Window Procedures**

User parameters can't be passed to Window Procedures. Instead A DWORD aligned block of dynamic memory is created with window API `GlobalAlloc()`, data or pointers to data are written to the memory block & a pointer to it returned by `GlobalAlloc()`, is passed to the window procedure as described below. After the window is destroyed, the memory block is deleted by window API `GlobalFree()`.

- a>Data may be placed in the window's Property List. A pointer to the memory block created is placed in the Property List by window API `SetProp()`, after the window is created by `CreateWindowEx()`. In the window procedure the pointer to the memory block is extracted and from the Property List by window API `GetProp()` and pointers to data are extracted from the memory block.
- b>Data may be placed in the extra area of a window. The bytes of extra area memory are specified in the 'cbWndExtra' member of the `WNDCLASS` structure, used to register the window in `MakWin()` & a pointer to the memory block created is placed here by window API `SetWindowLong()`, after `CreateWindowEx()`. In the window procedure the pointer to the memory block is extracted by window API `GetWindowLong()` and pointers to data are extracted from the memory block. Since this method requires Class registration, it can't be used for sub-classed windows. It can be used for super-classed windows and windows for which a new Class is registered.
- c>Data may be placed in the extra area of a Class. using `SetClassLong()` and extracted from there by `GetClassLong()`.

When a window is created by window API `CreateWindowEx()` for a newly registered Class. The window procedure registered for the window is called with a number of nonqueued messages (including `WM_CREATE`) before `CreateWindowEx()` returns, thus before `SetProp()`, `SetWindowLong()` and `SetClassLong()` are called. In these calls to the window procedure, `GetProp()`, `GetWindowLong()` and `GetClassLong()` will return 0 and an attempt to read the contents at this address will result in an access violation error. The inability to access data at this time doesn't pose a problem since the behaviour of these messages is not modified by the window procedure. However the behaviour of the `WM_CREATE` message may be modified. A pointer to the memory block can be accessed under the `WM_CREATE` message as described under '**Processing**' above.

If a window is created by window API `CreateWindowEx()` for a pre-registered Class & is subsequently sub-classed to a user defined window procedure. The default window procedure for the window Class is called with a number of nonqueued messages (including `WM_CREATE`), before `CreateWindowEx()` returns thus before subclassing is performed and before `SetProp()`, `SetWindowLong()` & `SetClassLong()` are called. Since the sub-Class function is called only after subclassing and after the property list is written by `SetProp()`, a call to `GetProp()`, `GetWindowLong()` and `GetClassLong()`, in the subclass function always returns a valid result.

# MsgLoop

**DWORD WINAPI MsgLoop(void \*param);**

## Return Value

When *opn*=1, i/p to the window API PostQuitMessage(**Return Value**), called under the WM\_DESTROY message in the window procedure, is returned. When *opn*=0, the return value is meaningless.

## Parameters

*param* Pointer to an array of parameters

*param*[0] - Handle to Window returned by MakWin()

*param*[1] - Message loop is for window(1) or foreground thread of a multithreaded application

*param*[2] - When *opn*=0 a 1 here indicates that the loop has begun. Unused when *opn*=1.

*param*[3] - When *opn*=0 a 1 here indicates that the loop continues. Unused when *opn*=1.

## Descripton

Message loop for a thread(application) message queue.

## Processing

- Call window API ShowWindow() to make the window visible on the screen by sending a number of non-queued messages to the window procedure.
- Window API UpdateWindow() examines the thread message que for the presence of a WM\_PAINT message. If the message is found, it is removed & sent as a non-queued message to the window procedure otherwise the function does nothing. The purpose is to perform faster screen updates.
- When *opn*=1, the 1st statement in the loop is Window API GetMessage(). The function waits for and extracts a message in the thread's message queue(FIFO). The loop is exited after the WM\_QUIT message is extracted.
- When *opn*=0, the 1st statement in the loop is Window API PeekMessage(). The function obtains a message in the thread's message queue(FIFO) if one is available, if a message is not available, the function doesn't wait. The loop is exited when *param*[2] & *param*[3] are 0.
- Messages may belong to the applications main window, its parent or child windows. The obtained message is sent to the OS which then calls the Windows procedure for the respective window.
- Window API IsDialogMessage() causes the following:-
  - 1>the *Alt* & *Tab* keys to behave as follows:-
    - Dn arrow -Changes Focus(moves cursor) to next child box
    - Alt+<key>-Keyboard accelerator can be used to press a BUTTON
    - Tab -Will cause the keyboard cursor to jump from one box, created with the WS\_TABSTOP style, to another. If the destination box is a GENERAL EDIT box, its contents will be selected, w/ a Blue background. Content selection can be disabled in a Sub-Class window procedure for EDIT boxes. For Dialog boxes this function performs the operations of the TranslateMessage() & DespatchMessage() functions described below. These functions are not called for Dialog Box messages.
- When window API IsDialogMessage() returns FALSE:-
  - 1>Window API TranslateMessage() translates virtual key messages to WM\_CHAR .
  - 2>Window API DespatchMessage() sends translated messages to the OS, which then calls the relevant window procedure with details of the message.

