

---

# Appearance: Not Just Another Pretty Interface

*The Macintosh was once the model of consistency: every application behaved and looked the same, making the user feel at home. But that consistency faded as, due to lack of support from the Macintosh Toolbox, developers created custom controls, menus, and windows, moving forward (or sideways) with user interface innovations while the Mac OS lagged behind. Now the Appearance extension takes the first step toward regaining that consistent look and feel we all remember so fondly, paving the way for switchable interface themes and making it easier to develop applications for the Mac OS.*



**EDWARD VOAS**

You've got this great idea for a user interface, but it means you have to write a slider CDEF. So you plug away, working to replicate what you've seen in dozens of applications, while dreaming of sliders that are available as part of the system. Well, your dream has come true. Meet Appearance, the biggest advancement of the Macintosh user experience since System 7. The Appearance extension provides sliders plus a lot more:

- Appearance implements a new look — Apple Grayscale — which was originally slated to be the default look of Copland, the former Mac OS 8 plan. Under Appearance the standard system windows, controls, and menus all have the Apple Grayscale look.
- Appearance adds new controls such as progress bars, tabs, disclosure triangles, and sliders to the standard set, eliminating the need for developers to roll their own.
- Appearance extends the Window, Control, Dialog, and Menu Managers to provide functionality that's necessary for some of the new features to work correctly. Some of the new functionality fills in the gaps that developers have had to fill in on their own because the Macintosh Toolbox didn't support some necessary or desirable features. For example, the system MDEF now supports extended keyboard modifiers for menu item keyboard equivalents.

With Appearance you benefit the most by using as many of the system-supplied user interface elements as possible. Your user interface won't have a patchwork look — with the system elements, all pieces of the UI blend together nicely. Plus, as Apple

---

**EDWARD VOAS** (voas@apple.com) is a staunch supporter of Truth, Justice, and Switchable Themes. He is currently the Technical Lead on Appearance and is the co-author of the popular shareware program Aaron. When Ed is

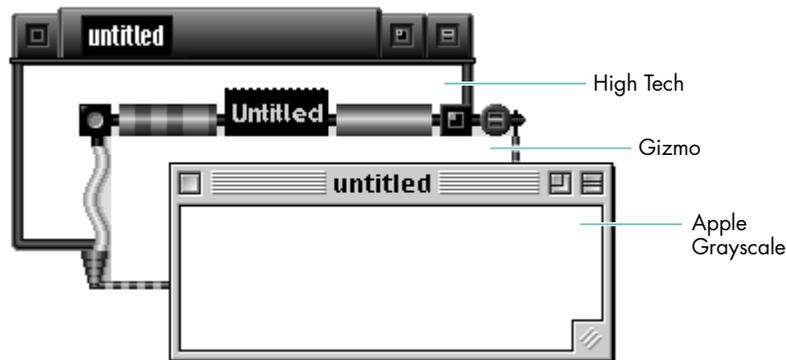
not busy coding, he is hard at work memorizing lines from the Star Trek movies and boring his coworkers with inane facts from those movies ("What's the prefix code of the U.S.S. Reliant?"). •

enhances the UI elements, your applications can immediately (and automatically) benefit as new system versions are released. This becomes particularly important as we move toward switchable themes (explained below). Another major benefit is that your applications can be smaller, because you don't need to implement UI elements that are now supplied by the Toolbox. A sample application that demonstrates Appearance accompanies this article on this issue's CD and on *develop's* Web site.

A beta version of the Appearance extension is provided on this issue's CD as part of the Appearance Software Developer's Kit (SDK), which also includes a control panel, several header files, and some shared libraries to link with. The Appearance extension is intended to be bundled with applications for users to install on Mac OS 7.6 or earlier (later systems will contain the functionality of the SDK as part of the base system). Your application can determine whether Appearance is running by checking a Gestalt selector (`gestaltAppearanceAttr`). This selector returns a bitfield indicating which features of Appearance are in effect.

## THE ON-RAMP TO THEMES

You may have heard about themes at Apple's Worldwide Developers Conference (WWDC) or read about them in discussions on Copland. Essentially, a *theme* is an interface look that spans all elements of the user interface and ties them together with a certain graphic design. Themes are data driven — all the data that describes the theme interface is contained in a theme file. The data-driven aspect makes it easy to switch themes on the fly. Figure 1 gives examples of three themes that were shown at the 1996 WWDC and elsewhere: Apple Grayscale, Gizmo, and High Tech.



**Figure 1.** Three themes under Appearance

Now before you get too excited, please note that the theme-switching mechanism isn't implemented in the first version of Appearance; however, switchable themes are very much a part of the future of the Mac OS. Appearance is the first step toward that future, and using the system controls, windows, menus, and other features provided by Appearance will allow your application to handle theme switches automatically when the time comes. I'll be referring to themes throughout this article, especially when talking about the Appearance Manager, which lets you get colors and patterns for the current theme.

## WHAT'S NEW AND IMPROVED

Appearance redesigns some old controls and provides many new ones. Windows sport a new look and added features, and there's a new help icon. Here we'll check out these snazzy Apple Grayscale UI elements and learn a bit about the new features. Then we'll look at the Appearance Settings control panel, which lets the user control

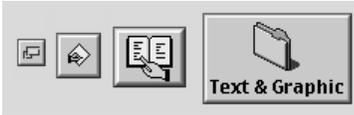
---

theme variations and the system font, among other things. In later sections, we'll describe how the new features work and what you need to do to adapt your applications for Appearance.

## NEW CONTROLS

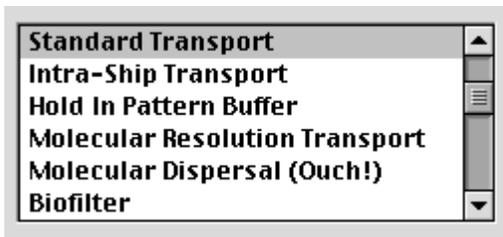
Many new controls are added to the system with Appearance.

### Bevel button control



The bevel button control implements a rectangular button with a beveled edge. Typically, a bevel button displays an icon, but it can display an icon, a picture, or text, singly or in combination. You can also attach a menu to this control. Multiple bevel thicknesses and several different button behaviors are supported, to suit just about any use. This versatility makes bevel buttons well suited for use in toolbars or tool palettes. (These buttons should never be used to replace push buttons, however.) The sample application accompanying this article shows many different variations of bevel buttons; you'll be astounded by the possibilities.

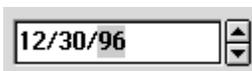
### List box control



The list box control implements a simple list box. It requires an auxiliary resource of type 'lides' to specify the features of the list, such as the number of columns and rows. (The Appearance Settings control panel, shown later, provides an example of columns in a list box.) This control allows filtering of keyboard events, and handles the default keyboard navigation you'd expect from a list box.

Note that if you use the list box control in a dialog, it won't respond to keyboard events — you can't tab into it, for example — unless the dialog has established an embedding hierarchy (described later). There's a similar restriction on the clock and editable text controls.

### Clock control

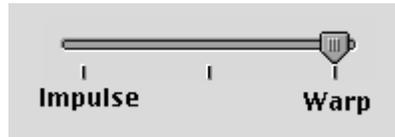


The clock control provides an editable date or time field, as you'd find in the Date & Time control panel. The little arrows next to the clock are part of the clock control. (Remember, you won't be able to tab or type characters into the clock control unless the dialog has established an embedding hierarchy.) You can also specify a noneditable version that simply shows the date or time. The noneditable clock permits live updating, so you can put a clock in your interface and let it tick

---

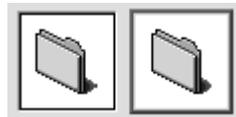
away. The clock control uses the date and time formats set in the Date & Time control panel.

### Slider control



Sliders are finally part of the Mac OS repertoire. You can choose which way the indicator faces or use a nondirectional indicator. You can also specify whether to draw tick marks. The slider supports live feedback (described later).

