Appearance: More Than Skin Deep

Ed Voas

This is a preliminary draft of an article that will appear in Issue 30 of *develop, The Apple Technical Journal.*

Appearance: More Than Skin Deep

EDWARD VOAS

The Macintosh was once the model of consistency, where every application behaved and looked the same, making the user feel at home. But that consistency faded as, due to inadequacies in the Macintosh Toolbox, developers created custom controls, menus, and windows, moving forward with user interface innovations while the Mac OS stayed put. Now the Appearance SDK 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.

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 Software Development Kit (SDK) 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 Control, Dialog, and Menu Managers to provide functionality that's necessary for some of the new, more complex controls 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 commands.

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 that patchwork look — with the system elements, all pieces of the UI blend together nicely. Plus, as Apple 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 next). 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.

The Appearance SDK consists of the Appearance extension, a control panel, several header files, and a shared library to link against if your application runs under the Code Fragment Manager. The Appearance extension is intended to be bundled with applications and installed when running on systems prior to System 7.7 (System 7.7 contains the functionality of the SDK as part of the base system). Bundling the extension allows your application to run with Appearance on older systems so that you're not limited to deploying on just the latest system release. 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.

You'll find the Appearance SDK on this issue's CD. You'll also find, accompanying this article on the CD and on *develop*'s Web site, a sample application that demonstrates Appearance.

THE ON RAMP TO THEMES

You may have heard of themes at Apple's Worldwide Developers Conference (WWDC) or read about them in discussions on Mac OS 8. 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 allows you to switch themes on the fly. Figure 1 gives examples of three themes that were shown at the WWDC and elsewhere — Apple Grayscale, High Tech, and Gizmo.

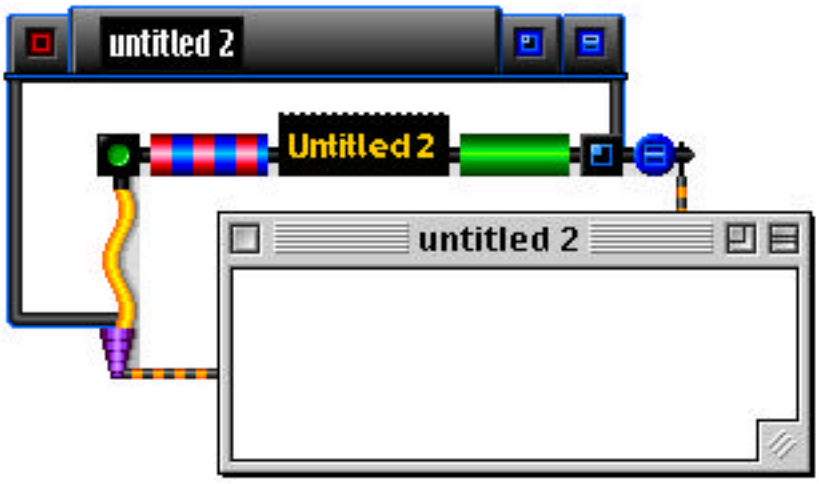


Figure 1
Three themes

<<put names in figure callouts: High Tech, Gizmo, and Apple Grayscale>>

Now, before you get too anxious, 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 Apple's future. Appearance is the first step toward that future, and using the system controls, windows, menus, and other UI elements provided by Appearance will allow your application to automatically handle theme switches 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. For now, think of Apple Grayscale as an actual theme.

WHAT'S NEW AND IMPROVED

This section gives you a quick run-through of the Appearance controls and windows, focusing on how Apple Grayscale looks and on the added features. It also introduces the Appearance Settings control panel, which lets the user control color and font, among other things. In later sections, we'll move on to describing 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. As an aid in describing the new controls and how they interact with dialogs, I've introduced some special terminology. Items in dialogs that are not controls (pictures, icons, editable text, static text, and user items) are called *dialog primitives*. Some of these primitives have control counterparts (for example, the counterpart of the editable text primitive is the editable text control). In referring to items in dialogs, the word *control* refers only to items that are *not* primitives.