# MakThread

**DWORD \*MakThread(void \*\*dat, TFUNC \*pra, DWORD n, DWORD opn);**

## Return Value

1D Array of background and foreground thread(application) return values

## Parameters

*dat* 1D Array of  $n$  pointers to data for threads. Each pointer is passed as a parameter to the background/foreground thread(application). The pointer can be an address of an array or structure containing data required by the application.

*pra* 1D Array of  $n$  pointers to background and foreground functions of type TFUNC.

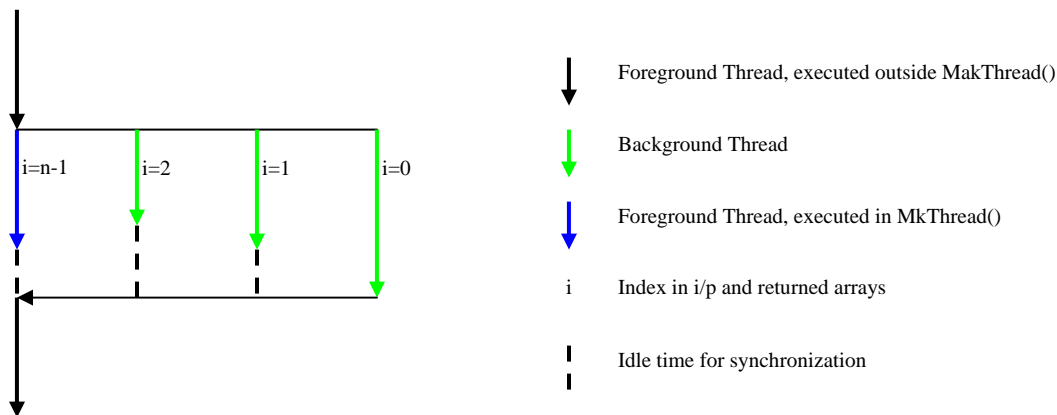
*n* Number of elements in *dat[]* and *pra[]*.

*opn* Return values exist(1)/don't exist(0)

**Note:** The last element in *pra[]*, *dat[]* and the return value, at index  $n-1$ , is for the foreground thread.

## Description

Generates and executes  $n-1$  background applications(threads), executes a foreground application, synchronizes the termination of these applications and returns a dynamic array of background values.



## Processing

Call window API `CreateThread()` to create  $n-1$  background threads. A pointer to data from *dat[]* and function addresses from *pra[]* are extracted. The functions are executed as parallel threads with the data pointers, passed to them as parameters. The foreground thread is executed in parallel and the termination of all threads is synchronized by window API `WaitForMultipleObjects()`. Window API `GetExitCodeThread()`, gets the return values of the background threads and window API `CloseHandle()` closes them. A dynamic 1D array is filled with return values, if *opn*=1 and its address returned.

# Menu functions

## MakMenu

**HWND MakMenu(void \*\*hMem, DWORD cen, DWORD x, DWORD y, DWORD cx, DWORD cy, DWORD colr, char \*hmen, char \*titl, WNDPROC proc, HCURSOR \*ncrs);**

### Return Value

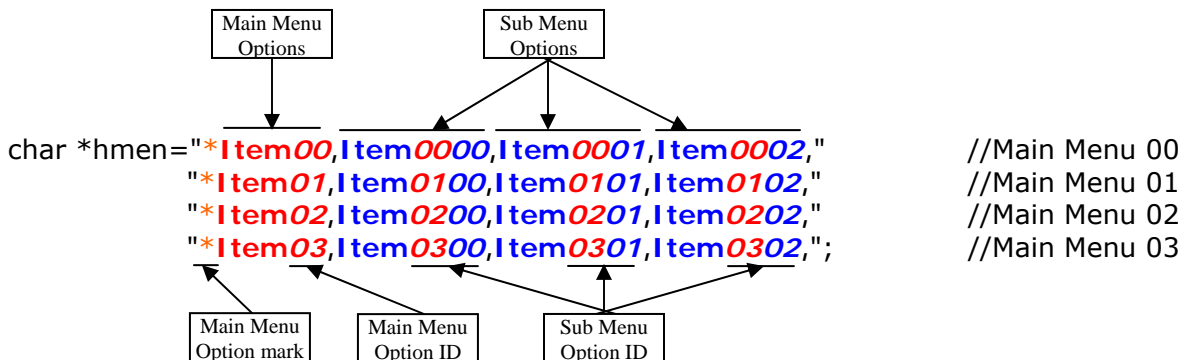
When successful, handle to the Window created otherwise NULL

