# Appearance 1.0 Technical Documentation

APPLE CONFIDENTIAL

## Overview

This document describes the toolbox enhancements for Mac OS 8.

## Goals

The main goals of these enhancements is to lay the foundation for switchable theme, try to bring back a consistent interface, and make it much easier to write programs for the Mac OS. We will accomplish these goals by doing the following:

- provide many new control types previously unavailable on the MacOS, such as sliders, tabs, and group boxes.
- allow applications to adopt these new controls so they will be theme savvy automatically when theme switching is available.
- provide enough functionality to make it no longer necessary for developers to create their own defprocs, etc. This allows us to avoid a patchwork appearance when running under themes.
- provide a richer environment for controls to allow multicolored backgrounds, embedding, and correct drawing order and hit testing.

## Deployment

**Extension**

Appearance 1.0 is delivered as a system extension.

**Appearance APIs**

APIs mentioned in this document are delivered as classic 68K trap-based routines, CFM-68K routines, and CFM-PPC routines.

## System-Wide Appearance

Appearance is by default system-wide. This means that all applications that are running automatically get the grayscale look. The new defprocs introduced with Appearance have different resource IDs (and hence proc IDs) than the classic System 7 controls. To cause applications to use the new Appearance defprocs, we implement a set of 'mapper' CDEFs. When an application asks for WDEF 0, it gets our mapper WDEF instead.

Some defprocs have a compatibility mode within them that are activated when called thru a mapper. Any special compatibility behavior is mentioned with each defproc description below.

## Compatibility Mode

For compatibility reasons, it is possible to turn off the system-wide aspect of Appearance in the Appearance control panel. This has the effect of putting the system back into the classic System 7 look. A restart is required for this change. When in this mode, the mappers simply call thru to the classic defprocs in the system file, causing any request for WDEF 0 to actually get the classic WDEF 0, as expected.

Applications that adopt the new Appearance defprocs directly and call a new routine (RegisterAppearanceClient) will continue to have a grayscale look when system-wide appearance is off. The mappers sense clients and call thru to the new defprocs in this case. If an appearance client adopts the Appearance defprocs directly by using the new defproc IDs, this will bypass the mapper defprocs, eliminating the overhead involved in mapping the calls to the right defprocs.

## System Font Replacement

The user can change the system font from Chicago to Charcoal and back in the Appearance control panel (the default is Charcoal). Some applications may not get along perfectly with Charcoal (we have found only 1 to date), so this option is made available to users so they can adjust it to suit their needs or personal preference.

## Control Manager Extensions

#### Control Feature Flags

The Control Manager defines bits which represents the feature set of a specific control. The features possible are listed below:

```
enum
{
  kControlSupportsGhosting        = 1 << 0,
  kControlSupportsEmbedding       = 1 << 1,
  kControlSupportsFocus           = 1 << 2,
  kControlWantsIdle               = 1 << 3,
  kControlWantsActivate           = 1 << 4,
  kControlHandlesTracking         = 1 << 5,
  kControlSupportsDataAccess      = 1 << 6,
  kControlHasSpecialBackground    = 1 << 7,
  kControlGetsFocusOnClick        = 1 << 8,
  kControlSupportsCalcBestRect    = 1 << 9,
  kControlSupportsLiveFeedback    = 1 << 10
};
```

To obtain a control's features, the GetControlFeatures routine is available.

#### New Control Messages

To provide for the extended functionality of controls, the following messages have been added:

```
enum
{
  kControlMsgDrawGhost            = 13,
  kControlMsgCalcBestRect         = 14,
  kControlMsgHandleTracking       = 15,
  kControlMsgFocus                = 16,
  kControlMsgKeyDown              = 17,
  kControlMsgIdle                 = 18,
  kControlMsgGetFeatures          = 19,
  kControlMsgSetData              = 20,
  kControlMsgGetData              = 21,
  kControlMsgActivate             = 22,
  kControlMsgSetUpBackground      = 23,
  kControlMsgCalcValueFromPos     = 24,
  kControlMsgTestNewMsgSupport    = 25
};
```

To send these messages to a control, a new API has been added: SendControlMessage.