Bevel button control

The bevel button control implements a standard square button. You can put an icon, picture, or text, singly or in combination, on the button. You can also attach a menu to this control. It sports multiple bevel thicknesses and several different button behaviors to suit just about any use. For example, you can specify momentary, toggle, or sticky behavior. With momentary behavior the button acts like a normal push button — after being clicked, it pops back up. With toggle behavior one click turns the button on, another turns it off. With sticky behavior, once clicked, the button stays down (on) — this is useful for tool palettes. The bevel button is by far the most complex new control. The sample application accompanying this article shows many different variations of bevel buttons. You'll be astounded by the possibilities.

Pop-up button control

The pop-up button control is the new version of the pop-up menu control. The main difference is that the control looks like a button. Also, the control behaves a bit differently: the menu opens below the button.

Slider control

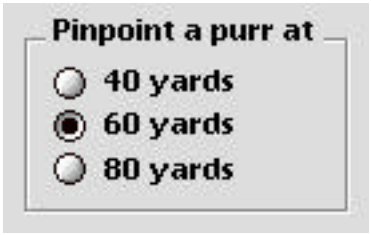
Sliders are finally part of the Mac OS repertoire. You can choose which way the indicator faces or use a nondirectional indicator. You can also specify whether to draw tick marks. Like the scroll bar, the slider supports live feedback.

Progress bar control

Progress bars are now very much a part of the standard control set. You can also tell a progress bar to switch into *indeterminate* mode to display an animated barber pole–

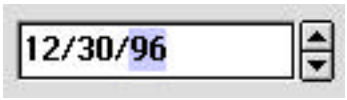
like bar. You might use the indeterminate mode to indicate that you haven't made a connection yet or are waiting for some piece of data before continuing. Because the indeterminate flag is separate from the value, you can switch back and forth without affecting the value.

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.

Clock control



The clock control provides an editable date or time field, as you'd find in the Date & Time control panel. 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.

Note that you won't be able to tab or type characters into the editable clock control unless the dialog has established an embedding hierarchy so that the control will be included in the dialog's keyboard focusing scheme (as described later). There's a similar restriction on the list box and editable text controls.

Little arrows control

The little arrows control implements the little up and down arrows you see tied to a box displaying a value. The arrows are used to increase or decrease the value. In the Date & Time control panel, for example, they help set the different parts of the date or time.

List box control

The list box control implements a simple list box. It requires an auxiliary resource of type 'ldes' to specify the features of the list, such as the number of columns and rows. This control allows keyboard filtering, and handles the default keyboard navigation you'd expect from a list box. (But remember, in order for a list box in a dialog to receive keyboard focus, the dialog must establish an embedding hierarchy.)

Tab control

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

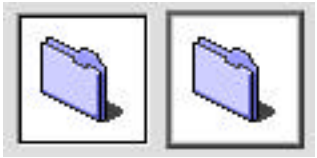
User pane control

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. Once grouped, you can hide, show, enable, or disable the user pane, and its embedded contents 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. You can also use them outside dialogs. 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 control 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.

Image well control



This control holds a graphic of some kind, such as 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 checked state, as shown above.

Window header control

The window header control is what the Finder uses to draw the header of the window where 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, perhaps to show information such as the current line number. You might have seen them in CodeWarrior. 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 right control.

Disclosure triangle control

The disclosure triangle control implements the tiny "turny triangles" that you've used in the Finder for some time now. They don't implement a hierarchical list, just the widget.

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.

Divider line control

The divider line control implements a simple visual separator.

CHANGES TO OLD CONTROLS

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

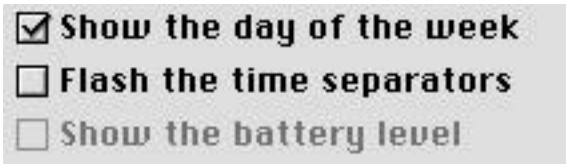
Button control

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 around the button. This flag also causes the control’s bounding rectangle to adjust itself correctly to accommodate the default ring.

Radio button control

Radio buttons have a new look.

Checkbox control



Our friend the checkbox sports a new look and real checkmarks!

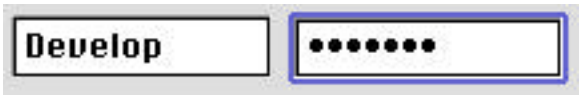
Scroll bar control

Scroll bars have a new look, too. One scroll bar variant supports live scrolling — you can set an action-procedure pointer to be called back as the indicator is moved. Each time your application is called back, the value of the control will indicate what position the user has dragged the indicator to.