### Image well control



This control holds an icon or picture. In Apple Grayscale, image wells look like an editable text box with a picture inside instead of text. Image wells have a normal and selected state, as shown above. A good use for an image well is as a drop target for images or document icons. Drag and drop support isn't built into this version of Appearance.

### Little arrows control



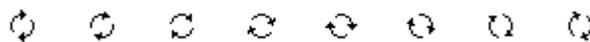
The little arrows control implements the little up and down arrows you often see tied to a box displaying a value. The arrows are used to increase or decrease the value. In the Memory control panel, for example, you click them to set the cache size or virtual memory partition size.

### Progress bar control



Progress bars are now part of the standard control set. You can tell a progress bar to switch into *indeterminate* mode, in which it displays an animated barber pole–like bar; you might use this mode to indicate that you haven't made a connection yet or are waiting for some piece of data before continuing. Because the indeterminate flag is separate from the value, you can switch back and forth without affecting the value.

### Chasing arrows control



These are the spinning arrows that usually indicate an asynchronous process. In other words, there's something going on in the background but you can continue to work. You've no doubt seen them in Find File when searching for files.

---

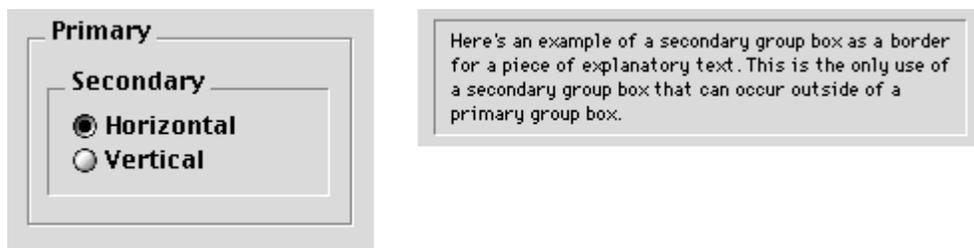
## Tab control



Currently, the new tab control supports only one row of tabs running along the top of the control. Future versions will support more variants. As with the list box, you use an auxiliary resource to specify the tab names and any icons that appear beside the names.

You should try to restrict your use of tabs to those times when they're really necessary. Too many tabs can result in a very complex and confusing UI. Also, tabs can be difficult to localize for different script systems: the width of a text string can increase up to 50%, causing problems if you've set up the tabs to fit perfectly in a Roman script system.

## Group box control



The group box control implements two group box looks — primary and secondary, as shown above. It also provides three different types of titles for the grouped items — text, checkbox, or pop-up menu. The pop-up title is useful for paged interfaces, such as the Sound and Speech control panels. You can embed other controls within the group box control, such as radio buttons.

Primary groups should be used as the first level for grouping items. Secondary groups should always be inside primary group boxes. The only exception to this is if the group box is being used for a border around some text, as shown above.

## Disclosure triangle control



You can use the disclosure triangle control in places where you want to hide some information to reduce clutter but still give the user a way to view it. The user clicks the disclosure triangle to expose the hidden information. The Finder uses disclosure triangles next to folder icons when in list view.

## Window header control



The window header control is what the Finder uses to draw the header of a window where text such as the disk information is shown. Text sold separately.

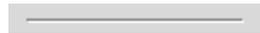
---

## Placard control



The placard control implements a small panel like those often found at the bottom of a window to the left of the horizontal scroll bar. You might have seen them in CodeWarrior. You could use a placard control to show information such as the current line number, as shown above. In Apple Grayscale, placards look the same as the window header control, but this won't be the case in all themes, so be sure to use the appropriate control.

## Visual separator control



The visual separator control implements a simple divider line.

## CONTROLS TO REPLACE DIALOG PRIMITIVES

Dialogs can contain many items that are not controls — pictures, icons, editable text, static text, and user items. I'll be referring to such items as *dialog primitives*. Some of these dialog primitives now have control counterparts. For example, the counterpart of the editable text primitive is the editable text control, which is a control with all the functionality of the editable text primitive. In referring to items in dialogs, the word *control* refers only to items that are *not* primitives. Using the new controls allows you to take advantage of features provided by Appearance's embedding hierarchy.

If a dialog has an embedding hierarchy, the controls discussed in this section automatically stand in for their primitive counterparts. So the easiest way to use the controls is to let the Dialog Manager convert your primitives for you. You don't have to change your items into resource-based controls and create 'CNTL' resources to use these controls (though you could if you had your heart set on it).

## Editable text control



The editable text control replaces the old editable text dialog item. Because it's a control, it can be enabled and disabled. If you've ever tried to disable an editable text field in a dialog, you know how difficult this can be. With Appearance it's one line of code. The editable text control allows keyboard-event filtering and supports password entry. (Like the list box and clock controls, if you use the editable text control in a dialog, it won't respond to keyboard events unless the dialog has established an embedding hierarchy.)

The text in the control is always displayed on a white background in the Apple Grayscale theme. The figure above shows three editable text controls, one using the password entry variant. The control accepting passwords has the current keyboard focus, as indicated by the ring around it. (Keyboard focus is described later.)

---

## Static text control

Online Offline

Static text controls replace the old static text items for embedding static text in dialogs. Since they're controls, they can be deactivated and will then be drawn disabled like other deactivated user interface elements. If you have a dialog with static text items in it and establish an embedding hierarchy for that dialog, the static text primitives will automatically become static text controls.

## User pane control

The user pane control replaces the old user item construct in dialogs. This control is essentially a stub control that calls user-installed procedures to do its drawing, hit testing, and other things that controls do. It also lets you track the mouse and draw with the correct highlighting — normal, highlighted, or whatever. Even in its most basic form, the user pane is very useful because you can embed controls within it, which allows you to group items. When you hide, show, enable, or disable the user pane, the group of embedded controls will follow suit automatically.

## Icon control and picture control

These controls were created to stand in for icon and picture primitives in dialogs that have an embedding hierarchy. The icon control allows you to display icon suites, as well as the usual 'ICON' and 'cicn' icon types. When used to simply replace the old icon or picture primitive in dialogs, these controls don't track the mouse (they behave as icons and pictures always did in dialogs). You can create an icon or picture control and add it to a dialog through a 'CNTL' resource if you'd like it to track the mouse.

## CHANGES TO OLD CONTROLS

Some old controls have a new look, more features, or both.

### Push button control



Button controls — now called *push button controls* to distinguish them from bevel button and pop-up button controls — have been changed to allow you to set a “default button” flag. When set, this flag tells the control to automatically draw the default ring (border) around the push button. The default ring is drawn outside the control's bounding rectangle. The default push button has a new look, as shown in the last two examples above.

### Radio button control

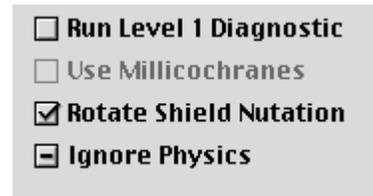


Radio buttons have a new look. They also support a *mixed* state. A mixed state is typically used when a radio button represents a selection that includes more than one state. Let's say you have a group of radio buttons that represent different planet classes: Class M, Class P, and Class K. As a user clicks a planet in a list, the radio buttons reflect the class of the selected planet. Now suppose the user selects four planets from the list box at once — for example, one Class K and three Class M planets. You can reflect this situation by putting the Class K and Class M radio buttons in a mixed state, indicating that some of the selection are of this type, but not all.

---

The `kControlRadioButtonMixedValue` constant is for setting the mixed value of a radio button. You can use it in a call to `SetControlValue` to set a radio button to the mixed state. Your radio button must have a maximum control value of 2 for this to work, since that's the value of the constant.

### Checkbox control



Our old friend the checkbox sports a new look and real checkmarks. The “X” variant of this control is still available for use in countries outside the U.S. Like radio buttons, checkboxes support a mixed state, as shown in the Ignore Physics checkbox above. The constant for the checkbox mixed value should look vaguely familiar — it's `kControlCheckBoxMixedValue`.

### Scroll bar control



Scroll bars have a new look, too. One scroll bar variant supports live scrolling.

