V3. Windows

ViewIt eliminates the old distinctions between "alerts" , "dialogs", and "windows". Any ViewIt window can be made to look and behave like a typical alert, dialog, or document window. A more important distinction supported by ViewIt is between modal and modeless windows. Modal windows require the presence of the ViewIt module, whereas modeless windows also require presence of the FaceIt or FaceSt modules.

Modal vs. Modeless

Modeless windows behave in a way that is similar to your program's main menus. The items in such windows, like the items in main menus, are always available to users. This "modeless" behavior is powerful but can be difficult to manage since your program must be able to respond at any time to events in modeless windows, and manage the state of any context-sensitive items in such windows.

Modal windows, on the other hand, are opened and closed as needed, making their management more straightforward. The disadvantage in this case is that the user does not have access to any options in other windows or the main menu bar while the modal window remains open (the program is in one "mode").

The type of window that you choose to use will depend upon the nature of the task involved. Some developers have suggested that all windows should be modeless, but we do not share this view. It often makes sense to force a user to deal with the options presented in a modal window before continuing.

The FaceIt command DoLoop and ViewIt command MdlWnd handle events from modeless and modal windows, respectively. The messages returned to your program by these commands are nearly identical, and are discussed in the "Modal Windows" subtopic below, so be sure to read this even if you intend to only work with modeless windows. A more general discussion of event handling in FaceIt-based programs can be found in the "The Main Loop" topic of the FaceIt guide.

Adding Windows

All ViewIt windows are opened from resource templates of type "FWND". Any number of windows can be opened from the same FWND resource. FWNDs are edited within running programs by entering edit mode (Option-âŒ˜-Shift), or from the window icon menu in the ViewIt Help window. New FWNDs are typically created in one of four ways:

1. Use ResEdit or other resource editor to make a copy of an existing FWND, and renumber and rename it for use with your program. Then call NewWnd to open a window based on it, and enter edit mode to edit the FWND.

OR 2. Run a program that opens the ViewIt Help window (this window) and use its window icon menu (at right) to open a new ViewIt window. ViewIt will add a new FWND for this window to the program's resource file and the FWND can be immediately edited on-line.

OR 3. Run a program that opens another ViewIt window and use the "Edit Another" option from the File menu when in edit mode to open a new ViewIt window. ViewIt will add a new FWND for this window to the program's resource file and the FWND can be immediately edited on-line.

OR 4. Execute a call to NewWnd that refers to an FWND not yet created. ViewIt will then ask you whether you wish to have it create the FWND before continuing. This new FWND will be added to the program's resource file and can be immediately edited on-line.

REMINDER: As discussed in the "Startup" notes, remember to use resource ID numbers < 1100 or > 7499 for program resources since the range of ID numbers from 1100-7499 is reserved for FaceWare modules.

COLOR NOTE: If you plan on displaying or drawing more than the original 8 QuickDraw colors in the window, make certain that the "Prefer Color Window" option is checked in the Window dialog so that ViewIt will create a color window when running on machines that support Color QuickDraw.

MODAL WINDOWS

Modal window management by your program occurs within isolated sections of your code in which the modal window is opened, responded to, and closed. These steps correspond to three ViewIt commands (see "Window Commands" for further info): NewWnd opens a new window, MdlWnd responds to events in the window, and EndWnd closes the window.

Most event processing in modal windows is automatically handled by ViewIt, meaning that very little code needs to be written to open simple modal windows. The following code fragments illustrate this point, and can be copied and pasted into an existing program to open a new modal window:

```
FaceIt(nil,NewWnd,1000,0,0,0);      Pascal
FaceIt(nil,MdlWnd,1000,0,0,0);
FaceIt(nil,EndWnd,1000,0,0,0);

FaceIt(0,NewWnd,1000,0,0,0);         /* C */
FaceIt(0,MdlWnd,1000,0,0,0);
FaceIt(0,EndWnd,1000,0,0,0);

FaceIt(0,NewWnd,1000);               /* C++ */
FaceIt(0,MdlWnd,1000);
FaceIt(0,EndWnd,1000);

call FaceIt(0,NewWnd,1000,0,0,0)   !FORTRAN
call FaceIt(0,MdlWnd,1000,0,0,0)
call FaceIt(0,EndWnd,1000,0,0,0)
```

The above code will cause ViewIt to attempt to create a new modal window based on FWND 1000. If FWND 1000 is not present, then ViewIt displays an alert which allows you to add a new FWND to your resource file (as noted above). A default window is then opened and can be immediately edited by entering ViewIt's editing mode.