Static text control

You use the static text control instead of the old static text items to embed static text in dialogs. Since they're controls, they can be deactivated and will then be drawn grayed out 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 items will become static text controls automatically.

Editable text control



The editable text control replaces the old editable text dialog item. Because it's a control, it can be disabled and enabled. 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 filtering and supports password entry. However, as with the clock control, these features are available only when 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 two text fields, one in password mode. The one in password mode has the keyboard focus, as indicated by the ring around it. (Keyboard focus is a new feature that's described later.)

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 <<*combine Figures 2a-2c*>>. Alerts are distinguished from dialogs by a red border; movable alerts also have a reddish title bar. Remember to use alerts only to warn the user of something or present important information; in all other cases, use a dialog. The many window variants are described later.

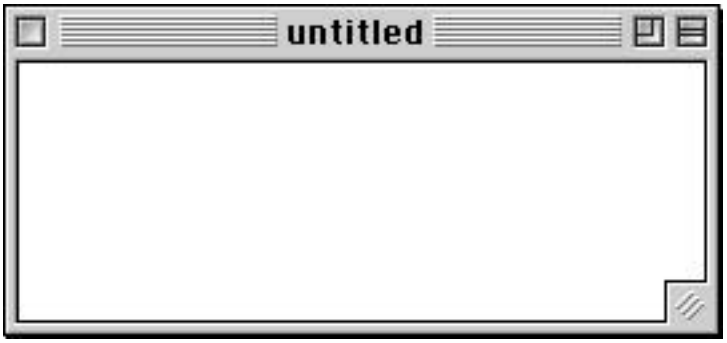


Figure 2a
Standard document window

Figure 2b
Standard modal dialog window

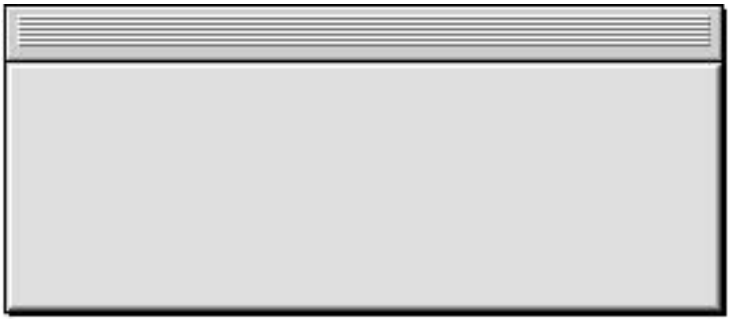


Figure 2c
Standard movable modal dialog window

THE APPEARANCE SETTINGS CONTROL PANEL

The new Appearance Settings control panel allows the user to control these settings:

- The accent color of the theme. This affects the coloring of menu items as they're selected, scroll bar and slider indicators, progress bar indicators, and focus rings.
- The system font. The Apple Grayscale system font in Truth, but users can choose the old Chicago font if they want.

One other important aspect of this control panel is that it allows the user to switch in and out of system-wide mode. When the System-Wide checkbox is checked, it means that every application gets the Apple Grayscale look; when it's unchecked, only applications that have explicitly adopted Appearance's features have the look. To reach the System-Wide checkbox, open the Appearance Settings control panel and click Options. Later in this article, you'll see the importance of this feature to developers in the process of adopting Appearance in their applications.

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
- provides a mechanism for controls to advertise the features that they support
- enables keyboard focus for controls
- makes it easy to get and set control data and choose fonts for control titles
- supports live scrolling for scroll bars

This section explains the changes and the rationale behind them.

DRAW ORDERING AND HIT TESTING

From the word *go*, control draw ordering has been backwards due to how the Control Manager handles 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 an order opposite to that 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.

Normally, this was never a problem because it was assumed that controls wouldn't overlap or be contained within each other. (It did make for some interesting drawing behavior on slower machines, though.) With new controls such as tabs and group boxes, this assumption is no longer valid. For example, consider the case where you want a tab control with three radio buttons and an editable text field. 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 3)! This happens because controls are drawn

first, and then dialog primitives. Needless to say, trying to manage the drawing order can be difficult.