**Supporting the New Messages**

In order to declare that you support the new features/messages, a CDEF should respond to the kControlMsgTestNewMsgSupport by returning the constant kControlSupportsNewMessages as the result code of the CDEF. If a CDEF does not respond to this message, it is assume to not know anything about the new messages.

**Control Features**

As a prerequisite to most of the following, a control which wishes to support the new features we'll be getting into should support the kControlMsgGetFeatures message. A CDEF should return as its result a bitfield comprised of the bits representing the features you support. These bits are the simple OR-ing of the constants shown above. Here is an example:

```
... in your main CDEF function someplace...

  case kControlMsgGetFeatures:
      result = kControlSupportsDataAccess | kControlWantsI-
dle;
      break;
```

This control would both support data access and would like to receive idle events. Both are explained below.

**Tagged Control Data**

There is a definite need to have read and write access to different attributes of a control. In most cases, these attributes are unique to a particular control. To facilitate accessing this data without exposing the implementation of a CDEF, there are a series of routines to allow you to get and set particular pieces of information in a CDEF. These routines are listed in the Control Manager Reference section below.

To advertise that a CDEF supports data access, it should return kControlSupportsDataAccess as one of its feature bits in response to the kControlMsgGetFeatures message. If you then call the new GetControlData or SetControlData routines, it will then be called with the kControlMsgSetData and kControlMsgGetData messages with the 'param' parameter holding a pointer to the following structure:

```
struct DataAccessRec
{
  ResType             tag;
  ControlPartCode     part;
  Size                size;
  Ptr                 dataPtr;
};
```

The tag field indicates the name of the piece of data we want, for example, we might want the transform of a bevel button's image. The part field indicates what part of the control this applies to, it is usually 0, meaning the entire control. For controls like tabs, it might refer to a specific tab. The size and dataPtr generally specify a buffer and how long it is. These two fields are used specially during the GetData messages. If the dataPtr is nil, the information should not be copied in (obviously, I hope). This tells the GetData handler that we are merely interested in the size of the data. In *all* calls to GetData, the CDEF should fill in the actual size of the data in the size field before returning.

It is the responsibility of the CDEF writer to return errDataNotSupported if the tag is unknown or invalid (perhaps you don't want people to set a particular value). The CDEF returns the error code as the function result of the CDEF itself.

Here's an example of a CDEF responding to a GetData message. Let's assume that the piece of data in question in a short integer:

```
... someplace in your CDEF main...

  case kControlMsgGetData:
      {
          OSErr       err;
          DataAccessPtrptr;

          ptr = (DataAccessPtr)param;
          if ( ptr->tag == kMyWizzyDataTag )
          {
              if ( ptr->dataPtr )
                  *(SInt16*)info->dataPtr =           Get-
MyWizzyData();

              ptr->size = sizeof( SInt16 );
              result = noErr;
          }
          else
              result = errDataNotSupported;
      }
      break;
```

```
. .. more stuff here ...

  /* return the result */
  return result;
}
```

Notice how we return errDataNotSupported for an invalid tag. We also fill in the size regardless of whether dataPtr is nil or not. This is the proper way to handle this message.

**Indicator Ghosting**

Scroll Bars and Sliders, while tracking the indicator, drag a ghost image of the indicator around instead of the old dotted outline. To accomplish this, these controls return the feature flag kControlSupportsGhosting in response to a kControlMsgGetFeatures message. When TrackControl is called to track the indicator, it checks this feature flag and if set, calls the control with the message kControlMsgDrawGhost, with the param parameter set to a region handle indicating where the ghost indicator should be drawn. This region is a copy of the indicator region offset by some amount, depending on where the user dragged it to.

**Live Feedback**

Live feedback is the more generic term for live scrolling. Scroll bars and sliders support live feedback thru different variants. When the right variant is chosen, these controls return the kControlSupportsLiveFeedback bit as part of their feature bit set in response to the kControlMsgGetFeatures message.

