

## V5. Controls

Controls within ViewIt windows are much more powerful than the standard controls seen in Mac dialogs. This power is based upon an extension of the Control Manager that (1) stores more information about the characteristics of each control, and (2) sends a wider variety of messages to each control. This extension is what makes it possible to support multiple styles and colors, as well as complex controls such as this help control that contains its own scroll bar and can respond to a full range of events.

This topic presents information about ViewIt controls that applies to all control types: how to get info about existing controls, which toolbox calls work with ViewIt controls, and ViewIt's support for control scrolling, growing, styles, colors, etc. Information about specific control types can be obtained from the corresponding driver's on-line help. Driver help can be opened from within ViewIt's Control dialog:

1. Add the control of interest to a ViewIt window
2. From within edit mode, select the control (click once on it)
3. Open the Control dialog (shortcut: triple click)
4. Press the "Driver Help" button

or, if the ViewIt Help window is open, from its "Drivers" menu.

The menu controls at the top of this window, for example, are supported by the BaseCt basic control driver which has an on-line help window that describes all of the control types that it supports: text, icons, lists, menus, dials, etc.

## Getting Info

The extra information associated with each ViewIt control is stored in a relocatable block whose handle can be found in the "ctrlDefProc" field of the control's standard control record (which is itself another relocatable block). Since it would be a headache for programmers to retrieve this info from relocatable blocks, ViewIt's GetCtl command can be used to get the info from both the standard control record and the extra block (see GetCtl in "Control Commands" topic).

GetCtl copies the content of the two blocks into the fRec variables cNext to cString, where cNext to cTitle corresponds to the standard control record, and cStuff to cString is a copy of the supplemental record. GetCtl also returns the control's control handle in cControl, info about its position in the control list in ciIndex, cvIndex, and ccIndex, and the driver's baseID in cBaseID (see the "fRec Record" topic for brief description of each variable).

Common uses of GetCtl include:

- getting the control handle for use in other calls
- determining the current value or state of a control
- getting the current control rect prior to drawing
- getting the handle of a resource linked to the control

**WARNING:** Do not assume that the content of fRec variables returned by GetCtl is preserved across calls to the Control Manager or the Facelt dispatching procedure. Values that might need to be reused, such as a cControl control handle, should be saved in program variables. One exception: Most utility-type commands preserve the "w" and "c" variables.

## Toolbox Calls

Although we have greatly enhanced the capabilities of controls in ViewIt windows, most of the Control Manager toolbox calls can continue to be used with ViewIt controls. Moreover, toolbox calls applied to view controls in ViewIt windows will automatically affect all daughter controls in the view (i.e., hiding a view will also hide controls within the view).

The toolbox calls supported are:

HiliteControl	SetCtlValue
ShowControl	SetCtlMin
HideControl	SetCtlMax
Draw1Control	GetCtlValue
SizeControl	GetCtlMin
MoveControl	GetCtlMax
DragControl	SetCRefCon
GetCtlAction	GetCRefCon
GetCTitle	SetCTitle

where ViewIt's GetCtl makes the "Get..." calls unnecessary, DrwCtl can be used in place of "Draw1Control", ShoCtl in place of "ShowControl" and "HideControl", ActCtl in place of "HiliteControl", SizCtl in place of "SizeControl", and MovCtl in place of "MoveControl" and "DragControl".

CAVEAT: The Control Manager does not always send the proper messages to controls that are hidden. This is due to the fact that it assumes that controls redraw themselves in a simple way based upon the current state of their control record. Many ViewIt controls, however, maintain private data that must be updated whenever the control is moved, resized, or has its value or hilite state changed. With respect to moving and resizing, ViewIt makes an effort to compensate for the Control Manager by fixing a hidden control's private size-related data whenever it is reshowed. When using the "HiliteControl" or "SetCtl..." calls, however, you may find that these do not work properly with complex, hidden ViewIt controls. A good rule to follow is to use the ViewIt commands in place of the toolbox calls to eliminate problems with hidden controls.