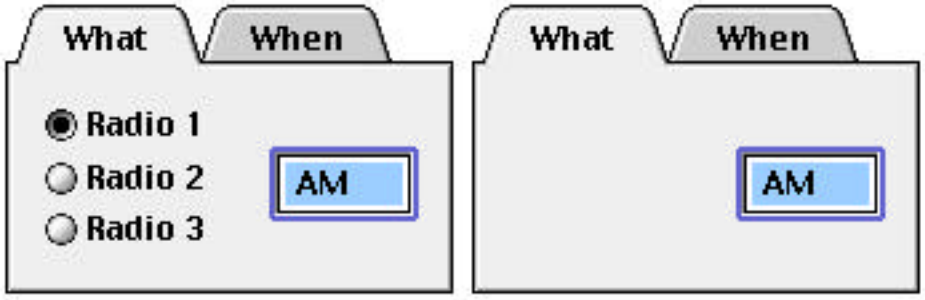


Figure 3
What you want vs. what you get

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. FindDialogItem also uses a linear search of the dialog items, which 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. Embedding makes this possible.

THE EMBEDDING HIERARCHY

To make drawing 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 enforces drawing order by making sure 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. This section describes a couple of key embedding concepts.

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 the appropriate dialog flag (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 3, the Dialog Manager would have embedded the radio buttons and editable text field into the tab control simply because they came after the tab in the 'DITL' resource and because they fit inside the tab control. The second reason is important: a control can be embedded automatically only in a control that already exists. So order still matters in your 'DITL' resource, 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. You can disable (or enable) all items in a window by disabling (or enabling) the root control of the window.

Doing things like switching tabs becomes remarkably simple. 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 is to be enabled (or shown) when its parent is enabled (or shown).” If you disable a control that’s latent, it becomes truly disabled and will not 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.

THE CONTROL FEATURES SET

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

FOCUS MANAGEMENT

With Appearance, you can get and set the *keyboard focus* of the 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. 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 SDK 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 type will support keyboard focus. 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 normally outset a maximum of three pixels from the control's bounding rectangle, except in the case of default buttons, where the focus ring overlays the default ring.

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 contrlData 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 implementation in future versions of the application. 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 is implemented via two new CDEF messages: 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, 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 that the data is passed back and forth as.

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 that you've created. To get a vital warp-engine intermix value from a control (a long integer, for those who don't know), you might call GetControlData like this:

```
SInt32   theValue;
OSErr    err;
Size     actualSize;

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

This data access mechanism is the cornerstone of many of the new features available with Appearance. It allows you to get and set control fonts, user-pane callback functions, bevel-button image information, and other useful things.

BETTER FONT CONTROL

Appearance allows you to set the font of any control, 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 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 Apple Grayscale, those fonts are Truth 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 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.

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 controls, this resource has been replaced by the new 'dftb' resource. When a 'dftb' resource is found for a dialog, any 'ictb' resource information is ignored. The 'dftb' resource has a straightforward resource format, making it easier to create and maintain than the old 'ictb' resource.

LIVE SCROLLING

Yes, you heard me right. Appearance allows you to have real live feedback with scroll bars and sliders. Some variants of the scroll bar and slider controls allow you to install a callback to be called whenever the user moves the mouse. Listing 1 shows a good example of how to install and use a live feedback callback.

Listing 1
Installing a live feedback callback 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)
       SetActionProc(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 above */
}
```

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

THE DIALOG MANAGER

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 have a theme-savvy background, that a root control be created for a dialog automatically when the dialog is created, or both. These features are requested through a bitfield in which each bit represents some feature. This flag field can be specified in two ways: through the new NewFeaturesDialog routine or in the new dialog and alert resource templates. In addition, the Dialog Manager provides an enhanced standard alert and standard movable modal behavior.

NEW RESOURCE TYPES FOR DIALOGS AND ALERTS

To make it simpler and more straightforward to add the new flags and also to clean things up a bit, we've created two new resource types for dialogs and alerts: 'dlog' and 'alrt'. They're pretty much the same as they used to be, but they now have the necessary slots to store the feature bits. For alerts, there's also a spot to specify a window title for movable alerts (described later).

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 flags that you would set in a 'dlog' or 'alrt' resource.

EMBEDDING REQUIRES CONTROLS

If you set the features flag for automatically creating a root control in a dialog, be aware that the new keyboard focus and embedding schemes require all dialog items to be controls in dialogs that establish embedding. This means, for example, that calling GetDialogItem on an editable text item will return a handle to a control, not a handle to text. Passing that handle, however, into GetDialogItemText still works as advertised and gives you the text. So sticking with the APIs is a good way to insulate yourself from the details. If you had assumed the handle was a handle to text, you would be unpleasantly surprised.