When TrackControl is called, it checks to see if this feature is supported, and that there is an actionProc installed (via SetControlAction). If so, it tracks the indicator, calling the CDEF with a kControlMsgCalcValueFromPos message whenever the user moves the mouse. The 'param' parameter contains a handle to the indicator region being dragged. The CDEF should respond by recalculating its value based on the new position of the region passed in. Once recalculated, the CDEF should redraw itself, making sure it draws the indicator in the position the region passed in represents. The region should NOT be changed. It is very important to draw exactly where the indicator is currently located, otherwise the feedback will behave improperly. Drawing where the region is also makes for a very smooth scrolling experience. When the user let's go of the mouse button, you will be asked to draw again. At this time, you can recalculate your correct position and redraw.

**Calc Best Rectangle**

In order to make group boxes work right, it was necessary to add the ability to ask check boxes and popups to calculate the best size so that they could be placed properly at the top of the group box. A control advertises that they support the kControlMsgCalcBestRect message by setting the right bit in their feature flags (kControlSupportsCalcBest). When called with this message, a CDEF is passed the address of a rectangle in its 'param' parameter. A control should calculate its best width and height and adjust the rectangle accordingly. It should merely set the bottom and right fields of the rectangle to the appropriate values. It should also return the baseline for where the text should line up based off the bottom of the rectangle (normally negative) as the CDEF function result.

With this message, a control's rectangle can automatically be sized to just fit the check box icon and the text, for example. Currently push buttons, check boxes and popup buttons are the only controls to support this message. In fact the StandardAlert routine (mentioned later) uses this to help autosize the push buttons in the alert.

**Handle Tracking**

Sometimes it is desirable to not have the default tracking behavior that TrackControl provides. In particular, if a control needs to do special tracking, such as the bevel buttons need when displaying a menu, the only way to do this is to hook into autoTrack. In this case, TrackControl will always return the part code that was initially hit, even if the user tracked off the menu. This is often undesirable. Also, controls like Bevel Buttons have toggling and sticky behavior, where they actually modify their own values after tracking.

To allow for this special behavior, a control can perform all aspects of tracking by advertising it wishes to do so by returning kControlHandlesTracking in the attributes returned via a GetControlFeatures call. With this bit set, the control is called when TrackControl or the new routine, HandleControlClick, is called. (HandleControlClick is virtually identical to TrackControl, only it allows modifier keys to be passed in.)

A structure of type ControlTrackingRec is passed in param when the CDEF is called with the kControlMsgHandleTracking message. The structure looks like this:

```
struct ControlTrackingRec
{
  Point           startPt;
  SInt16          modifiers;
  ControlActionUPPaction;
};
```

The action parameter should be called during tracking. The value of action can be a valid procPtr, nil, or -1. -1 indicates the control should do what it wants to if it actually has some special autoTrack behavior it wants to add. Most of the time, -1 would probably be treated like nil, meaning do nothing.

When the CDEF is done tracking, it should return the part code that was hit, or kControlNoPart if the user tracked off, etc. as the result code of the CDEF.

**Focus**

To accommodate the needs of focusing onto a control for keyboard input, there is a new keyboard focus messaging mechanism. A control tells the Control Manager it wants to receive keyboard input by returning kControlSupportsFocus as part of its feature bit set. When the control needs to be focused (which is determined by the Control Manager or some other outside influence), the CDEF is called with a kControlMsgFocus message, with param being the part code to focus. There are some special part codes that can be passed in param:

```
enum
{
  kFocusNoPart= 0,
  kFocusNextPart= -1,
  kFocusPrevPart= -2
};
typedef SInt16FocusPart;
```

The kFocusNoPart part code indicates the control should lose its focus. It might respond by deactivating its text edit handle and erasing its focus ring.

The kFocusNextPart and kFocusPrevPart part codes indicate that the CDEF should advance or reverse the focus to the next/previous sub-part. A date/time CDEF might advance to the day or year part, for example.

Alternatively, the CDEF might be asked to focus a specific part. It is up to the CDEF to decide how to behave in this case. Most controls only have one part and simply focus themselves.