### Pop-up button control



The pop-up button control is the old pop-up menu control with a new look.

### THE NEW HELP ICON

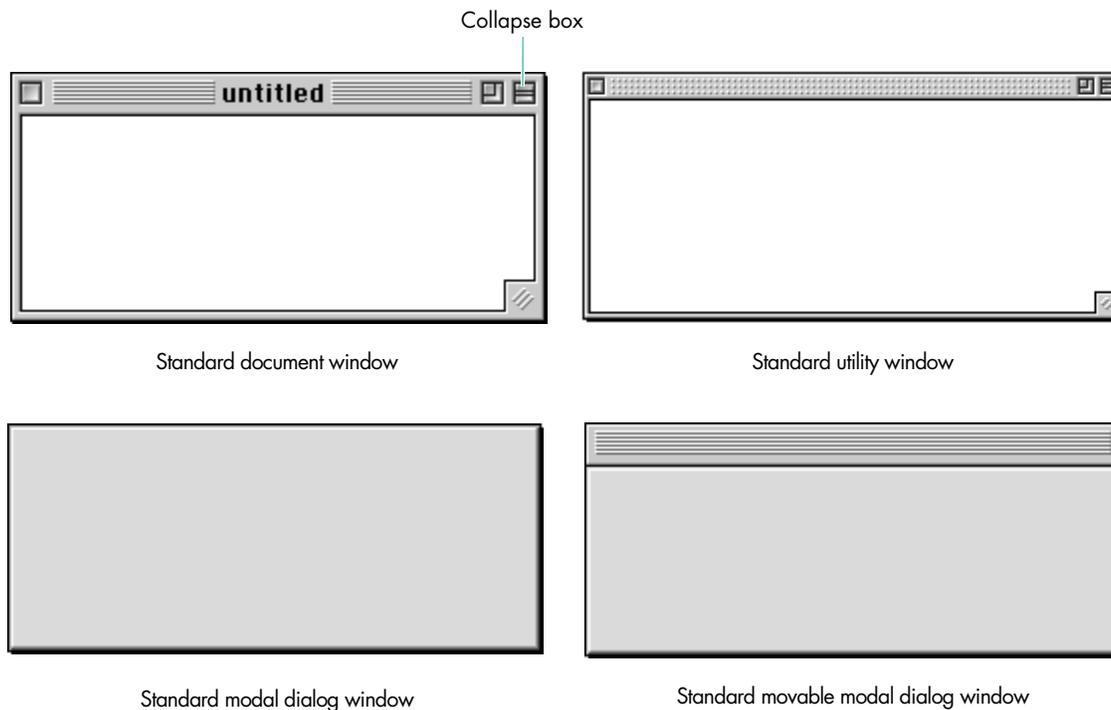


There's now a system-supplied help icon. You can combine this with a bevel button to get the new standard help-button look. The `StandardAlert` routine (described later) uses this approach to display its help button.

### CHANGES TO WINDOW APPEARANCE

Appearance gives windows a new look, as shown in Figure 2. (Note that floating windows are now called utility windows.) The standard document window has a thicker border than it used to. This border is functional: it allows you to grab the window and drag it from any side. Alerts are distinguished from dialogs by a red border; the new movable alerts also have a reddish title bar (giving new meaning to the term *red alert*). Remember to use alerts only to warn the user of something or to present important information; in all other cases, use a dialog. The many window variants are described later.

Document windows (and utility windows) also have a new element: the collapse box. Clicking it collapses the window into a title bar. You may wonder why the new



**Figure 2.** Standard Appearance windows

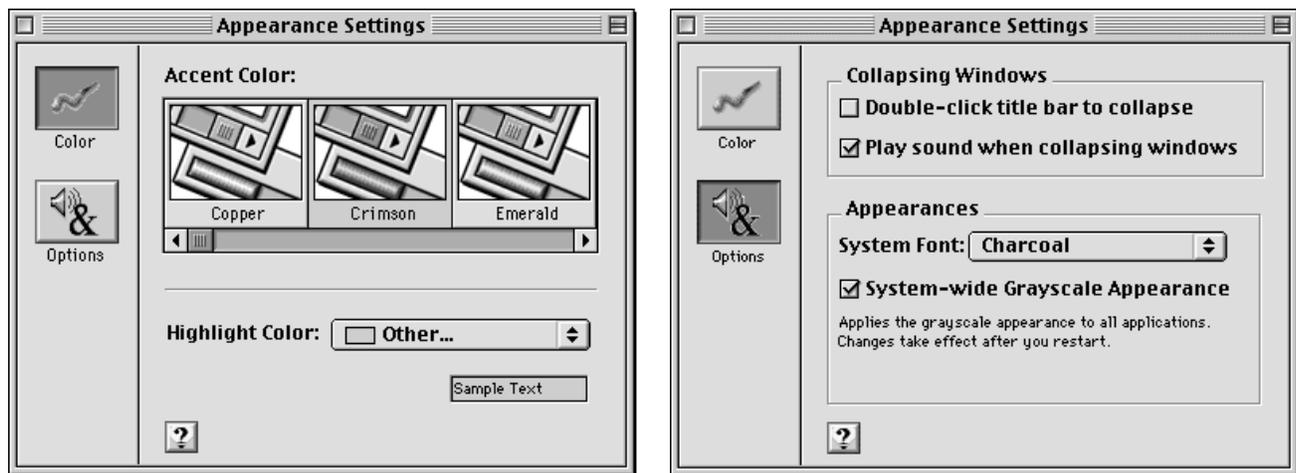
collapse box is where the zoom box used to be and the zoom box has moved over. For consistency, the most frequently used widget is always on the outside — in this case, the collapse box. (More windows have collapse boxes than have zoom boxes, and no window can have a zoom box without a collapse box.)

Also, the size box is now integrated into the frame of the window. If you're adapting these new windows directly (as described later), the size box is drawn for you automatically. You don't need to call `DrawGrowIcon`.

### THE APPEARANCE SETTINGS CONTROL PANEL

The new Appearance Settings control panel (Figure 3) allows the user to control these settings:

- The accent color of the current theme. This affects the coloring of menu items as they're chosen, scroll bar and slider indicators, progress bar indicators, and focus rings.
- The highlight color. This affects the coloring of any highlighted item — for example, selected text.
- Window collapsing. When Appearance is running, the WindowShade control panel is removed. When “Double-click title bar to collapse” is checked, the user can double-click the title bar in addition to using the collapse box to collapse a window.
- The system font. The Apple Grayscale system font is Charcoal, but users can choose the Chicago font if they want.
- System-wide Grayscale Appearance mode. When “System-wide Grayscale Appearance” is checked, it means that every application gets the Apple Grayscale look; otherwise, only applications that have explicitly adopted Appearance's features have the look.



**Figure 3.** The Appearance Settings control panel

## NOT YOUR FATHER'S CONTROL MANAGER

To make the new controls work correctly, it was necessary to add some features to the Control Manager. The Control Manager now does the following:

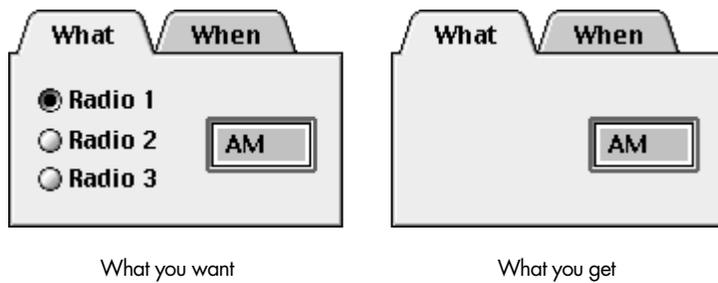
- supports embedding, to make drawing order and hit testing predictable and enable some Appearance features
- supports keyboard focus for controls
- makes it easy to get and set control data and choose fonts for control titles
- supports live feedback for scroll bars and sliders
- provides a mechanism for controls to advertise the features that they support

### DRAWING ORDER AND HIT TESTING

From the very beginning, the order in which controls draw in a window has been backwards due to how the Control Manager manages the control list for a window. As controls are created, they're added to the head of a window's control list. When the controls are drawn, the list is traversed, yielding a drawing order opposite to the order in which they were added to their owning window. To confuse things more, in normal dialogs, dialog primitives, such as editable text and static text items, are always drawn from first to last. When you have a mixture of controls and dialog primitives, the primitives are drawn from first to last after all the controls are drawn from last to first.

