A Brief History of the Windows API

The Windows API for Internet Basic programmers is now in its second generation. Originally introduced in 1996, the API consisted of the following files, each of which was "included" in your program:

- CosWin.ibs – The Driver Library. This file contains IB functions and support code necessary to perform various Windows functions. Your program will call subroutines from this library. The names of the subroutines begin with "COS.", such as COS.CreateButton.

- CosWin.inc - Declaratives for constants, variables, and formats that are used by the code contained in CosWin.ibs

- Windows.inc – Definition of constants used to represent all the possible styles, messages, and event notifications for each control available through the API. The constants have been derived from the C/C++ include file "windows.h" which is supplied as part of the C/C++ development platform.

Because Windows programs are "event-driven", the API had to include a method for notifying the IB program when an event occurred. This was accomplished through the use of "call back" subroutines. A call back subroutine is one defined in the IB program that will be executed by the API when a particular event occurs, such as the clicking of a button in the dialog. Since there are a number of types of events that the IB programmer may want to handle, this meant having to define a call back subroutine for each event type. While the API provided the IB programmer with the tools necessary to create true Windows programs for their Comet applications, its implementation was a little cumbersome.

Comet 2004 introduced the second generation of the API. The call back mechanism was replaced with the EventSub mechanism, which had proven itself in the implementation of hyperlinks. This greatly simplified event handling. Now all event types could be handled in one place. The new API also included a new file, CometWin.inc. It provides a new layer between the IB program and the API to take care of many of the housekeeping details previously left to IB program. It is the only one of the API files you need to include in your IB program. You will use subroutines and declarations from the original API files, but CometWin.inc will include those files for you. Subroutines in CometWin.inc are named beginning with "CW.", such as CW.CreateAutoModeDialog**.** Because CometWin.inc includes both declarative and executable code, you will include it twice in your IB program – once at the top of your declarative section and once at the bottom of your executable section.

Dialog or Direct?

The API accommodates two methods for creating the controls for your Windows program. The method you choose is a matter of personal preference.

One method is to use a visual resource editor to design your dialog using "drag-and-drop" tools. We call this the dialog method. When you're done with the design, the dialog is compiled to a .dll, which is loaded by your IB program. No coding is required to define the layout or style of the controls in the dialog. We support use of the resource editor included in Microsoft's Visual Studio.

The other method we call the "direct" method. In this case, the controls are created directly in your IB program by making calls to the API. We'll take a look at examples of each, then compare and contrast the two methods.

A Discussion of WinTemp.ibs

WinTemp.ibs was designed to be a template program to get you started on your Windows program using the direct method. It creates a window with just OK and CANCEL buttons, but has all the hooks for you to add other controls.

The Event Handler

As you know, Windows programs are event-driven. Rather than dictating the flow of the process, the program responds to events initiated by the user. Thus it makes sense that the heart of any Windows program is the code that intercepts these events. In WinTemp.ibs, this is a subroutine called **EventHandler**.

When you create your controls, you will tell Comet which events you want to be notified of. Then you tell it the name of the subroutine to call when an event occurs and you wait for your first event:

```
EventSub EventHandler CW.Event$ CW.Source$
Eventwait
```

CW.Event$ and CW.Source$ are API variables that will receive information about what event has occurred. Then, control will be passed to the subroutine referenced in the EventSub statement:

```
EventHandler:
        gosub CW.ParseEvent                             ! Determine which type of event occurred

        select case cosCtlId
                case IDOK                               ! User pressed the OK button
                        if cosCommand = BN.CLICKED
!                               To do: write code to process OK button
                        endif

                case IDCANCEL                           ! User pressed the CANCEL button
                        if cosCommand = BN.CLICKED
                                CW.ExitEvents = TRUE    ! Signal that we want to stop the event handler
                        endif

!               To do: write code for any other cases we expect to process
        endselect

        if CW.ExitEvents = FALSE                        ! Either re-establish or cancel event handler
                eventsub EventHandler CW.Event$ CW.Source$
        else
                eventsub
        endif
        return
```

The first thing your event handler should do is to determine which event occurred. You do this by calling **CW.ParseEvent**. It loads API variables with information about the event. The two you will use the most are:

cosCtlId The ID of the control that generated the event, ex. IDOK – the OK button
cosCommand The event, ex. BN.CLICKED – the button was clicked

When we return from CW.ParseEvent, we use a SELECT clause to process our event. When we've finished processing the event, we need to either re-establish the event wait mechanism so we can get the next event or

signal to Comet that we are done. In the example above, the event handler is stopped when the IDCANCEL button is clicked. This causes program flow to continue just after the EventWait statement at the top of our program.

So that's all there is to an event handler. You will add a CASE statement for each control from which you expect an event. Inside each CASE you'll have code for each event's cosCommand value.

