V4. Views

As described in previous topics, "views" are a special type of control that contain other controls. This window contains four views and several controls within each view. If you were to enter edit mode and drag a view, then the controls within it would also be moved. Similarly, if a view is hidden, then all of the controls within it are also hidden. Thus views make it possible for programmers to quickly move, hide, scroll, add, etc., groups of controls within ViewIt windows.

About Control Drivers

Views and controls in ViewIt windows are either supported by old-style CDEFs (control definition resources) or the new "control drivers" introduced with ViewIt. Control drivers can be much more powerful than CDEFs since they are sent many more events and messages and can store much more information with each control. Most of the controls found in the Import ("+") menu when in edit mode, for example, are supported by control drivers.

The following control drivers are shipped with ViewIt:

Name	BaseID	Use
ViewBV	1220	views & scrollable controls
BaseCt	1310	basic controls
PeekCt	1390	debugging (memory display)
ScrapCt	1800	scrap (clipboard) display
HelpCt	1900	on-line help text

Each of these drivers includes example controls that appear in ViewIt's Import menu (i.e., they include example FVEW or FCTL resources that ViewIt adds to its Import menu).

Each control driver includes its own on-line help which is available by either choosing the driver from the "Drivers" menu in the ViewIt Help window, or by hitting the "Driver Help" button in the Control dialog when editing a control that belongs to the driver. The type and amount of information presented in the driver's help is defined completely by the control driver. For simple drivers like PeekCt, the help presented is quite simple, but for more powerful drivers like BaseCt that support a large number of different control types, or more sophisticated controls, the help will often consist of multiple topics.

The View Driver

ViewIt views are supported by a special control driver named "ViewBV" which is part of ViewIt. The following topics describe ViewBV and the view-related features that it supports. ViewBV is the only driver that is used to support views (which is why we made it an integral part of ViewIt).

• Operation

ViewBV views are typically added to ViewIt windows from the FVEW resources listed under "View" in the Import menu. These views should always be type "Static" (set in Control dialog).

Each ViewBV view can contain any number of "daughter" (non-view) controls. These daughter controls will appear to be drawn above the view's background, but below and within the view's frame.

ViewBV also supports the automatic checking/unchecking of daughter check boxes and radio buttons. Check boxes are checked/unchecked in response to clicks within them or equivalent command key presses. Similarly, hits in radio buttons cause them to be checked, but also cause all adjacent radio buttons to be unchecked ("adjacent", that is, with respect to their position in the list of controls, not their screen position). Thus the standard behavior of radio buttons is obtained by simply placing the buttons adjacent to one another in the control list.

• Scrolling

ViewBV takes full advantage of ViewIt's support for "hand scrolling" of control contents (described in the "Controls" topic). If the Max V or Max H dimensions of the view's Content (set in the Bounds dialog) are not zero, then the view may have a content area that is larger than the visible content (i.e., part of the view may be scrolled out of view, hiding some controls). If the hand icon in the Bounds dialog is also checked, then such a view can be hand scrolled by dragging its contents. Such views can also be hand scrolled when in editing mode (whether the hand icon is checked or not) by OPTION-dragging the control's contents.

ViewBV also supports the use of scroll bars with any view that has a content area larger than the visible content. To add a vertical scroll bar when Max V > 0, add 1 to the view's VarCode (set in Control dialog).

To add a horizontal bar when Max H > 0, add 2 to the view's VarCode. Each bar is drawn as part of the view's frame, so you must also set the view's corresponding Bottom or Right Indent to at least 15 pixels to make room for the scroll bar. Scroll bars can be used with or w/o hand scrolling, and vice versa.

When the active state of a window changes, ViewBV will automatically change the active state of the scroll bars in a view if the "Active" flag for the view is set in the "Messages Sent" menu (in Control dialog).

TIP: If a scrollable view is also attached to a window edge, then make the Max H or Max V corresponding to the side that is attached a multiple of the "Δ" scroll increment entered in the Bounds dialog. This ensures that the window will display the view's full content area when expanded to its maximum size, otherwise a few pixels will be left out of view to support scrolling in "Δ" increments (see "Controls" topic for further information about scrolling and attaching controls).

• Content (Advanced)

Views typically do not have any "content" that is drawn above the view's body and below its frame (other than their daughter controls). The main program can optionally attach a program procedure to a view that will be called to draw its content when the control is being redrawn. This "content" will appear below any daughter controls within the same view, and is useful in cases where the program needs to include drawing within a view that would be difficult to do with individual controls. (In cases, however, that require relatively simple drawing, you should simply add a corresponding control to the view that fills its content area. A basic control linked to a PICT or PAT#, for example, could be used to draw a picture or pattern.)

The address of the drawing procedure should be placed within the "contrlAction" field of the view's control record. This can be done by either directly setting the "contrlAction" field or via the toolbox call "SetCtlAction":

```
/* C, C++ */
 SetCtlAction(fRec.cControl,(ProcPtr)MyDrawProc);
Pascal
 SetCtlAction(fRec.cControl,@MyDrawProc);
```

where fRec.cControl is the view's control handle obtained from a call to GetCtl, and "MyDrawProc" is the name of the program procedure that will draw the view's content. Note that this drawing procedure must be located within a nonrelocatable code segment as long as the control exists.