You might think that making dialog items controls creates problems, but it actually simplifies matters and 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.

It's important to remember that keyboard focus and embedding are in effect only when you specifically ask for them. You don't have to turn all dialog items into controls in existing dialogs if you don't want to use the new features.

You should also be aware that when embedding is turned on for a dialog, user items are always drawn last. This happens because user items are the only dialog primitive that Appearance doesn't automatically change into a control. The differences between user items and user panes were significant enough to prevent automatic conversion. So they end up drawing last, as described in the earlier discussions of draw ordering. This drawing behavior may work for you, but if not, it's time to start using user panes instead of user items and take advantage of all those neat user pane features mentioned previously.

STANDARD ALERTS

The new StandardAlert routine is the mother of all alert routines. It allows you to specify the text of the alert, and explanatory text if desired. You can display up to

three buttons with your choice of text, as well as a help button. 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 (Figure 4).

Figure 4
Basic alert with explanatory text generated by StandardAlert

MOVABLE MODALS

Appearance provides a standard movable modal behavior. Instead of having to write your own movable modal–handling code, you can now use the new movable-modal flag in the 'dlog' resource (or call NewFeatureDialog). The movable-modal flag tells ModalDialog to handle all the standard user interactions, such as dragging a dialog by its title bar or allowing the user to switch out of the application by clicking in another one. It’s up to you to use the right window type (kStdMovableModalDialogProc). Movable alerts (mentioned below) take advantage of this movable modal behavior.

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 application for handling; this allows you to handle suspend and resume events on switch, as well as any other events you might want to handle. You might want to continue to handle Apple events from other applications even though your application is in a modal state.

MOVABLE ALERTS

It’s now possible to make your alerts movable modal (Figure 5). All it takes is a quick flip of a bit (kAlertFlagsAlertIsMovable) in the 'alrt' resource. This gives you the same behavior.as setting the movable-modal flag in the 'dlog' resource.

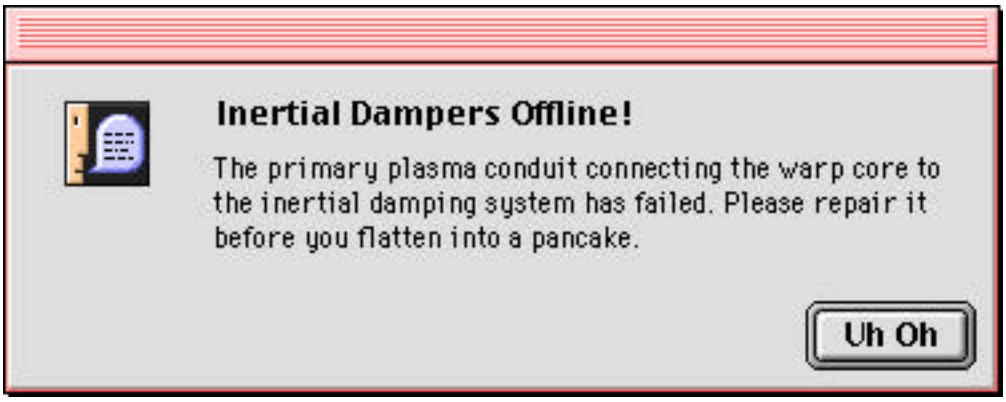


Figure 5
Standard movable alert

WINDOW MANAGER ADDITIONS

With Appearance running, the standard WDEF adds a new widget to the title bar, the collapse box, which is used to collapse and expand the window as WindowShade does in System 7.5. You can see the collapse box in the standard document window shown earlier in Figure 2. One advantage of the collapse/expand mechanism in Appearance is that windows actually remember that they've been collapsed (except after a restart), so when you can hide and show an application the collapsed state of all 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 this, but it will probably be added in the near 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. Currently the only feature supported is collapsing.

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 implies that the window has a collapse box, which serves as an obvious widget that can be clicked. Appearance supports double-clicking in the drag region of a window to collapse as well, but that's not easily discoverable, so it's important to display the collapse box.

There are no new messages to make a window calculate 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.