TIP: If the contents of a window need to be adjusted by the program just after opening it, then keep the window hidden when calling NewWnd to minimize any unsightly screen drawing. This is done by passing -FWND ID:

```
FaceIt(nil,NewWnd,-1000,0,0,0);
```

which keeps the window hidden until MdlWnd is called.

• Item Events

Most modal windows contain at least one control which performs some program-specific operation when pressed. In order to give the main program a chance to respond to clicks in such controls, any ViewIt control can be marked as "enabled" by checking the "Returns On Hit" option in the Control dialog. Clicks in such controls then cause ViewIt to exit from the MdlWnd call and return control to the main program with a menu or "pseudo-menu" event (a message that uses uMenuID, uMenuItem, and other variables).

To handle events from enabled items, a simple event loop can be placed around the call to MdlWnd:

```
FaceIt(nil,NewWnd,1000,0,0,0);
repeat
 FaceIt(nil,MdlWnd,1000,0,0,0);
 if (wcHit = 1) then
  [do first thing]
 else if (wcHit = 2) then
  [do second thing]
until [done];
FaceIt(nil,EndWnd,1000,0,0,0);
```

where in many cases the enabled items will be button-type controls. When returning from MdlWnd, the following fRec variables indicate which control was hit:

uMenuID = FWND ID of parent window
uMenuItem = wiHit = item ID
uResult = undefined
uString = undefined
wiHit = item ID (a programmer-defined number)
wvHit = view number (that contains hit control)
wcHit = control number (within parent view)
wClick = click type (1 = single, 2 = double, 3 = triple)
wEvent = raw event

The variables wvHit and wcHit identify the item hit on the basis of its position in the window. If a window contains 2 views that each contain 5 controls, for example, and the 4th control in the 2nd view is hit, then ViewIt returns wvHit = 2 and wcHit = 4.

A disadvantage of using wvHit and wcHit to identify items hit is that the order of such controls in a window cannot be changed without affecting the program. Alternatively, an "item ID" can be assigned to controls that is returned in wiHit and uMenuItem. This item ID is set in the Title or Control dialogs, and can be any number between -32767 and +32767. Use of an item ID allows you to edit windows without worrying about maintaining the order of controls in the window since a control's item ID is not changed when controls are reordered or moved from one view to another.

Finally, if you need to distinguish between single, double, and triple clicks, note that the a single click message (with wClick = 1) will always preceed a double click message (wClick = 2), and a double click message will preceed a triple click message (wClick = 3). Thus triple clicking an enabled control will produce three messages, but your program can decide which of these to respond to by checking the wClick variable.

WARNING: When responding to events, do not assume that the content of any "u", "w", or "c" prefixed fRec variables will be preserved across later calls to the Control Manager or the FaceIt dispatching procedure. Values that need to be used in multiple Control Manager or FaceIt calls should be preserved in local variables. One exception: Most utility commands preserve the "w" & "c" variables.

• Editable Items

Enabled editable-type controls work a little differently than other types of enabled controls. (As noted above, an "enabled" control is one that has its "Return On Hit" option checked in the Control dialog.) In this case a message is posted whenever the enabled editable control is selected or deselected (such as when tabbing between items). The message posted simply informs the main program that the selected state of an enabled editable control has changed:

uMenuID = 1200
uMenuItem = +2 (selected) or -2 (deselected)
uResult = control handle of selected/deselected control
& other event-related variables are undefined

To learn more about the affected control, pass the control handle returned in uResult to GetCtl:

```
FaceIt(nil,GetCtl,0,0,0,uResult);
```

which, for example, would return the item ID, view, and control numbers in ciIndex, cvIndex, and ccIndex, respectively. Information about the currently selected control can also be found in vSelectCtl, vSelectRec, and vSelectID.