### Parameters

- hMem* Pointer to memory block w/ data for the window procedure(*proc*). See MakWin() documentation.
- cen* Indicates that the window is to be centered in the display(1) or not(0). When *cen*=1, *x* & *y* inputs are ignored.
- x,y* Top LH corner co-ordinates of the window in pixels. Unused when *cen*=1
- cx,cy* Width & depth of the window *resp* in pixels.
- colr* Window background color passed as an integer returned by RGB(r,g,b). Where r,g & b are intensities of Red, Blue and Green components in the color, in the range 0 to 255
- hmen* A compound string consisting of comma delimited substrings of Main & Sub Menu options.
- titl* Pointer to string to be displayed as a title for the Window.
- proc* Pointer to Window Procedure(Message Handler) for the Window.
- ncrs* Pointer to dynamic Hour-glass cursor created in MakMenu().

### Format of *hMen*

Two adjacent string with only spaces between them are concatenated. This feature of the VC++ compiler can be used to split long strings into more than one line eg "abc" "def" = "abcdef". A Main Menu option string(caption) starts with a '\*'. This is followed by one or more submenu option strings(captions) w/o a '\*'. Leading and trailing spaces and spaces after '\*' are ignored.



### Re-organizing

```

char *hmen = \"*Item00,\" //Main Menu 00
            \"Item0000,\"
            \"Item0001,\"
            \"Item0002,\"
            \"*Item01,\" //Main Menu 01
            \"Item0100,\"
            \"Item0101,\"
            \"Item0102,\"
            \"*Item02,\" //Main Menu 02
            \"Item0200,\"
            \"Item0201,\"
            \"Item0202,\"
            \"*Item03,\" //Main Menu 03
            \"Item0300,\"
            \"Item0301,\"
            \"Item0302,\";
  
```

## Descripton

Create a Menu of the type shown:-

| Title           |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|
| <b>Item00</b>   | <b>Item01</b>   | <b>Item02</b>   | <b>Item03</b>   |
| <b>Item0000</b> | <b>Item0100</b> | <b>Item0200</b> | <b>Item0300</b> |
| <b>Item0001</b> | <b>Item0101</b> | <b>Item0201</b> | <b>Item0301</b> |
| <b>Item0002</b> | <b>Item0102</b> | <b>Item0202</b> | <b>Item0302</b> |
|                 |                 |                 |                 |