THE MENU MANAGER

Developers who use a non-system 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 also get extended modifiers, command IDs, text encoding, icon suites, and hierarchical IDs, which are described below. In addition to these new features, we reserved two long integers for your use, so that you can attach any other values to menu items that you wish. These are called, remarkably, refCon and refCon2.

EXTENDED MODIFIERS

You can now (finally!) attach extended modifier keys to menu items. Now you can go ahead and add the Command-Shift-Option-K to your application that you've been dreaming about. You use the GetMenuItemModifiers and SetMenuItemModifiers routines to get and set extended modifier information.

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 the 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 hof aving 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 shows what you can do whenever a menu item is chosen if you set the Quit menu item's command ID to be some specific value.

Listing 2
Using a command ID

```
menuID = HiWord(menuResult);
itemNo = LoWord(menuResult);

GetMenuItemCommandID(GetMHandle(menuID), itemNo, &command);
```

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

TEXT ENCODING

You can now set the text encoding for a menu item. Think of text encodings as script codes for now. Previously you had to set the keyboard equivalent of a menu item to $1C and set the icon ID of the item to the script code. The $1C would tip off the MDEF that it was really a script code in the icon field, not an icon ID. With the new routine, SetMenuItemTextEncoding, you can have a menu item with a special encoding *and* an icon.

ICON SUITES AND OTHER HANDLE-BASED ICONS

With Appearance, you can now also set the icon of a menu item to be a specific icon, giving 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 x 16) 'ICON' resources and still have a Command-key equivalent. Prior to Appearance, you needed to set the equivalent to $1E and $1D, for 'SICN' and 'ICON', respectively.

HIERARCHICAL IDS

The new SetMenuItemHierarchicalID routine lets you set the menu ID of a hierarchical menu to attach to a menu item. 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. The primary reason for the new routine is to allow you to use a full 16-bit integer for your submenu ID. It also allows you to have a checkmark next to a hierarchical menu.

THE APPEARANCE MANAGER

The Appearance Manager is your one-stop shopping center to get all those wacky colors and patterns needed to draw in the current theme. In the first version of Appearance, the current theme is Apple Grayscale, but it's important to start thinking about themes. With the APIs provided with the Appearance SDK, you can get colors for such things as the active window header text color or the inactive menu text color.

There are also several patterns you can get to make a window have the correct background color if you're simulating a dialog, for example.

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 use these routines to help their drawing. You might also use them to draw some elements in a theme-savvy manner 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 this stuff is 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 or perhaps 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 22 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 that accompanies the article has a routine to size a window and set a window's bounds. You'll also find what makes the GetWindowRects routine tick.

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);
}
```

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. With Apple Grayscale the size box is part of the structure region (look back at Figure 2). Because of this, the size box would be drawn automatically for a documentProc window type (you didn’t say “Hold the mayo”). We had to do some work to get around this for times when the user is running in system-wide 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.

APPEARANCE SAVVYNESS

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

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

These steps can be phased into your application at your own pace. It's not absolutely necessary to do everything at once. You can adopt Appearance as your schedule permits. For example, you might have some dialogs where you can convert to the new resource type, flip the 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.

Be sure to work while running Appearance with system-wide mode turned off. Turning off system-wide 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 mode, you won't be able to distinguish the changes you've made from those performed automatically by the system.

THE MENU BAR AND MENUS

To get the menu bar and menus to use the new theme-savvy defprocs, you must do a little work. First, for all menus that you've defined in resources, make sure to set the MDEF ID to 63. If you create menus on the fly, you should use the NewThemeMenu call instead of NewMenu, because NewMenu assumes MDEF 0.

```
menu = NewThemeMenu(kMyMenuID, title);
```

Now your menus are fine, but your menu bar needs a little work. Like NewMenu, GetNewMBar assumes MBDF 0, so use the new GetNewThemeMBar routine instead. If you're not getting your menu bar from a resource, but instead are creating

it on the fly, call InitProcMenu(63) before inserting any menus. The following code illustrates both methods:

```
menuBar = GetNewThemeMBar(kMyMenuBar);
SetMenuBar(menuBar);