In the case, for example, of tabbing between two controls that are both enabled, ViewIt would post two messages: one for the deselection of the first control (uMenuItem = -2) followed by one for selection of the second control (uMenuItem = +2). These messages are returned from MdlWnd in this order, and your program can choose to deal with one, both, or neither.

Posting a special message in response to changes in the selected control complements

FaceIt's optional support for notifying the program when the active window is changed via a similar message (uMenuID = 1100, uMenuItem = 2). The active window and the currently selected control together define the current program "context". A typical use of these messages is to change the state of menu items when the active window is changed, or to change the options presented in a ViewIt window as the user jumps (by click, TAB, or Command Key) from one editable item to another.

NOTE: The "1200, -2" message is not posted for controls that are deselected before being disposed of (i.e., when their parent window is being closed or when DspCtl is called), or when their parent window is being deactivated.

• Menu Events

The selection of program menu items from menu controls (controls associated with MENU resources) generate menu events that can be distinguished from hits in control items by the fact that uMenuID returns with the menuID of the selected menu (rather than the FWND ID). In this case,

uMenuID = menuID of selected menu
uMenuItem = selected menu item number
uResult = selected menu item label (or zero)
uString = selected menu item text
& other event-related variables are undefined

If the menu control is also enabled (i.e., it has its "Return On Hit" option checked in Control dialog), then the program will see both an item hit event and a menu event (in that order). See the FaceIt Guide for a further discussion of "program", "labeled", and "standard" menu items.

Note that menu events are not generated by standard menu items. This means, for example, that the code which runs this help window does not need to deal with any menu events since all of the menu items in the controls at the top of this window are standard items.

• Special Keys

The Return key's default behavior is to hit the default button in the ViewIt window (set by checking "Is Default Item" in the Control dialog). If, however, the "Return" item in the "Keys Accepted" menu is checked for an editable control, and that control is the selected control, then the Return key will be passed to the control's control driver for further processing.

The Enter key's default behavior is similar to the Return key and can be similarly modified by checking the "Enter" item in the "Keys Accepted" menu for an editable control. The default behavior of the Enter key can also be modifed by checking the "Use Enter as Tab Key" option in the Window dialog, which causes the Enter key to behave as if Tab was pressed.

The Tab key's default behavior is to tab from one editable control to the next in a ViewIt window (Shift-Tab reverses direction). If, however, the "Tab" item in the "Keys Accepted" menu is checked for an editable control, and that control is the selected control, then the Tab key is passed to the control's control driver for further processing.

Command key combinations can be associated with controls in ViewIt windows (set in Title dialog). Pressing such a command key combination results in hitting or selecting the control. If the "Command" item in the "Keys Accepted" menu is checked for a control, then command key combinations that do not appear in menus and are not associated with controls are passed on to the driver for further handling.

The Help key's default behavior is to be equivalent to pressing "âŒ˜/". To use a different command key for help, change the first character of STR 1201 from "/" to some other character, or enter a space character to have the help key passed to control drivers. (To make this change program-specific, modify a copy of STR 1201 in your program file.)

The Escape (= Clear) key's default behavior is to pass the key on to the control driver since many controls have built-in support for Clear. If you would prefer to associate Escape

with a command key combination (such as "âŒ˜."), then change the second character in STR 1201 from a space to the desired command key. (To make this change program-specific, modify a copy of STR 1201 in your program file.)

All other key down events get passed as messages to the driver that is supporting the currently selected editable control. If necessary, an override procedure can be used to intercept and modify these key events before they reach the driver (as described in the "Override" topic; "vDemoXY" also contains an example of this).

• Module Messages

Any FaceWare module can post a message back to the main program at any time. These messages are characterized by uMenuID = baseID of the module, with other event-related variables defining the type of message. The editable item event posted by ViewIt (uMenuID = baseID = 1200, and uMenuItem = 2) is an example of such a message.