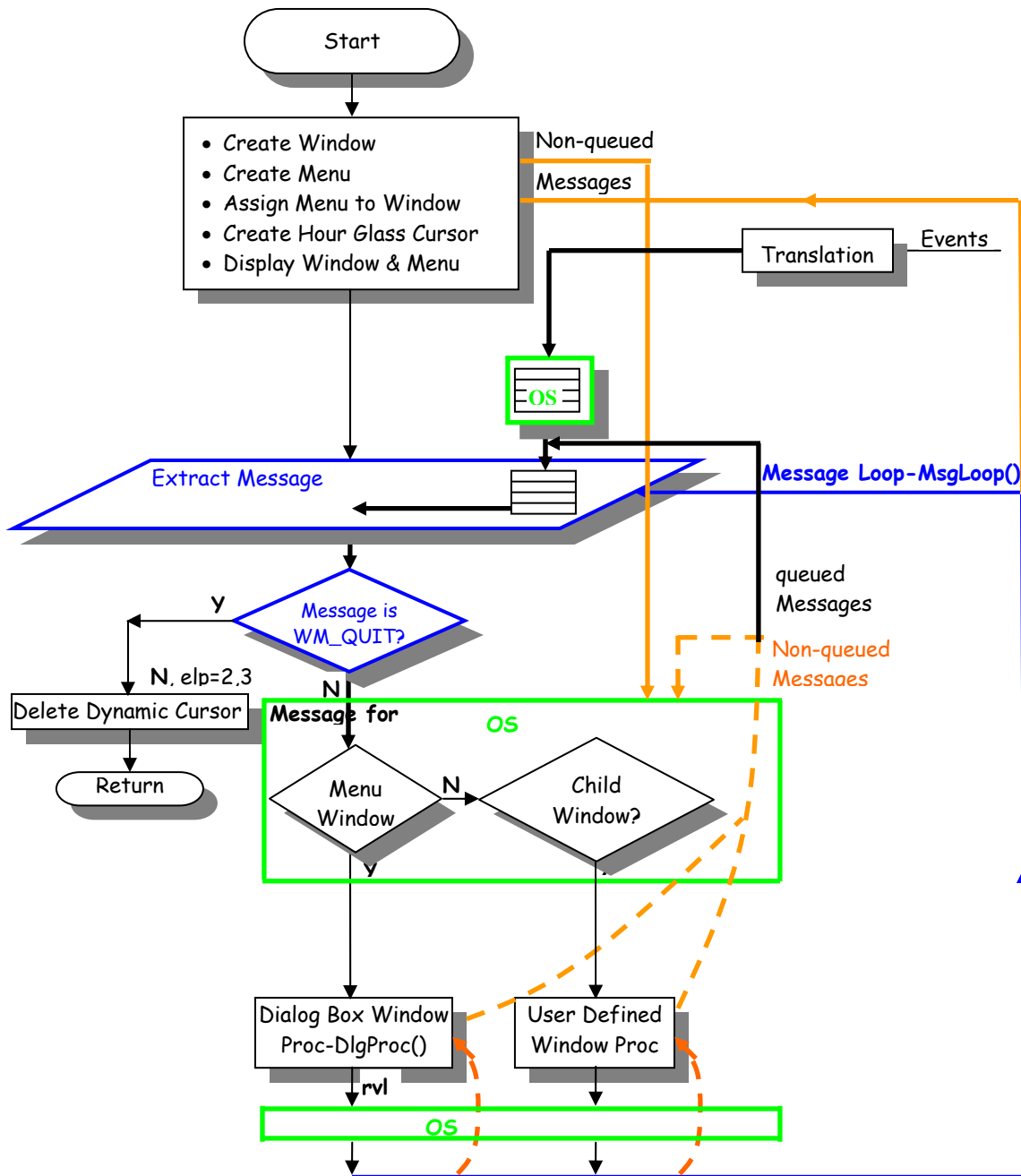
## Processing

### 1>Menu Creation

- A handle('hMenu') for the mainmenu is created with a call to window API CreateMenu().
- A loop is setup to parse the comma delimited substrings from the *hmen* i/p using UDF GetStr().
- For each mainmenu option string encountered a submenu handle('hNam') is created with a call to CreateMenu() and the string is inserted into the mainmenu by window API InsertMenu(). The mainmenu handle('hMenu') and submenu handle('hNam') are passed as inputs with Styles:MF\_POPUP, MF\_STRING & MF\_BYPOSITION. MF\_POPUP, indicates that this is a horizontal menu and the submenu handle passed is to be associated with the mainmenu option. MF\_STRING, indicates that the last i/p is a pointer to the option string and BF\_BYPOSITION indicates that the 2nd i/p is the zero based relative position of the menu option.
- After the mainmenu option string is extracted, submenu option strings are encountered & extracted. The submenu option is inserted into the submenu by window API InsertMenu(). The submenu handle('hNam') and submenu option ID are passed as inputs with Styles: MF\_BYPOSITION & MF\_STRING. The absence of MF\_POPUP, indicates that this is a vertical menu and a submenu option ID is passed instead of the submenu handle. The other styles are as above.
- In menus **Underlined Keyboard Accelerators** are hidden by default. to make it visible:-  
In **Control Panel->Display-> Effects** uncheck **Hide keyboard navigation indicators until I use the Alt key.**

### 2>Other Processing

- A menu window is created by UDF MakWin().
- The mainmenu created is associated with the menu window by window API SetMenu().
- Create Dynamic Hour-Glass cursor with call to window API LoadCursor()
- The menu window along with the menu is displayed a the message loop initiated by UDF MsgLoop().
- Delete Dynamic Hour-Glass cursor with call to window API DestroyCursor()



## ChkMenu

```
void ChkMenu(HWND hWnd, DWORD mndx, DWORD sndx, DWORD set);
```

### Descripton

Place or clear check mark next to submenu item

### Parameters

*hWnd* Window Handle

*mndx* Index of mainmenu option

*sndx* Index of submenu option

*set* Place(1) or Clear(0) '√' mark