InitProcMenu(63);
InsertMenu(GetMenu(kMyAppleMenu), 0);
InsertMenu(GetMenu(kMyFileMenu), 0);
```

WINDOWS

To get your windows to be theme-savvy, use the new constants (which are in AppearanceWindows.h) instead of the old ones. Table 1 shows how the old defproc IDs map to the new ones.

Table 1
Old to new window defproc ID mapping

Old defproc ID	New defproc ID
documentProc	kStdWindowGrowProc
noGrowDocProc	kStdWindowProc
zoomDocProc	kStdWindowFullZoomGrowProc
zoomNoGrow	kStdWindowFullZoomProc
dBoxProc	kStdModalDialogProc
movableDBoxProc	kStdMovableModalDialogProc
plainDBox	kStdPlainDialogProc
altDBoxProc	kStdShadowDialogProc
floatProc	kFloatProc
floatGrowProc	kFloatGrowProc
floatZoomProc	kFloatFullZoomProc
floatZoomGrowProc	kFloatFullZoomGrowProc
floatSideProc	kFloatSideProc
floatSideGrowProc	kFloatSideGrowProc
floatSideZoomProc	kFloatSideFullZoomProc
floatSideZoomGrowProc	kFloatSideFullZoomGrowProc

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 what 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),
   kWindowIsMovableModal   = (1 << 4)
};
```

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). We've also included a set of "mapper" WDEFs. These WDEFs map calls to WDEF 0 to the new WDEFs, so you can get away with zero code changes by using them. Your program thinks that WDEF 0 is giving you the new look. However, if you do this, you won't be able to use the extended features of the new WDEFs, such as horizontal and vertical zoom boxes, and you'll continue to have the problem with dialogs where a 3-pixel-thick area surrounds your content region (discussed later).

So, back to adapting your application for Appearance. For each window your application creates, you simply need to change the defproc ID to the new one. This is pretty straightforward. You should be able to just search and replace in your code to change any on-the-fly window creations. A quick bout with ResEdit should get the resource-based windows in line.

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." This is because 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 6. The part codes for the zoom box and all that you've come to know and love about zooming are still the same. It's merely a way to visually state that the zoom will act in a particular way. For example, the Apple Guide floating window used while coaching a user likes to collapse 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.

Full Horizontal Vertical

Figure 6
Zoom variants

DIALOGS AND ALERTS

Adapting dialogs and alerts involves a little more work than adapting a normal window. Dialogs require the new defproc numbers as well as the new 'dlog' resource format. The new resource allows you to set some flags that tell the Dialog Manager what features a dialog supports when creating the dialog. 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.” 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 using the new 'alrt' resource. If you want to save some code, use the new StandardAlert routine to present your alerts.

The new 'dlog' and 'alrt' templates have extra room for specifying the flags needed to use theme backgrounds and an embedding hierarchy. The 'alrt' template also lets you specify a title for your movable alerts. Speaking of which, you should try to make as many of your alerts movable as possible, since they aren’t system modal like normal alerts. This allows the user to potentially correct the problem you’re alerting them to before they click OK (or “Try Again”).

Using embedding.

If you turn on embedding in a dialog or alert, you must 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 accompanying this article gives examples of how to code a dialog with an embedding hierarchy. A good example is the code that demonstrates

StandardAlert. It puts up a dialog in which you can change the parameters of the alert. 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 dialog being very small, simple, and straightforward, especially compared to what it would have been if you had to do all of that stuff 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.

Weirdness banished.

One last important issue: the new dialog WDEF metrics 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 happen if you erase the background gray yourself and don't use SetWinColor to set the content color. I'm sure many of you have had to deal with this situation before, especially if you wanted gray dialog backgrounds.

CONTROLS

Making controls Appearance-savvy is easy. Simply use the new defproc IDs listed in AppearanceControls.h. Table 2 shows the mapping from the old numbering. Change any instances of the old defproc numbering to use the new constants in your code. Change your resources to use the new IDs as well.

Table 2
Old to new control defproc ID mapping

Old defproc ID	New defproc ID
pushButProc	kStdPushButtonProc
checkBoxProc	kStdCheckBoxProc
radioButProc	kStdRadioButtonProc
scrollBarProc	kStdScrollBarProc
popupMenuProc	kStdPopupButtonProc

NOW IT'S YOUR TURN

I've tried to show you what's so great about 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 when dealing with dialogs and embedding.

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 get a theme-savvy interface as well. Now go check out that sample code and get cracking!

Thanks
to our technical reviewers C. K. Haun, Steve Ko, Tim Maroney, Matt Mora and Arno Gourdol.

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?").