Initializing the API

OK, so now that you understand how events are handled, there are a few other details you need to attend to. The very first thing your program should do is initialize the Comet Windows API (CosWin). You do this by calling **CW.Open**. This will initialize required variables, hide the cursor, and establish the size of your screen, the default font you will use, and the color of the wallpaper behind your controls:

```
!!!!!!!!!!!                    Initialize CosWin
!       Set CW.Width and CW.Height to customize screen size.  They are ignored if set to 0.
        CW.Width = 80
        CW.Height = 30

!       Set CW.Font$ to change the view font.  It is ignored if set to ""
        CW.Font$ = "Arial"

!       To set your wallpaper color, define values for red, green, and blue.
!       The values assigned here define a sort of parchment color.  For white, set all to 255.
        cosRed = 255 & cosGreen = 254 & cosBlue = 236

!!      As an alternative, if you want the default dialog color, call COS.GetSysColor
!!      gosub COS.GetSysColor                   ! Sets cosRed, cosGreen, and cosBlue

        CW.Color = (cosBlue*65536) + (cosGreen*256) + cosRed
        gosub CW.Open
!!!!!!!!!!!                    Initialization complete
```

Among the API variables that are initialized by CW.Open, one that you may find particularly handy is CW.Remote. If your program is running on a Comet Anywhere client, CW.Remote will be set to TRUE. You will need to know this if you will be loading any bitmaps or launching any Windows files. In that case, you will need to first copy the files from the host to the remote.

If you're curious about the details of CW.Open and CW.ParseEvent, you'll find the code in CometWin.Inc on WDL.

Creating Your Controls

Now the fun part! Once you've got CosWin initialized, you'll create each of your controls. **Each control is really a small window.** It is important to keep this in mind when reading about Windows programming. Your controls will be child windows of your Comet window, which is called the parent window.

The CtlDemo programs include an example of each control supported by the API. For most of them, the process is pretty much the same. For each control you'll assign an ID number, the height, width, and screen position, and any desired style characteristics. You'll use the following API variables:

cosCtlId - the numeric identifier for the control

cosWidth - the width (in Centiunits)
cosHeight - the height (in Centiunits)
cosRow - the screen row (in Centiunits) where the control is to be positioned
cosCol - the screen column (in Centiunits)
cosStyle - the main style characteristics parameter
cosExStyle - the extended style parameter

For some controls, such as buttons and edit controls you'll also assign some text to be displayed on the control. Use **cosCtlText$** for this. And, for most controls you'll need to set **cosDlgFlags** = 0. This is used to tell the API when you're using additional modifiers.

Here's an example that creates a "Next" button:

```
cosCtlId = IDC.NEXT                    ! IDC.NEXT is a symbolic constant set to 41001
cosWidth = 800  &  cosHeight = 150     ! The button will be 8 chars wide and 1.5 chars high
cosRow = 1400  & cosCol = 3600         ! Position button on row 14, column 36
cosStyle = WS.VISIBLE + WS.CHILD + WS.TABSTOP + WS.GROUP + BS.DEFPUSHBUTTON
cosExStyle = 0
cosCtlText$ = "Next"
cosDlgFlags = 0
cosCWnd$ = cosDlghDlg$
gosub COS.CreateButton
```

One thing that might be puzzling in this code is the cosStyle assignment. This defines characteristics of the button. The style values that begin with "WS." are generic "Window Styles" and apply to any control. So, for example any time you create a control and you want it to be visible on your screen, you'll include the WS.VISIBLE style, regardless of the type of control. Then, for specific controls, there may unique styles values. For a button, BS.DEFPUSHBUTTON indicates that this is the default button. A full list of the style values can be found in Windows.inc. Descriptions of the styles available for each control can be found on Microsoft's web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_styles_used_by_mfc.asp

Whenever a window is created it is assigned a handle to uniquely identify it to Windows. Since each control is really a window, every control has its own handle. The parent window also has a handle. The WS.CHILD style indicates that this control will be a child window. We need to pass the parent's handle to Comet when we create the new control. **cosDlghDlg$** will contain the handle to the parent. We pass it in **cosCWnd$.**

After you create the control, you need to tell Comet which event(s) you want to process. You do this by appending the control ID and event to the CosWin API command filter. So, to request that we be notified any time our "Next" button is clicked, we code:

```
cosCWnd$ = cosDlghDlg$                 ! Pass the handle of the parent window, as it owns the command filter
cosCtlText$ = intel(IDC.NEXT) + intel(BN.CLICKED)
gosub COS.AppendCmdFilter
```

cosCtlText$ is a concatentation of word pairs. The first word is the control's ID; the second word is the event ID. As a matter of fact, you can send multiple pairs to the command filter in one call. For example, if you had a CANCEL button as well an a NEXT button, you could either call COS.AppendCmdFilter once for each button, or make just one call:

```
cosCWnd$ = cosDlghDlg$                 ! Pass the handle of the parent window, as it owns the command filter
cosCtlText$ = intel(IDC.NEXT) + intel(BN.CLICKED) + intel(IDCANCEL) + intel(BN.CLICKED)
gosub COS.AppendCmdFilter
```

This method of building the command filter is used for most of the controls. A few controls do it a little differently. The List View control is one. Instead of calling COS.AppendCmdFilter, you will call COS.AppendNotifyFilter2. It

requires 3 parameters for each control event. You can see examples of how this is done in Ctl10.ibs for the List View control.