In response to a focus message, the CDEF should return the part code that actually was focused. If it is out of parts, i.e. it has run off the end or beginning of its subparts, it should return kFocusNoPart. It should also return kFocusNoPart if called with kFocusNoPart. This tells the focusing mechanism to jump to the next control that supports focus.

Some controls actually want the focus when clicked, while others do not. To make sure a CDEF gets the focus with a click, it needs to set the kControlGetsFocusOnClick feature bit in response to a kControlMsgGetFeatures message. With that bit set, if the control is clicked on, whatever part is returned by a call to TestControl should be passed in as the part code for focusing.

**Idle Processing**

A CDEF can specify that it wants to get idle time by OR-ing the constant kControlWantsIdle into its feature bits. When this is set, it is called with the kControlMsgIdle message whenever someone calls IdleControls on the window the control is in. The param parameter is undefined. The chasing arrows and indeterminate progress indicator controls use idle time to do their animation.

#### Embedding

If a control is an embedder, i.e. it is designed to have other controls and widgets within its contents, it should set the kControlSupportsEmbedding flag in response to a GetFeatures call. This lets the Control Manager know to treat the control differently. See the section on Control Embedding below. When a CDEF is an embedder that has a background, such as a window header, it should also support the background message, mentioned below.

#### Activate Events

It is often desirable for a CDEF to know that it is becoming deactivated at a high level. The only way for a CDEF to determine such a change in state in the past was to check the current hilite state against a previously saved hilite state on each call to draw. The kControlMsgActivate message eliminates the need to do this. When a control is going to be deactivated, its feature set is checked to see if its kControlWantsActivate bit is set. If so, it is called with the kControlMsgActivate message with the value of param being either 1 or 0, with 1 indicating the control is becoming active. The control can do any special processing it needs, such as deactivating its TEHandle or ListHandle.

#### Background Color

Some controls that embed other controls sometimes have their own fill color. This may or may not be different than the current window background color. We need to make sure that any controls that are drawn on top of it can erase to the correct color using EraseRect or EraseRgn. It is very important that these two calls work, as a control might call toolbox routines such as TETextBox, which internally call EraseRect.

To make sure the background is always correct when drawing a control, before drawing it, the Control Manager works its way backwards from the control to be drawn, checking to see whether any control behind it has its kControlHasSpecialBackground feature bit is set. If it is, the control is asked to set up its background color and/or pattern. The Control Manager saves and restores the graphics state before and after the drawing. This way the CDEF can draw as it always has, using standard routines. The CDEF should never assume, however, that the background is a flat color and not a pattern, so it is wise to call BackPat with a white pattern before erasing to a specific color.

#### Special Font Styles

It is now possible, thru the new data access support, to set the text style of a control. All controls that display text have been written to support this feature. This allows for easier handling of control fonts by not forcing frameworks etc. to have to use the window font variant and constantly muck with the font of the window to make sure everything draws right. Since so many controls support this piece of tagged data, there is an API to actually set this information: SetControlFontStyle.

There is also a new resource type ('dftb') which is automatically read in by the Dialog Manager to facilitate a data-driven approach to setting font information for all controls in a dialog. This resource simply consists of an array of ControlFontStyleRecs. This resource is meant to replace the ictb, since the old control color table information is ignored under Appearance. Also, ictbs don't allow font specification for controls, just edit and static text. When this new resource is read in, the control font styles are set, and the resource is then purged.

## Control Embedding

#### Overview

We have introduced the concept of a containment hierarchy to the Control Manager to help impose drawing and hit testing order based on visual containment. Standard control drawing order is the order of the controls in the control list of a window, which is backward from the order that items are added to a window. This is due to the fact that the Control Manager adds controls at the head of the list, creating a push-down stack of controls.

A hierarchy is a very useful method of making sure embedder controls draw before their embedded content. It also is helpful in doing an "inside-out" hit testing function to determine the most deeply nested control that is hit by the mouse. Other advantages to this hierarchy are helping to correctly setting up a control's background color, as mentioned in the Background Color section above (we can easily know what's 'behind' something), and helping with keyboard focus.