This isn't a problem if you can assume that controls don't overlap or contain other controls. With new controls such as tabs and group boxes, however, you can't make this assumption. Consider the case where you want a tab control containing three radio buttons and an editable text field. Let's say you add them to the 'DITL' resource in this order: first the tab, then the three radio buttons radio 1, radio 2, and radio 3, and then the editable text item. When they're drawn, you get this order: radio 3, radio 2, radio 1, tab, editable text. You've just covered up your radio buttons with the tab control (see Figure 4)! This happens because controls are drawn first, and then dialog primitives. Needless to say, trying to manage the drawing order can be difficult.

Hit testing has similar problems when items are inside tabs, group boxes, and other such controls. The FindControl routine uses a linear search over the control list to find the first control that returns a part code other than kControlNoPart. This isn't always accurate, as disabled controls are skipped even though they were hit.



**Figure 4.** The problem with control drawing order

FindDialogItem also uses a linear search of the dialog items, and stops when it finds the first enabled item that a given point is in. Consider what happens with that tab control with the three radio buttons if your application calls FindDialogItem with a point that's in one of the radio buttons. FindDialogItem never finds the radio button because the first item searched is the tab control. The point is certainly within the tab control (the radio button is inside the tab), so FindDialogItem returns the dialog item number of the tab control. The right approach is to do an “inside-out” hit test to find the most deeply nested control hit by the mouse. Read on to see how Appearance makes this possible and simplifies managing the drawing order.

### THE EMBEDDING HIERARCHY

To make drawing order and hit testing predictable and easy to follow, we've added an *embedding hierarchy*, where controls can be embedded within other controls, giving you a rudimentary “view system.” Embedding is a really exciting aspect of Appearance. The hierarchy ensures that parent items are always drawn before their children. It also helps hit testing since the hierarchy can be traversed quickly to find out which control the cursor is over. A hierarchy can exist in any window. It's not restricted to dialogs, though it's easiest to use there, since the Dialog Manager deals with focus management and event handling for you.

#### Root controls.

To enable control embedding in a window, you must create a *root control* for that window. The root control is the container for all other window controls. You create the root control in one of two ways: by calling the CreateRootControl routine or by setting a dialog flag to tell the Dialog Manager to create one for you (more on this later). You can't embed controls without a root control, and any attempt to do so will result in all the money in your bank accounts being transferred into mine. So watch it.

When a window has a root control, calls to NewControl (and GetNewControl) automatically add controls to the root of the window. Calling EmbedControl or AutoEmbedControl is the only way to explicitly change this. You use EmbedControl to specifically embed one control in another. The Dialog Manager uses AutoEmbedControl when creating items from a 'DITL' resource.

AutoEmbedControl uses visual placement to automatically determine what control, if any, a control should be embedded within, based on bounding rectangles. For example, going back to the tab and radio buttons in Figure 4, the Dialog Manager would have embedded the radio buttons and editable text field in the tab control for two reasons: they came after the tab in the 'DITL' resource and they fit inside the tab control. The ordering of 'DITL' items is still important — a control can be embedded automatically only in a control that already exists — but the results are a lot more predictable with embedding. Create your elements from back to front and

---

everything will fall into place. The sample code accompanying this article provides some good examples of how to do this.

### **Groups and latency.**

The embedding model has many advantages, one of which is that it makes it possible to treat a set of items as a group. By acting on a common parent, you can move, disable, or hide groups of items. Disabling the root control of a window, for example, will disable all items in the window.

Doing things like switching tabs becomes remarkably easy. You can simply use a blank user pane control as the common parent for all items in a particular “page” of a tab control. After creating as many user panes as you have tabs, you can just hide one and show the next when a tab is clicked. All the controls embedded in the user pane will be hidden and shown automatically when the user pane is hidden and shown. The sample code provides an example.

In hiding, showing, disabling, and enabling groups of controls, it’s important to preserve the state of an item when it’s hidden or disabled so that when its parent is shown or enabled, the item appears in that same state. To accomplish this, we’ve added the concept of *latency*. Controls are considered latent when they’re disabled or hidden only because their parent control is disabled or hidden. It’s effectively saying, “This item should be enabled (or shown) when its parent is enabled (or shown).” If you disable a control that’s latent, it becomes truly disabled and won’t be enabled when the parent is enabled. Likewise, if you enable a control whose parent is disabled or latent, the control becomes latent until its parent is enabled.

When enabling and disabling controls, to ensure that this latency information is always correct, you should use the new routines `DeactivateControl` and `ActivateControl` instead of just setting the highlight with `HiliteControl`, as you undoubtedly have always done in the past. It would be smart to use these routines even when no embedding or latency is involved — they’ll set the highlight code correctly and redraw the control. For controlling visibility, the old `HideControl` and `ShowControl` routines have been modified to deal with latency when an embedding hierarchy is present for a window.

### **FOCUS MANAGEMENT**

With `Appearance`, you can get and set the *keyboard focus* of a window. The control with the keyboard focus is the one that receives all keystrokes. For example, the `Dialog Manager` tests to see which control has the focus when a keyboard event is processed and sends the event to that control. It’s possible for nothing to have the focus, in which case the keystroke is simply discarded. As mentioned earlier, to indicate that a particular control has the keyboard focus, a focus ring is drawn around the control. (It’s a lavender ring by default, but the user can choose a different color in the `Appearance Settings` control panel.)

The keyboard focus routines introduced with the `Appearance` extension are available only when a root control has been created for a window. In windows with an embedding hierarchy, you can use several routines to get, set, advance, reverse, and clear the keyboard focus. The default focusing order is a simple linear progression through all the enabled, visible controls in a window. The order is based on the order in which controls are added to the window. This is the same approach as using the order in which editable text items appear in a DITL to control tabbing order. (Eventually, other system-supplied focusing heuristics may be available, such as spatial focusing that’s based on the visual placement and grouping of controls, not on the order in which controls are added.)

---

Currently, the controls that support keyboard focus are the editable text, list box, and clock controls. In future versions of Appearance every control that can receive user input will support keyboard focus (push buttons will, but visual separators won't, for example). You can plan for that day by ordering your controls so that the focus will move from one to the next in the order you desire and by making sure they have enough space around them to allow for focus rings. Focus rings are outset a maximum of three pixels from the control's bounding rectangle.

## GETTING AND SETTING CONTROL DATA

Developers need to get and set different attributes of a control. In most cases, these attributes are unique to a particular control. In the past, the only way to allow access to control-specific information was to create a handle to hold such data, place it in the `controlData` field, and publish the interface. A good example of this is the menu handle of a pop-up control. Unfortunately, this approach makes it hard to change the control implementation in future versions of the control.

In Appearance, we've added a mechanism by which controls can allow the outside world access to their specialized data without exposing how it's stored. This data access mechanism is the cornerstone of many of the new Appearance features, allowing you to get and set control fonts, user-pane callback functions, bevel-button image information, and other useful things. Two new CDEF messages implement the data access mechanism: `kControlMsgGetData` and `kControlMsgSetData`. To advertise that a CDEF supports these messages, it must return `kControlSupportsDataAccess` in its feature flags.

Each piece of information that a CDEF wants to provide access to is referenced by a *tag*. A tag is some constant that is meaningful to the CDEF and represents the data in question. For example, to get at the clear text of a password field (the actual password), you would use the tag `kEditTextPasswordTag`. To set the indeterminate flag for a progress bar, you would use `kProgressBarIndeterminateTag`.