• Program Messages

Programs can use the utility command PstEvt to post messages back to their own event loops (i.e., the part of their code that responds to messages from DoLoop or MdlWnd). See the description of PstEvt in the "Other Utilities" topic for a discussion of such messages, plus the "faking" of user actions by posting your own events.

• Other Modal Events

ViewIt's Window dialog contains the option "Return Modal Events". If this option is checked, then ViewIt also returns all unprocessed events to the main program. Such events include disk events, Apple events, and update events from non-ViewIt windows. Control returns from MdlWnd with,

uMenuID = 0
wEvent = raw event
& other event-related variables are undefined

where the program is again able to distinguish this event by examining uMenuID.

In the case of disk events, ViewIt will mount uninitialized disks inserted in a floppy disk drive before returning the disk event, so there is no need to have disk events returned for this purpose.

Returning update events from non-ViewIt windows is useful in environments like HyperCard where draggable modal ViewIt windows can be opened above stack windows that require updating if a modal ViewIt window is dragged over them.

• Event Summary

A modal event loop (a loop around MdlWnd) that responds to both item hits, menu selections, and other events would need to use uMenuID to distinguish these event types:

0 -> unprocessed raw event
menuID -> menu event
FWND ID -> item hit
baseID -> module message
other -> program message

Items hit can then be identified by item ID, or combination of view and control numbers.

In practice, most modal event loops are simpler than this since the programmer knows what type of events will be returned. If, for example, "Return Modal Events" is not checked, and the window does not contain any menu controls with program menu items, and there is only one control in the window that returns when hit, and the window is to be closed when that control is hit, then the event loop would be reduced to the original three lines (NewWnd, MdlWnd, and EndWnd).

• Loop Options

Parameter b of the MdlWnd command can be used to force ViewIt to return control to your

program. If b = -1, then control is returned without processing any pending events. This might be useful in a situation where an initially hidden modal window needs to be made visible for a time before entering a modal event loop:

```
FaceIt(nil,NewWnd,-1000,0,0,0); modal hidden
[finish hidden window setup]
FaceIt(nil,MdlWnd,1000,-1,0,0); modal shown
[continue visible window setup]
repeat
 FaceIt(nil,MdlWnd,1000,0,0,0); modal in use
 ...
```

If b = -2, then control is returned from MdlWnd after processing any pending events. This can be useful in cases where your program must monitor something or perform some periodic action while a modal window is open. In this case, if no event was the cause of control being returned, then uMenuID and wEvent.what will be zero:

```
repeat
 FaceIt(nil,MdlWnd,1000,-2,0,0);
 if (uMenuID = 0) then
  [do your own thing]
 else
  [respond to events]
until [done];
```

MODAL ALERTS

Modal alerts are a special type of modal window that require just one call to NewWnd (with b = 3) to open, manage, and close the window. For example, the following line,

```
FaceIt(nil,MdlWnd,1000,3,0,0);
```

will cause ViewIt to open the window based on FWND 1000 as a modal window, and then keep the window open until an enabled item (typically a button) is hit (i.e., ViewIt does the work of calling MdlWnd and EndWnd). After closing the window, the number of the control hit is returned in uResult (or 0 if the window could not be opened).

Modal alerts provide a simple way to open simple alert-type windows and can be used in place of standard Macintosh alerts (such as those opened with the ShoAlt command), but severely limit the amount of control you have over the window and the amount of data that can be transferred to and from the window.

MODELESS WINDOWS

Modeless windows require use of the FaceIt or FaceSt event handling modules (described in the FaceIt Guide). The simplest approach is to use FaceIt to get it to do most event handling by calling DoLoop at the top of your program's main event loop. The following discussion assumes that you are using FaceIt, although the menu or pseudo-menu events returned from DoLoop are also seen when using FaceSt.

Modeless windows can be opened or closed at any point in your main program code. As with modals, NewWnd is used to open a modeless window (with b = 1) and EndWnd to close it, but, unlike modals, these commands need not be used together in isolated sections of your program code.