#### Root Control

To enable embedding in a window, a window needs to have a root control created for it. This control merely serves as the top level of the containment hierarchy for its window. No embedding can take place until a call to CreateRootControl is successfully made. Once created, the root can be retrieved by calling GetRootControl. Once a root is created, all controls created after that are automatically added into the root. If any controls exist prior to calling CreateRootControl, an error is returned and the root is not created. The root control is implemented as a User Pane, one of the new CDEFs added with Appearance.

#### Embedding

Embedding of controls is accomplished by two routines, EmbedControl and AutoEmbedControl. EmbedControl tells the Control Manager to embed one control specifically into another. AutoEmbedControl tells the Control Manager to find the most likely container for the control based on where it is compared to what else is in the window. If a control is visually within a group box, for example, it will be embedded in that group box automatically with AutoEmbedControl, provided the group box already exists (see the DITL Ordering section, below).

New routines are available to correctly deal with the hierarchy, and classic routines have been changed to support this new construct. Consult the API section at the end of this document for specifics.

#### DITL Ordering

The DITL ordering plays multiple roles with an embedding hierarchy. First, it helps determine what gets embedded in what. As items are added to a dialog during dialog creation, controls that exist in the window (because they've already been created), can be valid target containers for any new controls that are created, provided they support embedding. It is therefore important to control the order that things are placed in the DITL. The large, embedder controls should be at the beginning. Smaller ones should follow. So you'd add your tab control first, and then follow it with some radio buttons, etc. Because the tab control would be created and already in place, the radio buttons can then be autoembedded with the tab control, as long as they were actually contained within it visually.

DITL ordering also affects focus ordering. The default focus order is the order things are added into the hierarchy. Future versions of the toolbox will most likely support other, more visceral methods of focusing.

#### Latency

To properly handled embedded content, it is necessary to have a certain *latent* state when dealing with the enabled state and visibility of a control. For example, consider a control within a group box. Disabling the group box disables the control as well. When the group box is reenabled, the controls within should reenable if they were enabled originally. If an embedded controls was disabled, it should remain disabled.

To handle this we have introduced the concept of latency. When disabling an embedder, any embedded content which is enabled becomes latent, or 'pending enabled'. This lets the Control Manager know to reenable them when the embedder becomes active again. This same concept applies to visibility as well. Clients should never need to know whether something is latent or not - every thing will just seem to work.

## CONTROL DEFINITIONS

#### Check Boxes, Radio Buttons and Push Buttons

These controls have been modified to handle drawing with the new appearance.

There is a mechanism to specify that a push button get the default appearance, i.e. it is drawn with a default ring around it. This is accomplished thru the data access mechanism. The default ring is drawn outside the control rectangle.

#### Bevel Buttons

Bevel buttons are the most complex new control type. There are a multitude of states, along with three different behaviors. On top of this are three different bevel size choices, and the ability to display an icon, text, or a picture. It is also possible to have the button display a combination of text and a graphic.

Bevel buttons allow the caller to control the content type (pict/icon/etc.), the behavior (push-button/toggle/sticky), and the bevel size. The caller also has the option of attaching a menu. When a menu is present, the caller can specify which way the popup arrow is facing (down or right).

This is all made possible by overloading the Min, Max, and Value parameters for the control, as well as adjusting the variant code. A similar approach is used in the current popup menu control.

| Parameter | Meaning |
|---|---|
| Min | Hi byte = Behavior; Lo byte = content type. |
| Max | ResID for resource-based content types. |
| Value | MenuID to attach; 0 = no menu, please. |

The variant code is broken down into two halves. The low 2 bits control the bevel type. Bit 2 controls the popup arrow direction (if a menu is present) and bit 3 controls whether or not to use the control's owning window's font.

The three behaviors of bevel buttons are push button, toggle, and sticky. The push button behavior makes bevel buttons pop back up after clicking them, just like the normal push button control. The toggle behavior allows the buttons to toggle state automatically when clicked (from on to off). Sticky buttons never pop up after clicking them. They stay down permanently, until the client calls SetControlValue(0) on them. These are useful in tool palettes. All of these behaviors are handled by the CDEF by setting itself up for self tracking when initialized. The high byte of the Min parameter contains the behavior of the button.