Each tagged piece of data can be any data type. It might be a menu handle, a `UniversalProcPtr`, or a structure. It's up to the creator of the CDEF to define the tag and the data type for the data that's passed back and forth.

To get and set data, you use the `GetControlData` and `SetControlData` routines, which use a format similar to that used by the Collection Manager and Apple events. These calls are pointer based, and storage is always owned by the caller, which means that if you're trying to get the menu handle from a bevel button control, for example, you would pass in a pointer to a menu handle. To get a vital warp-engine matter/antimatter intermix ratio from a control (a `Fixed` value, for those who don't know), you might call `GetControlData` like this:

```
Fixed    theRatio;  
OSErr   err;  
Size    actualSize;
```

```
err = GetControlData(myControl, 0, kWarpIntermixTag, sizeof(theRatio),  
                    (Ptr)&theRatio, &actualSize);
```

## BETTER FONT CONTROL

Appearance allows you to set the font of any control title, independent of the system font or window font. Previously, your only choices were the system font and, if the control supported that variant (and most did), the window font. Now you can set the control title font to any font your heart desires.

---

There are also new constants available to take advantage of some system-defined fonts — big system font, small system font, and small emphasized system font. In the U.S. version of Apple Grayscale, those fonts are Charcoal 12, Geneva 10, and Geneva 10 bold, respectively. By using system-defined fonts, you can be sure to get the correct font when running with a different script system, such as Kanji. This helps your programs adapt automatically to different locales. Never, ever hard-code a font number or font size. I mean it!

To save you the hassle of setting control title fonts for each item after a dialog is created, we've added the new 'dftb' resource type for a dialog font table. The 'dftb' resource is used to specify the initial font settings for each item in a dialog. This resource runs parallel to a 'DITL' resource and has an entry for each item in a dialog. It should have the same resource ID as the 'DITL' resource for the dialog. Of course, you don't need to create a 'dftb' resource if you want to use the system or window font.

There's an old resource — the 'ictb' resource — that very few people seem to know about or use. The 'ictb' resource allows you to set the font information for editable text and static text items, but not controls. Since it's now possible (and desirable) to change the font of individual control titles, Appearance adds the new 'dftb' resource. The 'dftb' resource has a straightforward resource format, making it easier to create and maintain than the old 'ictb' resource. If a dialog with no embedding hierarchy has both a 'dftb' resource and an 'ictb' resource, the 'ictb' resource is used for the static text and editable text items only. If the dialog has an embedding hierarchy, any 'ictb' resource information is ignored.

### **LIVE SCROLLING**

As mentioned earlier, Appearance allows you to have real live feedback with scroll bars and sliders. With some variants of the scroll bar and slider controls you can set an action procedure to be called back as the indicator is moved. Each time the action procedure is called back, the value of the control will indicate what position the user has dragged the indicator to. Listing 1 shows a good example of how to install and use a live feedback callback.

Be sure you set the action proc to be a `ControlActionUPP`, not the `DragGrayRgnUPP` that normal indicator dragging uses. If an action proc isn't passed into `TrackControl` or set with `SetControlAction`, no live scrolling occurs — the control reverts to dragging a ghost of the indicator.

### **THE CONTROL FEATURES SET**

To get the set of features supported by a control, you can call `GetControlFeatures`. This new routine returns a bitfield in which each bit represents a feature the control supports, such as keyboard focus or data access. If you want to write a CDEF that supports any of the new features, the CDEF must respond to the `kControlMsgGetFeatures` message.

### **OLD DIALOG MANAGER, NEW TRICKS**

The new Mac OS user experience provided by Appearance calls for some new features to be added to the Dialog Manager. For example, you can now ask that a dialog's background fit the theme, that a root control be created for a dialog automatically when the dialog is created, or both. These features are requested through a bitfield where each bit represents some feature. This bitfield can be specified in two ways: through the new `NewFeaturesDialog` routine or in the new dialog and alert extension resources. In addition, the Dialog Manager provides an

**Listing 1.** Providing live feedback for a scroll bar

```
ControlHandle CreateMyLiveScrollBar(WindowPtr window, Rect* bounds,
    Sint16 value, Sint16 min, Sint16 max, Sint32 refCon)
{
    ControlHandle    control;

    control = NewControl(window, bounds, "\p", true, value, min, max,
        kStdScrollBarLiveProc, refCon);
    if (control)
        SetControlAction(control,
            NewControlActionProc(MyScrollBarAction));
    return control;
}

pascal void MyScrollBarAction(ControlHandle control, Sint16 part)
{
    Sint32    oldA5 = SetCurrentA5();

    if (part == kControlIndicatorPart)
        /* At this point, the value will be updated to properly reflect */
        /* where the user has scrolled to in our imaginary document. */
        ScrollToLocation(GetControlValue(control));
    SetA5(oldA5);
}

void TrackMyScrollBar(ControlHandle control, Point where)
{
    /* Assuming the mouse was clicked in the indicator of a control, */
    /* you'd call TrackControl like so: */
    TrackControl(control, where, (ControlActionUPP)-1L);
    /* You could equally as well have passed the action proc in here */
    /* instead of calling SetControlAction in CreateMyLiveScrollBar. */
}
```

enhanced standard alert routine and lets you request standard movable modal behavior for dialogs and alerts.

### NEW RESOURCE TYPES FOR DIALOGS AND ALERTS

To make it simpler and more straightforward to add the new flags, we've created two new resource types to hold extended information for dialogs and alerts: 'dlgx' and 'alrx'. These resources specify the feature bits and other information, such as a window title for movable alerts. You relate these resources to 'DLOG' and 'ALRT' resources by their ID. If you created a 'DLOG' resource with an ID of 128, you would create a 'dlgx' resource with an ID of 128 to add extended information. The Dialog Manager looks for a 'dlgx' or 'alrx' resource after reading in the 'DLOG' or 'ALRT' resource.

Those who want to create dialogs programmatically can use the NewFeaturesDialog routine. NewFeaturesDialog is identical to NewDialog (and NewColorDialog), except that it takes a flags parameter. These flags are the same ones that you would set in a 'dlgx' or 'alrx' resource.

## AUTOMATIC CONTROL CREATION

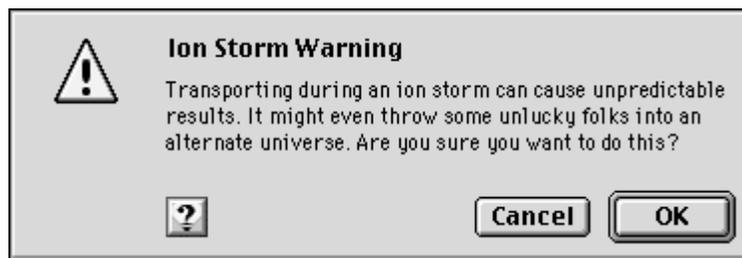
If you set the features flag for automatically creating a root control in a dialog, be aware that all dialog primitives are replaced with their control counterparts. The Dialog Manager routine `GetDialogItem` still works as it always did: if you call `GetDialogItem` on a static text item, you'll get back a handle to text, not a handle to the control. This call has been modified to sense when an embedding hierarchy is present and to talk to the controls to get the appropriate data. If you want to access the actual control for a static text item, you would use the `GetDialogItemAsControl` routine. You can then act on the item as you would with any control. The `SetDialogItem` routine works as it always did except for a few restrictions when the dialog has established an embedding hierarchy. In that case, you can't change the type or handle of a dialog item (except for user items, for which you can still set the drawing procedure).

Converting dialog items into controls makes it possible to do things you couldn't do before. For example, when all dialog items are controls, you can highlight, enable, and disable everything in a dialog, including static and editable text items. Now it's as simple as a call to `DeactivateControl` or `ActivateControl`.

In fact, when an embedding hierarchy is established in a dialog, if you deactivate the dialog, all dialog controls are automatically disabled. When the dialog is reactivated, the items are reactivated. This is the desired behavior and part of the new Human Interface Guidelines. Only the frontmost window should have active controls and other UI elements. This helps distinguish the active window at a glance. Everything else should fade into the background, so to speak, so that the user can concentrate on the window that's active.