Content-drawing procedures must be of type "Pascal" and have a single, 4-byte parameter. This parameter will be the address of the fRec record, but main programs will typically have access to fRec as a global record, so the passed parameter can be ignored.

```
/* C, C++ */
 pascal void function MyDrawProc(FacePtr fPtr);
Pascal
 procedure MyDrawProc(fPtr : FacePtr);
```

On entry, the "c" variables in fRec will contain information about the view (as if you had called GetCtl), and the window's clip region and port characteristics will be set to correspond to the control's current font, size, style, and color settings. The procedure should simply draw the view's contents into the control's cContent rectangle, which may be larger than the visible content area if the view is scrollable (i.e., you should offset all your drawing to fit the current cContent rectangle, regardless of its position in the window).

The content-drawing procedure must preserve the contents of fRec since these are used again by the view driver. If you need to make FaceIt calls within the drawing procedure that will clobber fRec, then simply save the control handle on entry and call GetCtl for the view before leaving the procedure:

```
 ...
 saveView := fRec.cControl;
 [do stuff that clobbers fRec]
 FaceIt(nil,GetCtl,0,0,0,ord(saveView));
 ...
```

DRAWING TIP: If the view's content area is much larger than its visible area, and drawing this entire content area would be too time-consuming, then you can "logically clip" your drawing to just the visible content area within cContent (i.e., you only draw stuff within the visible area). This visible area can be found in cClip. To further restrict drawing to only the area needing updating, use the intersection of cClip with the bounds of the window's visible region, cOwner^.visRgn^^.rgnBBox.

MIXING PROC.S: The form of the drawing procedure (with one 4-byte parameter) is the same as that

used for many other FaceWare callbacks. If you are using a single procedure to support more than one type of callback, then the variable uCommand can be examined to determine the type of callback being made. uCommand = 1221 if the procedure is being called to support content drawing in a view.

• Instructions

Special instructions can be placed in the "Instructions" string that is associated with each ViewBV view (set on page 2 of Control dialog). These instructions have the format,

a©[control numbers]=[activation condition]
s©[control numbers]=[show/hide condition]

where "a" denotes an "activation" instruction that is used to activate/inactivate controls, and "s" is a "show" instruction used to show or hide controls.

The "control numbers" are 2-digit numbers (the number of the control in its parent view) separated by commas or periods, where a period is used to denote a range of control numbers. The controls designated by these numbers will be activated/inactivated or shown/hidden according to whether the condition is true or false.

The "condition" is of a set of 2-digit control numbers separated by "+" or "-", where "+" indicates that the corresponding control must be checked for the condition to be true, and "-" that the control must be unchecked. Also note that controls specified in a "condition" must be check box or radio button-type controls.

Any number of activate and show instructions can be added to a view's instruction string if each of these is followed by a carriage return. For example,

a©07.10,12=+15-16+23
a©12,44=+34
s©44=+34

would inactivate controls 7-10 and 12 unless control 15 is checked and 16 is unchecked and 23 is checked (AND logic). Control 12 can also be made active if control 34 is checked (OR logic), and control 44 will be both inactive and hidden unless control 34 is checked.

Instructions are evaluated whenever a control in a view is hit (by clicking or command key), and when SetVal is called for all controls in all views (c = d = 0). Calls to SetVal or SetCtlValue for specific controls do not result in executing view instructions.

• View Types

Several example FVEW view resources can be imported from the Import menu when in edit mode. A brief discussion of these FVEWs is presented here.

The "Plain" view with its solid background and no frame is commonly used within simple dialog windows. TIP: If the plain view is to act as a fixed background in the window, you can attach it to the right and bottom sides of the window (via Bounds dialog) and lock its position (via "Lock" menu) so that it can't be accidentally moved when in edit mode.

The "Framed" view is commonly used to support paging of groups of controls (multiple overlapping framed views are hidden and shown in succession using ShoCtl) or to break up the contents of a window into groups of controls.

The "Scrollable" view offers another way to reduce clutter by supporting the scrolling of groups of controls. Controls in such views can also be dynamically manipulated using AddCtl (to add controls), DspCtl (to remove controls), and ScrCtl (to scroll or resize view's content area).

The "Transparent" view can be useful in cases where there is a reason to place one visible view above another, and the underlying view's background or controls must be seen.

• Attached Views

All view types can be optionally "attached" to the right or bottom sides of the window, and in turn can have daughter controls attached to themselves. This window, for example, has its first view attached to the bottom and right sides of the window, and a help control attached to this view. If the window is resized, then ViewIt automatically resizes the view to fit the window and the help control to fit the view. Further info about "attaching" controls to support growing and zooming can be found in the "Controls" topic.

• Paged Views

Views are often used to support the "paging" of groups of controls. In most cases this is done by simply hiding/showing successive views of the same size that overlap one another in the window. Alternatively, the ViewIt commands DspCtl and AddCtl could be used to dispose of one view and dynamically add another, but this would be slower than hiding and showing.

When creating such views from within edit mode, you'll need to often hide one view and show another. This can be done by either using the "Hide" menu item to hide one view and then clicking on the view number of a hidden view in the controls bar to show it, or by selecting the view to be hidden and then OPTION-clicking on the view to be shown in the controls bar.

• Colored Views

When using views to fill a window with background color(s), you may notice that the window is first erased in white before controls are redrawn. If this white "flash" is annoying, you can prevent ViewIt from erasing in white by either 1) setting a window background color (in Window dialog), or 2) making the first view in the window a solid view that completely fills the window's content area. The latter trick works even if the first view is hidden.

• Limitations

All controls specified in view instructions must be in the same view, and views themselves cannot be shown/ hidden or act./inact. with instructions. To do the latter, or for creating more complex interactions between controls, use commands such as ShoCtl and ActCtl, or toolbox calls.

ViewBV provides no support for displaying title text, no support for colors beyond frame, body, & content, and no support for data linking, resource linking, control values, and other features common to non-view controls.