It is also possible to mark a button as having multi-valued menus. This means that the button does not maintain the menu value as it normally would (i.e. only one item can be selected at a time). This essentially allows a user to toggle entries in a menu and have multiple items checked. In this mode, the GetBevelButtonMenuValue routine returns the value of the menu item last selected.

One last behavior is to offset the contents while pressed. Some people believe it gives it a more realistic button feel.

The four types of data that can be displayed in a bevel button are icon, picture, text, and CIcon. The IDs for the icon/pict resource are passed in the Max parameter. The content type is passed in the low byte of the Min parameter. The variant code kControlUsesOwningWindowsFontVariant applies when text content is used.

An example call:

```
control = NewControl( window, &bounds, "\p", true, 0,
                  kContentIconSuiteRes + kBehaviorToggles,
                  myIconSuiteID, bevelButtonSmallBevelProc, 0L );
```

Attaching a menu:

```
control = NewControl( window, &bounds, "\p", true, kMyMenuID,
              kContentIconSuiteRes, myIconSuiteID,
              bevelButtonSmallBevelProc + kBevelButtonMenuOnRight,
              0L );
```

This will attach the menu with ID kMyMenuID to the button, with the popup arrow facing right. This also puts the menu to the right of the button.

Bevel buttons with menus actually have two values: the value of the button (on/off), and the value of the menu. The menu value can be extracted with the routine GetBevelButtonMenuValue.

One can mix graphics and text by selecting a graphical content type while providing a control title.

It is possible to align and place graphic and text content in special ways. For example, text buttons can have their text aligned to the left, right, centered, or use the current script direction. Graphic contents can be aligned likewise, but can also be aligned to the top, bottom, left, right, and all four corners of the button. With each of these alignment options, you can specify an offset from the particular side you are aligning the element to. For example, you can specify that the graphic be aligned to the top of the button, but allow 4 pixels of space.

Text placement can be specified as well if you are combining text and a graphic. You can specify whether the text should go above, below, to the left, or to the right of the graphic. This can be combined with the graphic alignment property to create a button where the graphic and text is left justified with the text below the graphic. You can also use a script direction placement in combination with a script direction graphic alignment. This means it is possible to have a graphic on the left with the text to the right in left-to-right systems, and the graphic on the right with the text to the left of it on right-to-left systems. All of this is automatic.

All bevel button private data is hidden. Accessor routines get and set values.

The caller can create its own control and then set the content to an existing handle to an icon suite, etc., using the accessors. Resource-based content is owned by the control, while handle-based content is owned by the caller. The CDEF will not try to dispose of handle-based content.

The bevel button can return 3 possible part codes: kControlNoPart, kControlButtonPart, and kControlMenuPart. The most complex case is when a Menu is attached. If the user selects a menu item, the part code kControlMenuPart is returned. If the user tracks out of the menu, but is still over the button when the mouse is released, the kControlButtonPart code is returned. If the user tracks outside of the button and the menu, kControlNoPart is returned. The button always returns kControlNoPart when it is disabled, as is expected.

#### Chasing Arrows

Chasing arrows are a small CDEF. Animation is handled on idle, which this CDEF sets itself up to be called with an idle message (when IdleControls is called) by OR-ing in the kControlWantsIdle bit into its feature flags.

This control's min, max, and value parameters are reserved.

**Clock**

This CDEF implements either an editable or non-editable time/date field, such as can be found in the Date & Time control panel. It is focusable and keyboard aware. The little arrows it uses to allow manipulating a particular portion of the date or time are actually coded as part of this control, i.e. it does not use the actual Little Arrows control. This is to make sure the little arrows will never get the focus on their own when in this type of control in the future when generic focusing is introduced all around.

The clock also has a 'live' variant. With this variant, the clock actually ticks on idle. You can use the non-editable version to place a live clock in a window, etc. If you combine the live variant with the editable variant, you end up with a clock that will actually affect the system clock. This is what the Date & Time control panel uses.

This control's min, max, and value parameters are reserved.