## THE MOTHER OF ALL ALERTS

The new `StandardAlert` routine is possibly one of the most useful routines in `Appearance`. It allows you to specify the text of the alert and optionally some explanatory text. You can display up to three buttons with your choice of text, as well as a help button (see Figure 5). The alert auto-sizes itself based on the amount of text passed into it and also auto-sizes and places the buttons. `StandardAlert` makes it simple to generate alerts with the standard Mac OS alert look without using resources.



**Figure 5.** Alert generated by `StandardAlert`

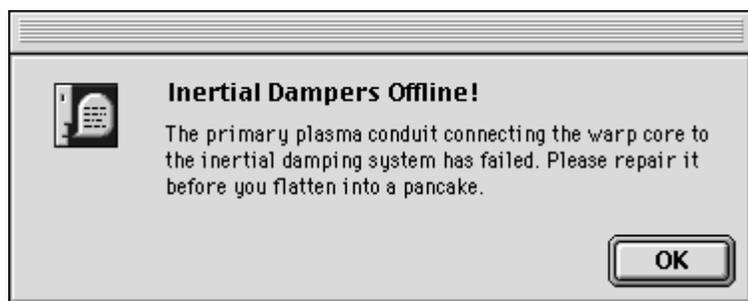
## MOVABLE MODAL DIALOGS AND ALERTS

`Appearance` provides a standard movable modal behavior. Instead of having to write your own code for handling movable modal dialogs, you can use the new movable-modal flag in the 'dlgx' resource (or call `NewFeaturesDialog`). The movable-modal flag tells `ModalDialog` to handle all the standard user interactions, such as dragging a dialog by its title bar or switching out of the application by clicking in another one. It's up to you to use the right window type (`kWindowMovableModalDialogProc`).

To allow your application to handle events while the dialog is up, you simply pass in a `ModalFilterUPP` as you do with `ModalDialog`. One major difference is that *all* events

are passed to your ModalFilterUPP for handling; this allows you to handle suspend and resume events when your application is either put into the background or brought to the front, as well as any other events you might want to handle. You could use this ability to allow your application to handle Apple events from other applications even though your application is in a modal state.

It's just as easy to make your alerts take advantage of this movable modal behavior (see Figure 6). All it takes is a quick flip of a bit (`kAlertFlagsAlertIsMovable`) in the 'alrx' resource. This gives you the same behavior as setting the movable-modal flag in the 'dlgx' resource.



**Figure 6.** Standard movable alert

### MOVING AND SIZING DIALOG ITEMS

The `MoveDialogItem` and `SizeDialogItem` routines were added to help you keep controls and dialog item rectangles in sync. If you call the old `MoveControl` and `SizeControl` routines on a dialog item, only the control is affected, making it easy to forget to make corresponding changes to the dialog item rectangle. The new routines affect both the control and its dialog item rectangle.

### WINDOW MANAGER ADDITIONS

Not only does the standard WDEF add a collapse box to the title bar when Appearance is running, but the collapse/expand mechanism is improved. Windows that support the new API actually remember that they've been collapsed (except after a restart), so when you hide and show an application the collapsed state of those windows remains intact.

The tracking of the collapse box is handled by the system for you. There's currently no mechanism available to allow you to intercept tracking, but we know it's useful and are working on a way to make it available in the future. Calls to `FindWindow` will return the new widget's part code, so your application should be able to deal properly with part codes it doesn't understand.

Appearance offers three routines for collapsing and expanding windows: `CollapseWindow`, `CollapseAllWindows`, and `IsWindowCollapsed`. These routines work only on windows that actually support the new collapsing mechanism. How does Appearance know they support collapsing? As with controls, there's a new message to which windows can respond by returning a bitfield of features.

When a WDEF lets the Window Manager know that the window can be collapsed (through the feature bitfield), it can then be collapsed in the proper manner. The feature bit tells you that the window has a collapse box, which serves as an obvious widget that can be clicked. Appearance supports collapsing by double-clicking in the

---

title bar of a window as well, but that's not as obvious to the user, so it's important to display the collapse box.

There are no new messages to make a window calculate its regions in its collapsed state. When called with a `wCalcRgns` message, the `WDEF` should check to see whether it has been collapsed with a call to `IsWindowCollapsed` and calculate its regions based on whatever it considers its collapsed look to be. Normally this is the title bar alone.

## **SOMETHING NEW ON THE MENU**

Developers who use a custom `MDEF` may find that their menus look out of place when running with `Appearance`. To avoid this problem, `Appearance` extends the system `MDEF` to provide many of the features that developers have been using other `MDEFs` for. Now you can set the font of any menu item, making `WYSIWYG` font menus simple to create. You can also set extended modifiers, command IDs, text encodings, icons, and hierarchical menus, as described below. In addition to these new features, we reserved two long integers for your use, so that you can attach any other values you want to menu items. These are called, remarkably, `refCon` and `refCon2`.

You can specify the features you want for your menus either by calling the new routines mentioned below or by specifying the information in a new resource type ('`xmnu`'). When `GetMenu` is called, the Menu Manager looks for a resource of type '`xmnu`' with the same resource ID as the '`MENU`' resource. If the resource is found, it sets the extended information for each menu item of the menu for you. After the menu is created, you can adjust the values with the new routines.

### **EXTENDED MODIFIERS**

You can finally attach extended keyboard modifiers to menu items. So go ahead and add the `Command-Shift-Option-K` to your application that you've been dreaming about. You can specify the extended modifiers in the '`xmnu`' resource or by calling `SetMenuItemModifiers`. Use the `GetMenuItemModifiers` routine to get the current modifiers for an item.

You might wonder how those keyboard events will get processed correctly. After all, `MenuKey` doesn't have a modifiers parameter. The answer is `MenuEvent`, a new routine that takes a pointer to an event record. It returns a long integer as `MenuKey` and `MenuSelect` do, with the low word containing the item number of the chosen command and the high word containing the menu ID of the menu containing it. If nothing was chosen, 0 is returned.

### **COMMAND IDS**

To help out frameworks and other technologies like `OpenDoc`, you can assign a command ID to any menu item. This lets you forget about the position of menu items. Instead of having to track down which menu item corresponds to a given menu ID and item number, you can use the item's command ID to identify it. Listing 2 gives an example of using command IDs. Assuming you've set a different command ID for all your menu items, you can just call `GetMenuItemCommandID` to get the command ID and then perform the corresponding operation. In Listing 2 this is done via a switch statement.

### **TEXT ENCODINGS**

You can set the text encoding for a menu item. Think of text encodings as script codes. Previously you had to set the keyboard equivalent of a menu item to `$1C` and the icon ID of the item to the script code. The `$1C` would tip off the `MDEF` that

---

**Listing 2.** Using command IDs

```
menuID = HiWord(menuResult);
itemNo = LoWord(menuResult);
GetMenuItemCommandID(GetMHandle(menuID), itemNo, &command);

switch (command) {
    ...
    case kCmdQuit:
        PrepareToQuit();
        break;
}
```

there was really a script code in the icon field, not an icon ID. Now you can set the text encoding in the 'xmnu' resource or call the `SetMenuItemTextEncoding` routine. This enables you to have a text encoding and an icon simultaneously.

## ICONS

To set an icon for a menu item, you give it a handle to an 'ICON', 'SICN', or 'cicn' resource or an icon suite. The `SetMenuItemIconHandle` routine takes a parameter to determine the type of icon handle you're passing in, and a parameter for the icon handle itself. If you set an icon with this routine, it overrides any icon ID you may have set with `SetMenuItemIcon`. Using the new routine also allows you to plot 'SICN' resources and compressed (16-by-16) 'ICON' resources and still have a keyboard equivalent. Prior to Appearance, you needed to set the equivalent to \$1E and \$1D, for 'SICN' and 'ICON', respectively. Of course, you can also specify this information in the 'xmnu' resource.