Shutting down the API

After you stop the event handler and are about to exit your program you need to shut down the API and restore your environment. This is done with a simple call to CW.Close:

```
!!!!!!!!!!                    Program Exit
ProgExit:
        eventsub                        ! Cancel the event handler in case F3 or fatal error occurred
        gosub CW.Close                  ! Restore environment

        if RUNSTATE = 6 exit
        stop
```

CW.Close destroys all of the controls you created, removes your wallpaper, and restores the cursor.

A Discussion of DlgTemp.ibs

DlgTemp.ibs was designed to be a template program to get you started on your Windows program where you've used a resource editor to create a .dll for your dialog. DlgTemp.ibs can be used to create either a modal or a modeless dialog with just OK and CANCEL buttons, but has all the hooks for you to process other controls in your dialog.

The Event Handler

Like WinTemp.ibs, the Event Handler is the control center of the program. When using a dialog, however, there are a couple of important differences. First, in addition to catching events for each control, you will also want to detect when the dialog is being loaded and when it is being closed. Look at the CASE ELSE below:

```
EventHandler:
        gosub CW.ParseEvent                             ! Determine which type of event occurred

        select case cosCtlId
                case IDOK                               ! User pressed the OK button
                        if cosCommand = BN.CLICKED
!                               To do: write code to process OK button
                        endif

!               To do: write code for any other control events we expect to process

                case else                               ! We got an event for something other than our controls
                        gosub CW.IsWindowInitializing
                        if cosState = TRUE              ! Dialog is being loaded
                                gosub InitDialog
                        else
                                gosub CW.CheckAutoModeClosing
!!!!                            If the dialog is closing, CW.ExitEvents will be set to TRUE by the API
                        endif
        endselect

        if CW.ExitEvents = FALSE                        ! Either re-establish or cancel event handler
                eventsub EventHandler CW.Event$ CW.Source$
        else
                eventsub
        endif
        return
```

When the dialog is being loaded, you'll want to initialize the state of your controls such as populating a list box, or setting radio buttons or check boxes. It's also a good time to establish your command filter. In WinTemp.ibs we did these things at the time we created the control.

```
InitDialog:
        cosCWnd$ = cosDlghDlg$
        cosCtlText$ = intel(IDOK) + intel(BN.CLICKED)
        gosub COS.AppendCmdFilter                       ! Setup the command filter

        ! This would be a good place to load any data you need into the dialog and to
        ! initialize the state of each control, ie check boxes, radio buttons,etc.
```

```
cosState = true                          ! Tell the API to size the CosW window to our dialog
gosub CW.AutoSizeDialog

cosCWnd$ = cosDlghDlg$
cosState = SW.NORMAL
gosub COS.ShowWindow                     ! Make the dialog visible
return
```

Another difference when using a dialog is that clicking either the OK button or CANCEL button will cause the dialog to close automatically. All you need to do is to detect that the dialog is closing by calling CW.CheckAutoModeClosing. If the dialog is closing, it will set CW.ExitEvents to true and your event handler will not be re-established. When the dialog is closing, if you want to know which button was clicked, you can save its ID. It is set by CW.ParseEvent into coswParam.

Note: If you want an event other than the clicking of OK or CANCEL to cause the dialog to close, you can call CW.CloseAutoModeDialog when you get that event. Conversely, if you don't want the dialog to close automatically when either or both OK or CANCEL are clicked, you may disable this by handling the event explicitly. As a matter of fact, if you want to retrieve data from the dialog when OK is clicked you will have to handle that button explicitly. Just tell the command filter you want to handle that button, and add your code to the event handler. This gives you total control. The DlgTemp.ibs example handles the OK button explicitly.