• Modeless Events

Modeless window events are handled within a program's main event loop. One way to think about this is to consider FaceIt's DoLoop command as a replacement for MdlWnd. This is illustrated by the following code fragment:

```
FaceIt(nil,NewWnd,1000,1,0,0); open window
```

```
...
repeat
 FaceIt(nil,DoLoop,0,0,0,0);     get events
 if (uMenuID = 1000) then          FWND 1000
  if (wvHit = 2) then              view #2
   if (wcHit = 4) then             control #4
    ...
 else if (uMenuID = 101) then     menuID 101
  if (uMenuItem = 2) then         item #2
   ...
until [done];
...
FaceIt(nil,EndWnd,1000,0,0,0); close window
```

where "done" refers to some condition that causes the loop to be exited, and the final EndWnd command was included to illustrate the relationship between this example and the modal window code presented above. In practice, when and whether modeless windows get opened or closed is under your program's control.

As with modal windows, uMenuID indicates the type of event being returned by FaceIt, and the same variables are used to indicate which control or menu item was chosen. Unlike modal events, however, a modeless event could be from any modeless window, or any menu control, or any main program menu. This means that the program's main event loop around DoLoop is likely to have more cases to consider than typical modal event loops, and that you'll be paying more attention to uMenuID.

As with modal windows, unprocessed events are returned with uMenuID = 0, but the raw event is found in fEvent rather than wEvent, and you cannot stop such unprocessed events from being returned (the "Modal Events" flag set in the Window dialog is ignored). All other event types from modeless windows return the raw event in wEvent.

Enabled editable items in modeless ViewIt windows behave the same as in modal windows, with the same message being posted (uMenuID = 1200, uMenuItem = ±2, uResult = control handle) when the identity of the selected control is changed.

• Modeless <-> Modal

ViewIt supports the temporary conversion of modeless windows to modal windows. A modeless window can be made modal by calling MdlWnd with a = 0 (see "Window Commands" topic for a complete description of this command) which causes the window to be brought to the front above all other windows. This call is usually part of a modal event loop in an isolated section of program code:

```
repeat
 FaceIt(nil,MdlWnd,1000,0,0,0);
 ...
until [done];
```

which keeps control of the window until some condition or event occurs. The window can then be made modeless again by calling MdlWnd with a = 1:

```
FaceIt(nil,MdlWnd,1000,1,0,0);
```

which puts the window back in its original position in the window list and marks it as being modeless.

FLOATING WINDOWS (require FaceIt)

A "floating" window is a special type of modeless window that floats above the active window. Any number of floating windows can be open at the same time, and all will remain above the active window. All floating windows are "active" in the sense that clicks in their content area are immediately processed and each such floating window can contain any number of editable controls.

Floating window management is supported by code within the FaceIt module, and thus can only be used when FaceIt is in use (not FaceSt). Any FWND can be used to open a floating

window by passing its ID to NewWnd with b = 2 to designate that the window should be installed as a floating window. Although any window type can be a floating window, in most cases you'll want to use our custom palette WDEF for floating windows so that they can be easily distinguished from other modeless windows (see last 2 choices in our Window dialog when in edit mode).

All commands that apply to modeless windows can also be used with floating modeless windows. Floating windows can also be made modal temporarily using the same MdlWnd call described above for modeless windows ("Modeless <-> Modal" discussion). To convert from modal back to floating, however, pass b = 2 when calling MdlWnd.

The close box in a floating window is never associated with the standard "Close" menu item, and either hides the window or returns a close message when pressed (see discussion below).

MENU WINDOWS (require FaceIt or FaceSt)

Any modeless or floating modeless window can be displayed as a menu. The menu can be 1) any of the main or hierarchical menus auto-initialized by FaceIt (but not FaceSt), 2) menus in pop-up menu controls, or 3) menus popped up with PopMen.