## HIERARCHICAL MENU

Appearance offers an improved method for attaching submenus to menu items. Previously, the only way to do this was to set the keyboard equivalent of the menu item to \$1B and then set the mark character for the menu to the ID of the submenu you wanted to attach. Now you can use the 'xmnu' resource or the new `SetMenuItemHierarchicalID` routine to set the menu ID of a submenu. Both allow you to use a full 16-bit integer for your submenu ID. Freeing up the mark character makes it possible to have a checkmark next to a hierarchical menu.

## DRESSING UP WITH THE APPEARANCE MANAGER

The Appearance Manager is your one-stop shopping center for getting any colors and patterns needed to draw consistently with the current theme. In the first version of Appearance, the current theme is always Apple Grayscale, but it's important to start thinking about themes. With the APIs provided with the Appearance extension, you can get colors for such things as the active window header text or the inactive menu text. There are also several patterns you can get for dialog background patterns and the like.

Getting Appearance Manager colors and patterns makes it easier to create custom defprocs for UI elements that blend with the theme. If you're not using the Dialog Manager for certain windows, these routines can help you set your text color properly so that it matches the current theme.

Along with colors and patterns, there are also routines to draw Appearance primitives. Appearance primitives (as opposed to dialog primitives) are such things as

---

visual separators, group box lines, placards, and focus rings. The controls provided with Appearance call these routines to help their drawing. You might also call them to draw elements that match the current theme when you don't want to use a control.

In future versions of Appearance, there will be routines to do high-level things such as draw button backgrounds. Using such a routine, you could create a button with a specialized content type and be guaranteed that your button background would always draw correctly for the current theme.

## **ADOPTING APPEARANCE**

There are varying degrees to which you can prepare for Appearance. This section covers the basic preparations and then gives some specifics on using Appearance in your application. The sample code accompanying this article is a concrete example of an application that uses Appearance, so you can peruse the code after reading this section to get a feel for how Appearance features are used in an honest-to-goodness application.

### **NEVER ASSUME**

You should never assume things about the environment your application is running in. Always use routines like Gestalt and those in the Script Manager to get information about the environment. If there's no direct way to get the information, you probably shouldn't be doing whatever it is you're doing. Let's look at a couple of examples.

#### **Window metrics.**

Properly determining window metrics will help you position windows correctly, no matter what the window looks like. For example, in the past you may have let your application blindly assume that the window border is 1 pixel thick or the title bar is 19 pixels high. However, the structure region of a window is controlled by the WDEF, and with Appearance it's not guaranteed to be any particular number of pixels thick. Apple Grayscale borders are thicker than the old System 7 look, and when switchable themes are implemented, borders will vary by theme. Your application should be able to deal with this intelligently by getting the structure and content rectangles for the window and using them to calculate the width of the window border. Listing 3 shows how to do this properly. The sample application has a routine to size a window and set a window's bounds; the code also shows what makes the GetWindowRects routine tick, in case you're interested.

#### **Window variants.**

"Be careful what you ask for, you just might get it." Many applications ask for a documentProc window type and then never call DrawGrowIcon because the window isn't supposed to have a size box. Better to ask for the variant you really want — in this case, noGrowDocProc. As shown earlier, with Apple Grayscale the size box is part of the structure region. Because of this, the size box would be drawn automatically for a documentProc window type. We had to do some work to get around this for times when the user is running in System-wide Grayscale Appearance mode. Using the correct window variant is a step in the right direction. The Apple Grayscale WDEF has a set of variants that make it clear whether a window has a size box or not.

## **APPEARANCE SAVVYNES**

Now that you've got these assumptions out of the way, becoming Appearance-savvy is really not that difficult. Just do the following:

- Call RegisterAppearanceClient early in your application code, before you draw the menu bar or create any UI elements.

**Listing 3.** Using the structure region to position windows

```
void MoveWindowStruct(WindowPtr window, SInt16 horiz, SInt16 vert,
    Boolean update)
{
    Rect    structRect, contRect;
    Point   structTL, contTL;
    SInt32  diff;

    /* Get the structure and content rectangles in global coordinates. */
    GetWindowRects(window, &structRect, &contRect);

    /* Calculate the difference between the top left of the content */
    /* region and the top left of the structure region. */
    structTL = topLeft(structRect);
    contTL = topLeft(contRect);
    diff = DeltaPoint(contTL, structTL);

    /* Add the difference, since MoveWindow moves based on the */
    /* top-left corner of the content region. */
    horiz += *(Point*)&diff.h;
    vert += *(Point*)&diff.v;

    MoveWindow(window, horiz, vert, update);
}
```

- Use the new system-supplied windows, controls, and menus.
- Use the new 'dlgx' and 'alrx' resources to supplement your 'DLOG' and 'ALRT' resources.
- In dialogs, change any user items that are now available as controls (for example, a group box user item) into controls.
- Enable embedding and Appearance-savvy backgrounds in your dialogs.
- Make your alerts movable, and use the new StandardAlert routine whenever possible.
- Use the Appearance Manager to get any colors and patterns you need to draw consistently with the current theme.

The RegisterAppearanceClient routine tells Appearance that you want to map all calls to the classic defprocs (WDEF 0, for example) to the new defprocs automatically. The routines that translate from old to new form the compatibility layer, which is part of the mechanism that produces the Apple Grayscale look when “System-wide Grayscale Appearance” is checked in the Appearance Settings control panel. So calling RegisterAppearanceClient is an easy first step in adopting Appearance. To adopt Appearance completely and take advantage of some Appearance features, you have to call the new defprocs directly, as described in the following sections.

You can phase the above steps into your application at your own pace, adopting Appearance as your schedule permits. It’s not absolutely necessary to do everything at once. For example, you might have some dialogs where you can add the new resource 'dlgx' type, flip the right feature bits, and be fully Appearance-savvy without doing any more work. Those would obviously be the ones to convert first. Then you can go

on to other dialogs where you need to replace old user items with the new controls. When I converted the Date & Time control panel to use Appearance, I eliminated all but one of the user items (the menu bar preview) because Appearance provided controls to replace them (group boxes, icons, list boxes, and so on). So take your time, but remember that the sooner you adopt Appearance, the sooner your application will be ready for switchable themes when they're released.

When converting an application, be sure to run Appearance with System-wide Grayscale Appearance mode turned off. Turning off this mode puts your system back into the old System 7 look for applications that haven't adopted Appearance, which makes it easy for you to tell where you've implemented the new look and where you still have work to do. If you're running in System-wide Grayscale Appearance mode, you won't be able to distinguish the changes you've made from those performed automatically by the system.

### COMPATIBILITY LAYER VS. DIRECT ADOPTION

As you begin to adopt Appearance directly and rely less on the compatibility layer, you'll want to change the defproc IDs for windows, controls, and menus to the new IDs listed in Table 1. Even when calling the new defprocs directly, your application should always call RegisterAppearanceClient so that system elements such as the Apple menu will draw correctly. Read on about some differences you'll encounter when calling the new defprocs directly instead of going through the compatibility layer.