The toolbox calls not supported are:

DrawControls	UpdtControl
NewControl	GetNewControl
DisposeControl	KillControls
FindControl	TestControl
TrackControl	SetCtlAction

where ViewIt's AddCtl or AddVew should be used in place of "NewControl" and "GetNewControl", DspCtl in place of "DisposeControl" and "KillControls", and the others are replaced by other ViewIt features and commands. NOTE: These restrictions and substitutions only apply to ViewIt controls. Private controls maintained as part of a ViewIt control (such as the scroll bar in this help control), are treated as standard Mac controls by the control driver.

## Rectangles

The settings displayed in ViewIt's Bounds dialog are the ones saved in FWND, FVEW, or FCTL resources. When controls are initialized from such resources, the "Pen", "Bounds", "Indent", and "Content" information from the resource is converted to the cRect, cClip, cContent, and cLimit rectangles used by ViewIt and control drivers.

cRect is the standard control bounds. cClip is the visible content area of the control (= cRect - frame and indent). cLimit defines the minimum & maximum bounds of cRect when resizing the control. cContent is either the same as cClip, growing and shrinking with the control bounds, or is a fixed size that can be larger or smaller than cClip.

Many control drivers ignore cContent and always draw their content to fit cClip. In this case it is best to set up cContent so that it tracks cClip. This is done by setting the "Max H" and "Max V" content values in the Bounds dialog to zero. This help control, for example, makes no use of cContent, and manages its own scrolling.

Other control drivers and types do make use of cContent. The SICN-based static controls at the top of this window, for example, have non-zero "Max H" and "Max V" content values to inform the driver that their contents should not be stretched to fit cClip.

## Scrolling

Another use of cContent is to support "hand scrolling" of a control's content. This support is built into ViewIt, and is activated by setting a non-zero "Max H" or "Max V" and checking the hand icon in the Bounds dialog. When above the control, the cursor is then changed to a hand which can be used to "hand drag" the control's contents: dragging the content down moves cContent lower relative to cClip, up moves cContent higher, etc. This feature will only work with controls that draw their contents into cContent. The most common use is to support hand dragging of the content of views, although views can also support scroll bar-based scrolling independently.

Controls with content areas larger than cClip can also be scrolled directly by programs via the ScrCtl command. A program can also use ScrCtl to resize the content area of a control. This capability is most often used in programs that dynamically construct views that have a varying number of daughter controls: AddVew adds the view, ScrCtl adjusts its content size, and AddCtl adds daughter controls.

## Growing

Controls can be "attached" to the right or bottom sides of their parent views, and views to the right or bottom sides of their parent window, by setting the "Attach Right" or "Attach Bottom" options in the Bounds dialog. If the window is zoomable or growable (options set in Window dialog), then the attached controls and views that are not of a fixed size ("Min H"  $\neq$  "Max H" or "Min V"  $\neq$  "Max V") are used to determine the minimum and maximum window size (via the control or view's cLimit rectangle described above).

This first view in this help window, for example, is attached to the bottom and right sides of the window, and the help control within it is attached to the bottom and right sides of the view. If the window size is changed, then the attached view and control are resized to fit the new window size (try it). The help control is aligned with the right edge of the view's visible content area, which explains why there is space between the bottom and right sides of the help control and the window since the parent view has a non-zero right and bottom indent.

If the window has a grow box, then ViewIt displays it in the first control of the first view that is attached to both the right and bottom sides of the window (or at the bottom, right of the window if no such control exists). In the case of this window, the help control meets this criteria, which explains why the grow box is drawn within it.