When displayed, the contents of the menu will look exactly the same as that of its associated window. As the mouse is dragged across the menu, any button, check box, or radio button-type controls will be hilited. If the mouse is released while hiliting a control, then that control receives a "hit" and ViewIt responds in the same way as a direct hit in the parent window (i.e., check boxes are checked/unchecked, enabled items return a message to the program, etc.). No menu-type event is generated.

Menu windows are created via MENU resources that have the following characteristics:

- normal resource ID and menu ID numbering
- just one menu item equal to the FWND ID of parent window
- MDEF ID = 1201 (our custom MDEF)

The parent window must be opened by the program with a call to NewWnd some time before the menu is used, but need not be visible. If the window does not exist, then no menu is displayed.

Menu windows can be "torn off" to become true windows by dragging outside the content area of the menu (we simply show and move the parent window to the desired location). The "torn off" window will be modeless or floating depending on how the parent window was opened.

Menu windows cannot be torn off if a modal window is open or FaceSt is in use. You can also prevent menu windows from being torn off by placing a zero in front of the FWND ID in the MENU resource (i.e., "01010" vs. "1010"), or, if the menu is a non-main menu, by using "-" as the first character in the menu's title instead of "+".

A common use of menu windows is to support "palette" type menus which contain multiple picture- or icon-based controls that are of type button, check box, or radio button. They are also useful for giving the user quick access to setup dialogs without needing to make such dialogs visible.

The "fDemoXY" program contains an example of a floating window that can be displayed as either a hierarchical or pop-up menu, and then torn off to show the floating window. The "Quick Help" menu at the bottom of this help window also illustrates use of a menu window that can be torn off (if the help window is modeless and FaceIt is in use).

Switching Content

The NewWnd command not only supports opening completely new windows, but also the replacement of an existing window's contents with controls from another FWND. The following, for example, replaces the window whose contents are associated with FWND 1000 with controls from FWND 1010 (see NewWnd in the "Window Commands" topic for further info):

```
FaceIt(nil,NewWnd,1010,1000,0,0);      Pascal
FaceIt(0,NewWnd,1010,1000,0,0);        /* C */
```

```
FaceIt(0,NewWnd,1010,1000);              /* C++ */
call FaceIt(0,NewWnd,1010,1000,0,0)   !FORTRAN
```

Once switched, this "new" window behaves as if it had been opened by FWND 1010 (i.e., all on-line editing and messages will correspond to FWND 1010, not FWND 1000).

One drawback to switching window contents with NewWnd is that it is more time consuming than simply showing/hiding or dynamically adding views in the same window (as discussed in "Paged Views" under the "Views" topic). On the other hand, it takes more time to initially open a window with many views, and may be slightly more difficult to edit such a window, so which approach you use will depend on the number and type of views in a window.

Closing Windows

ViewIt modal or non-floating modeless windows that have a close box also support the "Close" standard menu item. A hit in this close box, or selection of the "Close" item, causes ViewIt to return a close message to the program: uMenuID = FWND ID and uMenuItem = wiHit = wcHit = wvHit = -1.

For modeless windows, you can alternatively have ViewIt hide the window instead of posting the close message by setting the "Close = Hide" option in ViewIt's Window dialog.

For floating windows, a hit in their close box will return the close message for the floating window (unless "Close = Hide" is checked), but selection of the "Close" menu item will return a close message for the active non-floating window (if any), not for the floating window.

A typical response to the standard close message will be for the program to close the window by calling EndWnd. Before doing this, however, programs often enquire whether the user wishes to save the contents of the window. If working with controls that support the "Save" message (such as a TextCt text-editing control), these controls can be notified of the pending window closing by calling SavCtl, thereby giving users a chance to save their contents:

```
...
if (uMenuID = 1000) then      FWND 1000 event
 if (uMenuItem = -1) then     close box/item
   begin
    FaceIt(nil,SavCtl,1000,0,0,0);
    if (uResult = 0) then
     FaceIt(nil,EndWnd,1000,0,0,0);
   end;
...
```

where the window closing should be aborted if uResult returns a value from SavCtl other than zero, and passing c = d = 0 to SavCtl notifies all controls in the window (see SavCtl in the "Control Commands" topic for a complete description of this command).