**Table 1.** Old to new defproc ID mapping for windows, controls, and menus

Old defproc ID	New defproc ID
documentProc	kWindowGrowDocumentProc
noGrowDocProc	kWindowDocumentProc
zoomDocProc	kWindowFullZoomGrowDocumentProc
zoomNoGrow	kWindowFullZoomDocumentProc
dBoxProc	kWindowModalDialogProc
movableDBoxProc	kWindowMovableModalDialogProc
plainDBox	kWindowPlainDialogProc
altDBoxProc	kWindowShadowDialogProc
floatProc	kWindowFloatProc
floatGrowProc	kWindowFloatGrowProc
floatZoomProc	kWindowFloatFullZoomProcID
floatZoomGrowProc	kWindowFloatFullZoomGrowProcID
floatSideProc	kWindowFloatSideProcID
floatSideGrowProc	kWindowFloatSideGrowProcID
floatSideZoomProc	kWindowFloatSideFullZoomProcID
floatSideZoomGrowProc	kWindowFloatSideFullZoomGrowProcID
pushButProc	kControlPushButtonProc
checkBoxProc	kControlCheckBoxProc
radioButProc	kControlRadioButtonProc
scrollBarProc	kControlScrollBarProc
popupMenuProc	kControlPopupMenuProc
textMenuProc	kMenuStdMenuProc

### Window variant codes.

The variant codes for the new WDEFs are different from the codes for the old WDEFs, so you need to change any place in your code where you rely on a window

variant. Variant codes are window-specific and don't provide a generic way to determine a window's features. To remedy that, we've provided a better way to find out about window features: `GetWindowFeatures`. This routine is the window version of `GetControlFeatures`; it returns a bitfield to help determine which features a window supports. Here are the bits that are currently defined:

```
enum {
    kWindowCanGrow          = (1 << 0),
    kWindowCanZoom         = (1 << 1),
    kWindowCanCollapse     = (1 << 2),
    kWindowIsModal         = (1 << 3),
    kWindowCanGetWindowRgn = (1 << 4),
    kWindowIsAlert         = (1 << 5),
    kWindowHasTitleBar     = (1 << 6),
};
```

If a window doesn't respond to `GetWindowFeatures`, it's probably an old-style window and you should just use its variant code. The `AppearanceHelpers` library that comes with the SDK includes a set of routines to make it easier to determine a window's features using the variant code (for example, `IsWindowModal`).

### Zooming variants.

What's not apparent from Table 1 is that there are actually two window definitions now, one for windows and one for dialogs. The utility window has also been split into two, one for the normal variety, the second for the side title-bar version. We did this to clean things up a bit and add new features. In this release, we've improved the zooming variants.

Notice that the new defproc ID constants say "FullZoom" and not just "Zoom." It's now possible to specify horizontal, vertical, and full zoom. As a result, the zoom box is drawn differently for each variant, as shown in Figure 7. The part codes for the zoom box and all that you've come to know and love about zooming are still the same. The new boxes are merely a way to visually state that the zoom will act in a particular way. For example, the Apple Guide floating window used for coaching a user collapses down into a compressed version of itself (just the buttons and topic are visible) when you click the zoom box. The horizontal zoom box would be perfect to indicate how the window will zoom when clicked.



**Figure 7.** Zoom box variants

### Size boxes.

Size boxes in standard utility windows are now 11-by-11 pixels. The standard scroll bars can be made to fit into this space without looking scrunched, as they've been changed to support this width as well as the standard 16-pixel width. When you use old defproc IDs and go through the compatibility layer instead of calling the new utility window defprocs directly, the size box is 16-by-16 pixels.

Although not recommended, it is possible for your size box to appear elsewhere than in its new location as part of the window frame. To do this, you would use a window variant that doesn't draw a size box and then put your size box where you want it, but you'd have to handle hit testing yourself. A future version of `Appearance` will have a

---

theme-savvy size box control that can be put anywhere inside a window; for now, you'll have to draw your own.

### **Weirdness banished.**

The new dialog WDEF metrics you get when adapting windows directly are slightly different from the old WDEF metrics. Modal and movable modal variants no longer have that weird 3-pixel portion of the structure region that looked like part of the content region. This means you can finally run your content up to the edge of the window. It also means you won't have any problems with a gray background and a white border, which would happen if you erased the background gray yourself and didn't use `SetWinColor` to set the content color. I'm sure many of you have had to deal with this situation before.

### **Enabling dialog and alert features.**

There's a big payoff to adapting dialogs and alerts directly — you get all those great new features described previously — but it involves a little more work than adapting a normal window. To enable the new features for a dialog, you either create the dialog with `NewFeaturesDialog` or, for resource-based dialogs, create a 'dlgx' resource with the same ID as the dialog's 'DLOG' resource. Both `NewFeaturesDialog` and the 'dlgx' resource allow you to specify some flags that tell the Dialog Manager what features a dialog supports. These are the bits:

```
enum {
    kDialogFlagsUseThemeBackground    = (1 << 0),
    kDialogFlagsUseControlHierarchy   = (1 << 1),
    kDialogFlagsHandleMovableModal    = (1 << 2),
    kDialogFlagsUseThemeControls      = (1 << 3)
};
```

The `kDialogFlagsUseThemeBackground` bit tells the Dialog Manager to make sure that the background of the dialog is painted in the right color or pattern for the current theme. The `kDialogFlagsUseControlHierarchy` bit tells the Dialog Manager to create a root control for the window and establish a control embedding hierarchy. The `kDialogFlagsHandleMovableModal` bit tells the system that if this dialog is frontmost when `ModalDialog` is called, and its window type is movable modal, it should handle the dialog as described earlier in “Movable Modal Dialogs and Alerts.” Don't forget to set the `kDialogFlagsUseThemeControls` bit, or the Dialog Manager will create old-fashioned System 7 controls on your nice grayscale dialog. Ick.

You can make alerts Appearance-savvy just by adding the new 'alrx' resource and setting the right bits. If you want to save some code, call the new `StandardAlert` routine to present your alerts. If your dialogs are Rez-based, it's really easy to create new resources. The `Appearance.r` file included with the SDK has the Rez definitions of the new resource templates.

### **Using embedding.**

If you turn on embedding in a dialog or alert, you may need to alter your code to deal with the fact that all items in the dialog or alert are considered controls in that mode. The sample application gives examples of how to code a dialog with an embedding hierarchy. A good example is the code that demonstrates `StandardAlert`. The sample code puts up a dialog in which you can set the parameters to a call to `StandardAlert`. Because everything is a control, you can easily enable and disable editable text fields and groups of items, for example. This results in the code for that sample dialog being very small, simple, and straightforward, especially compared to what it would have been if you had to do all those things yourself!

---

With Appearance's new controls, you should be able to eliminate most of the user items in your dialogs. This is a fairly straightforward process of changing the user items into control items. Remember that if your dialog has an embedding hierarchy, you should change your user items to (at the very least) user pane controls. The callback to draw is practically the same.

### **Menus on the fly.**

If you create menus on the fly and want to adapt them directly, you should use the `NewThemeMenu` call instead of `NewMenu`, because `NewMenu` assumes `MDEF 0`, which forces you through the compatibility layer. You'll find `NewThemeMenu` in the `AppearanceHelpers` library.

### **SO WHAT CAN YOU RELY ON?**

One of reasons for adopting Appearance is to prepare your application for switchable themes. In an environment where the entire look of the interface can change at any moment, it might seem that you have to be ready for anything. In some respects, that's true, but there are some things that you can rely on. These aren't assumptions, they're facts!

- Default rings and focus rings can be outset a maximum of 3 pixels.
- Editable text frames, group box frames, and list box frames can be a maximum of 2 pixels thick.
- Progress indicator borders can be outset a maximum of 2 pixels.
- Metrics of controls are the same across all themes, though borders, which are drawn outside a control's rectangles, can change for default and focus rings.
- Window structure metrics actually *do* change.
- The menu bar height can vary, but will never be more than 24 pixels.

This information is provided to help you lay out your UI elements with enough room to look good in all themes. When theme-switching is available, an Apple event will inform your application if a theme switch occurs. Then your application can adjust window positions to accommodate changes to window frames and menu bar heights.

### **NOW IT'S YOUR TURN**

I've tried to show you some of the highlights of Appearance, but there's only so much you can convey in words. To get a better feel for what it's all about, check out the sample application. It has a lot of useful, real-world examples of using Appearance, especially dealing with dialogs and embedding. For a deeper look at Appearance, try the Appearance SDK documentation, which has technical details beyond the scope of this article.

I think you're going to love using the new routines and features that are now available. You'll find you can get much more out of the Toolbox, which translates into less code that you have to write and less time required to implement your interface. But wait, there's more! Along with those benefits, you'll be ready for switchable themes as well. Now go check out that sample code and get cracking!

---

**Thanks** to our technical reviewers Sharon Everson, Arno Gourdol, Steve Ko, Tim Maroney, and Matt Mora. •