Although ViewIt makes growing and zooming quite easy to implement, there are a few guidelines you should follow to achieve the best results:

- If the window is growable or zoomable, then at least one view in the window should be attached to both the bottom and right sides of the window.
- If more than one view is attached to the both the bottom and right sides of the window (such as when supporting the "paging" of views), then each of these overlapping views should have the same frame and indent size. This ensures that the grow box is drawn properly as views are switched.
- Do not attach controls to views that are set up to be hand scrolled or scrolled via scroll bars (think about it). Such views can themselves, however, be attached to windows.
- ViewIt does not protect you from adding attached controls or views whose limits cannot be reconciled (i.e., when one control's maximum size is smaller than another control's minimum size). Strange zoom or grow behavior will result in such cases.
- ViewIt updates window and control size limits according to the attached controls and views in just two cases: (1) when windows are created, and (2) when leaving editing mode. If your program adds, removes, moves, or resizes (including resizing the content area with ScrCtl) attached controls or views using ViewIt or toolbox commands, then call SizWnd with  $b = c = 0$  to force ViewIt to update all attachments. This requirement does not apply to hiding and showing.

## Floating

Controls that are attached to the right or bottom sides of their parent view or window, but are of a fixed size ("Min H" = "Max H" or "Min V" = "Max V"), will appear to "float" with the right or bottom edge as the parent view or window is resized (i.e., they remain attached by moving rather than stretching). Two of the guidelines given above also apply to floating controls:

- Do not attach floating controls to views that are set up to be hand scrolled or scrolled via scroll bars.
- ViewIt updates the position of floating controls or views at the same time that it updates the size of attached controls or views that grow with their parent view or window.

This help window contains several floating views in addition to the view containing the help control and grow box:

- a small view containing icon menus at the top, right which floats with the right edge of the window
- a small view containing the "Interrupt" button at the bottom, right which floats with both the right and bottom edges
- the view containing commands and messages at the bottom of the window which floats with the bottom edge, but stretches with the right edge of the window

## Dragging

In some cases you may be interested in having the user drag a control within its view. The toolbox call "DragControl" could be used to do this, but DragControl knows nothing about the views within a ViewIt window. The ViewIt command MovCtl (described in "Control Commands") does a much better job of supporting dragging since it properly restricts dragging to within the parent view and will auto-scroll

scrollable views.

The following code fragment, for example, would support dragging the enabled control at v2c4 in the ViewIt window based on FWND 1005:

```
if (uMenuID = 1005) then
  if (wvHit = 2) and (wcHit = 4) then
    begin
      Facelt(nil,GetCtl,1005,0,2,4);
      Facelt(nil,MovCtl,ord(cControl),3,0,0);
    end;
```

Finally, note that although MovCtl can be used to drag any control type, the mouse button must be down at the time that MovCtl is called, which will usually only be the case when clicking on an enabled static control. To support user dragging of other control types, it may be necessary to override controls and intercept the mouse down message (uCommand = 8) before the driver has a chance to process this message.

## Styles

Each control has a text font, size, and style associated with it. This information is found, respectively, in cTxFont, cTxSize, and cTxFace after calling GetCtl, and can be reset from within ViewIt's edit mode via its Style menu.

ViewIt's StlCtl command can be used by a program to directly reset the text font, size, or style of an existing control (see "Control Commands" topic). This is equivalent to using ViewIt's Style menu in edit mode, and results in redrawing the control.

## Colors

Each "part" of a control can be a different color. With the introduction of System 7, Apple defined 15 distinct control parts corresponding to the first 15 items in ViewIt's Color menu. Most control drivers support the first three of these colors: frame, body (background), and content. The "System" or default colors for these parts are black frame, white body, and black content.

If the "System" item is checked in the Colors menu, then the control uses the default colors and its cColors handle will be nil. If at least one color has been chosen from the Colors menu for a control part (even if it is the same color as the part's default color), then ViewIt creates a control color table for the control, a handle to which can be found in cColors (after calling GetCtl). cColors is a handle to a relocatable block that has the following structure:

```
6 bytes miscellaneous stuff
2-byte integer = number of entries - 1
2-byte integer = part number
6-byte RGBColor = part color
2-byte integer = part number
6-byte RGBColor = part color
...
```

where the "part numbers" for frame, body, and content are, respectively, 0, 1, and 2.