**Disclosure Triangle**

This is a fairly straightforward CDEF with two possible values, 0 and 1 for collapsed and expanded, respectively. There is a variant code bit to select between right- and left-facing versions. There is also a variant which allows autotracking to take the burden off application programmers. This control maintains its last value, so it knows what transition is taking place when a SetControlValue is called on it (expanded to collapsed, or vice versa). A function is available to set the last value of the control to make sure animation is set up properly.

**Editable Text Control**

A CDEF implements editable text complete with theme-savvy border and focus rings.

This control advertises that it should be included in the Dialog Manager focus chain by setting a flag in the control's feature flags: kControlSupportsFocus. It has two variants: the normal variant is used in a window (non-dialog) situation, and in this state it maintains its own TE-Handle. The second variant is used in dialogs, so that it shares the dialog's common text handle, just like the edit text dialog primitive does. This is to provide maximum compatibility, and to make sure that routines like DlgCut, etc. still work, since they are implemented as glue routines in MacOS.lib. They assume the text edit handle in the dialog record is valid and up-to-date.

This control can also have a key filter attached to it to handle filtered input. The filter is attached via the Data Access routines.

There is a password variant of this control which is script manager-savvy. The clear text of the password can be gotten thru GetControlData. The tag is kEditTextPasswordTag.

This control's min, max, and value parameters are reserved.

#### Group Box

The primary group box is implemented as a CDEF with variants for no header, check box header, text header, and popup header. The part code returned from TestControl or TrackControl depends on what type of header is in use. If the header is text, this always returns kControlNoPart. If it is a check box, it will return kControlButtonPart if the check box was hit. If it is a popup menu, it will return kControlButtonPart if the mouse was released over the button and kControlMenuPart if an item in the menu was selected. If the user tracked completely out of the control, kControlNoPart is returned.

Secondary group boxes are a variant with all the same options and a slightly different group box line look.

It is up to the caller to perform any pane-switching when using a popup title variant and the value of the popup changes. Likewise, the caller must enable/disable contents if using a check box variant. The easiest way to do this is to simply embed all content of the group box into a user pane.

#### Icon CDEF

This CDEF merely takes an ID to a cicn, ICON, or icon suite in its Value parameter on creation and displays that icon in its contrlRect. After the control is initialized, the value parameter is reset to zero.

There is a 'no track' variant which tells it to just return the part hit immediately and return, it doesn't actually track the mouse in this mode. This is used in dialogs when the dialog has an embedding hierarchy and wants an icon. This control is created with the no-track variant so that it behaves like it always has.

This control’s min, max, and value parameters are reserved.

#### Image Well

A simple CDEF performs imaging for icons and picts. The control is controlled in much the same way as the bevel button, but with fewer options and states. Menus may not be attached. Currently, it is used for display only, but future versions will support drag and drop functionality that an application can plug into. This control’s min, max, and value parameters are reserved.

#### Little Arrows

This simple CDEF acts like a subset of a scroll bar, i.e., it returns the part codes kControlUpButtonPart and kControlDownButtonPart. Callers use ControlActionUPPs (as with scroll bars) to be called back during tracking. The control has a minimum, maximum and value.

**List Box**

This CDEF allows clients to put a List Box into dialogs with minimal effort. An auxiliary resource type ('ldes') is used to provide the information necessary to create the list. The ID of this resource is passed into the Value parameter of the control when created. The Min, Max, and Value parameters currently serve no purpose. Cursor navigation is included for moving around with the arrow keys. Double-clicking an item returns a special part code to make you aware of such an action. There is a keyFilter available for this control. The list handle that the list box creates has its refCon filled out with the control handle of the list box control. This allows any custom LDEFs to determine whether or not the control should be drawn active or inactive by looking at the current state of the control. Clients should never reset this field to anything else and instead use the control's refCon field to store data. This control's min, max, and value parameters are reserved.

**Picture CDEF**

This CDEF merely takes an ID to a PICT resource in its Value parameter on creation and displays that picture in its contrlRect. There is a 'no track' variant which tells it to just return the part hit immediately and return, it doesn't actually track the mouse in this mode. This control's min, max, and value parameters are reserved.