Loading Your Dialog

Getting your dialog loaded is really easy. All you need to do is to tell the API the name of your dialog and where to find your .dll. So, at the top of our program, you see:

```
! Establish BEFORE calling CW.CreateAutoModeDialog or else we'll miss the "dialog is initializing" event
eventsub EventHandler CW.Event$ CW.Source$

cosDllNames$ = "DlgTemp.dll"             ! DLL filename
cosDllDir$ = sub(pstat(PARTITION$),41,3) ! Same dir as this program
cosDlgId = IDD.DIALOG1                   ! Dialog ID

! If your dialog was designated as "child" it will be modeless. If "popup" it will be modal.
gosub CW.CreateAutoModeDialog

if cosDlgOpen = FALSE
        print "Dialog execution failed"
        go ProgExit
endif

eventwait
```

Note: If your program is running on a Comet Anywhere client, the call to CW.CreateAutoModeDialog will automatically transfer the .dll from the host to the $(CATOOLS) folder on the remote.

In the Resource Editor

The Resource Editor is where you design your dialog and create its controls. Running DPBuild.exe will quickly generate a template project for you.

When you compile your project to generate the .dll, an ID will be assigned to each control and to the dialog itself. A control's ID is what's assigned to cosCtlID; the dialog's ID is what's assigned to cosDlgId. We recommend using SET statements to give a meaningful name to each ID. As an alternative to entering these IDs manually,

you'll find a program called Inc2Use.exe in the WDL release that will make an include file that declares the IDs for your IB program. Setup Inc2Use to run as part of your "Post-Build Steps" in your project and your IB include file will automatically be updated whenever you make a change to your dialog. Note: This will be done for you if you have created your project through CometDialogProjectBuilder.exe.

Dialog or Direct?

Now that you've seen examples of each, which method is best for you? It really is a matter of personal preference. Here are a few thoughts about each method:

Dialog

- It is really convenient to be able to create your screens by dragging and dropping the controls where you want them.
- It's easy to select a group of controls and have them automatically sized and aligned the same.
- Assigning attributes to a control is simple. A right click on the control pops up a window with check boxes for the style choices, a place to enter your caption, etc.
- Since the code for creating all the controls is encoded in the .dll, your companion IB object will be smaller than a similar program written using the direct method. However, the sum of the two pieces will be larger than the single IB object.
- Visual Studio will take care of assigning identifiers to each of your controls for you.

Direct

- You only need to deal with a single development environment.
- You don't have to package the .dll along with your application. Your IB object contains everything.
- You can intersperse other GDI objects such as hyperlinks with your controls.
- You can always write debug messages directly to the window.

A Beginner's Guide to Events and Functions for Selected Windows Controls

A complete list of events for each control can be found in Windows.inc. They are called "notify" or "notification" codes. A complete list of functions for each control can be found in CosWin.ibs. The Demo programs, WinTemp.ibs, and DlgTemp.ibs are released as part of the "DLG" folder.

Control	Events	Functions	Demo Sample
Button	BN.CLICKED	COS.CreateButton	any CtlDemo program
Check Box	BN.CLICKED	COS.CreateButton COS.CheckDlgButton COS.IsDlgButtonChecked	Ctl3.ibs, Dlg3.ibs
Radio Button	BN.CLICKED	COS.CreateButton COS.CheckRadioBtn COS.GetCheckedRadioBtn	Ctl2.ibs, Dlg2.ibs
Static		COS.CreateStatic	Ctl1.ibs
Edit	cosEN.LOSTFOCUS EN.SETFOCUS	COS.CreateEdit COS.EditLimitText	Ctl5a.ibs, Dlg5a.ibs
List Box	LBN.SELCHANGE LBN.DBLCLK	COS.CreateListBox COS.ListAddString COS.ListDelString COS.ListInsertString COS.ListFindString COS.ListGetCount COS.ListResetContent COS.ListSetCurSel COS.ListGetCurSel COS.ListGetText	Ctl4.ibs, Dlg4.ibs

Combo Box	CBN.SELCHANGE CBN.DBLCLK	COS.CreateComboBox COS.ComboAddString COS.ComboDeleteString COS.ComboInsertString COS.ComboFindString COS.ComboFindStringExact COS.ComboGetCount COS.ComboResetContent COS.ComboSetCurSel COS.ComboGetCurSel COS.ComboGetText COS.ComboLimitText	Ctl14.ibs, Dlg14.ibs
List View	NM.CLICK NM.DBLCLK NM.RCLICK LVN.COLUMNCLICK	COS.CreateListControl COS.ListView.InsertTextColumn COS.ListView.SetColumnWidth COS.ListView.InsertStringItem COS.ListView.SetFullRowSelect COS.ListView.DeleteAllItems COS.ListView.GetNextItem COS.ListView.GetItemText COS.ListView.SortItems	Ctl10.ibs, Dlg10.ibs
All Controls		COS.SetDlgItemFocus COS.EnableDlgItem	

Each function requires setting of parameters before making the call. To determine how to do this for a given function, either look at the sample code from the CtlDemo or DlgDemo program, or search CosWin.ibs for the function name. Comments provided there explain what is required and what will be returned following the call.