ViewIt's ClrCtl command can be used by a program to directly reset the part colors of an existing control (see "Control Commands" topic). This is equivalent to using ViewIt's Color menu in edit mode, and results in redrawing the control. This command requires that the control has a non-empty color table with the corresponding entries defined, which can be assured by directly setting these control colors using ViewIt's Colors menu when in editing mode.

The utility command GetFgC can be used to get a color from a control's cColors table (see "Color Utilities"):

```
Facelt(nil,GetFgC,0,-2,ord(cColors),2);
```

where "2" in this case is the part number corresponding to the control's content, and the color is returned in uRGB.

## Best Colors

For the best appearance across all types of Macintoshes, use relatively light body (background) colors (such as the yellow in this control), and darker content and frame colors. This will ensure that control backgrounds do not turn to black on black-and-white screens, and that content and frames do not become white. The reverse use of darker backgrounds and lighter content and frames does not map

well to lower screen depths.

Also note that, when testing the display of control colors at different screen depths, the appearance of a control on a one-bit deep black-and-white device with Color QuickDraw installed will not always be the same as its appearance on a Mac without Color QuickDraw (such as a Mac+). Always check the appearance of colored controls at varying screen depths, in both color and non-color windows, and on older Macs without Color QuickDraw.

## FSSC Support

ViewIt control drivers can optionally provide support for ViewIt's Font, Size, Style, & Color (FSSC) menus. Note that this support is not related to choosing a control's overall Font, Size, Style, or Color when in edit mode (discussed above), but rather to the user choosing styles within a control's content when not in edit mode (like the multiple styles seen within this help text).

Controls that support the FSSC menus have the "Supports FSSC" option checked in ViewIt's Control dialog. A driver's on-line help will also note whether it supports the FSSC menus. The only controls accompanying ViewIt that support the FSSC menus are HelpCt (as shown by the style menu in this window) and BaseCt's editable text control (see "FSSC Selection" in "Editable Text" topic in BaseCt's help).

Assuming that you are using a control that supports the FSSC menus, how can such menus be made available to the user? On DoInit, ViewIt initializes the FSSC menus from MENU resources 1216-1219 as non-main menus. These menus have menuIDs 196-199, respectively, and can be attached to hierarchical menu items in any other menus. The style menu (italic "A") at the top of this window, for example, is linked to MENU 1235 which has 4 hierarchical items linked to menuIDs 196, 197, 198, and 199. When a control that supports the FSSC menus becomes selected (such as this HelpCt control), then the FSSC menus become enabled and their operation and appearance are handled by ViewIt and the control driver.

## ID/RefCon

The "Item ID" set in the Title or Control dialog is a very useful way of identifying controls (as was discussed in the "Windows" topic). This value is actually stored as a 4-byte "RefCon" integer in the FWND, FVEW, or FCTL resource from which the control is created. This RefCon is used to set the initial value of the "contrlRfCon" field in the control handle associated with the control. A copy of the RefCon is also stored in a separate block associated with the control. The program is then free to use the "contrlRfCon" field to store program-specific information without affecting the copy of the RefCon that contains the item ID (use the toolbox call "SetCRefCon" to do this).

The above scheme makes efficient use of the single 4-byte RefCon stored with each control in FWND, FVEW, or FCTL resources, while accommodating programs & custom CDEFs that use the RefCon and/or contrlRfCon field to store special information. When using the RefCon or contrlRfCon to store such information, their current values can be found in the cTmplRefCon and cRefCon variables, respectively, after calling GetCtl to get control info. In most cases, however, you'll simply be using the control's RefCon as its item ID, and can find this ID in wiHit after hits in enabled items, or in ciIndex after calling GetCtl to get control info.