**Placard**

This CDEF implements a small placard control. Its value, min, and max are reserved, as a future version will allow a pushbutton variant. This control supports embedding.

**Popup Button**

The popup button has been revved for the new grayscale look. The older implementation made many assumptions about the menu handle, its numbering, and its inclusion in the menu list, which diminished its usefulness in modeless panels and other contexts requiring closer control of the menu handle. A special menu ID (-12345) value tells the control not to try to create the menu handle itself, to allow for a NULL menu handle, and to insert the menu in the menu list with a unique ID only for a short time directly around the PopupMenuSelect call.

Setting the title width (Min) to -1 tells the popup button to auto-calculate the title width.

**Popup Glyph**

This is a simple CDEF with 4 variants that draws the popup glyph. The pixel data will be embedded in the CDEF. The CDEF does no mouse tracking or hiliting and has no values. It's min, max, and value fields are reserved.

**Progress Indicator**

Both determinate and indeterminate progress indicators are supported, and it is possible for one mode to transition to the other.

Indeterminate progress is accomplished using the Data Access APIs to set the control's indeterminate tag. From that point forward, the control will request idle events, which is what drives the animation. By resetting the indeterminate flag, it resumes its normal function.

**Scroll Bar**
Scroll bars have been given a facelift. They also have support for ghosting its indicator, as well as live scrolling.

**Separator Line**
A simple CDEF draws separator lines. Orientation of the bounding rectangle will determine the orientation of the line, i.e. if the bounding rect is more horizontal than vertical the horizontal line will be drawn. (Scroll bars currently do this as well). The CDEF does no mouse tracking or hiliting and has no values. It's min, max, and value fields are reserved.

**Sliders/Slider Tick Marks**
The slider control is relatively straightforward, with a minimum, maximum, and value. Depending on whether he control is taller or wider, a vertical or horizontal slider will be created. This CDEF supports ghosting and live feedback. By default, the indicator points either down or to the left, depending on the orientation. You can reverse this by adding the kSliderReverseDirection variant into the procID for this control. Ticks marks are normally not shown, but can be by adding kSliderHasTickMarks to the procID. The number of tick marks is passed in the value parameter - after initialization in this case, the value is set to the minimum and the number of tick marks is stored internally.

There is also a non-directional thumb variant. Using this variant disables the tick mark and reverse direction options. They are not allowed to be combined.

This control also supports live feedback.

**Static Text**
CDEF Implementation of Static Text. It supports getting and setting its style, like a dialog's ictb information. It also supports different justification options.

This control's min, max, and value parameters are reserved.

**Tabs**
The tab mechanism is implemented as a CDEF. An auxiliary 'tab#' resource holds the tab names and icon IDs. This resource ID is passed into the Value parameter of the control. Callers check the value after getting a hit; they switch to the appropriate pane through whatever mechanism they prefer, such as AppendDITL. The value of the control is the one-based index of the currently selected tab (front most tab).

It is possible to get the content rectangle for tabs and also get/set a particular tab's enabled state using the Data Access routines. This CDEF is an embedder.

**User Pane**
This CDEF is a general purpose control. It is used as the root pane for a window, but could also be used by clients to hook in callbacks for drawing, hit testing, etc. This is especially useful for frameworks that wish to tap into the new control manager's hierarchy. This should be used in place of UserItems in dialogs when in Appearance-Savvy mode (see below).

This control’s min, max, and value parameters are free for you to use once the control is created.

#### Window Header

A CDEF provides both icon and list view headers for windows. This two-state functionality is handled by a variant. The list view header lacks the bottom line. This is an embedding control.

This control’s min, max, and value parameters are reserved.

### MENU DEFINITIONS

#### New Menu Features

The following support has been added to the standard menus:

- Support for extended modifiers keys (option, control, etc.)
- Support for icon suites
- Ability to store application specific data for a menu item
- Ability to set a command ID for a menu item.
- Ability to set a hierarchical ID for an item with a high-level API.

A replacement to MenuKey has been added to allow modifiers to be considered when searching for the item. The new routine is called MenuEvent and takes an event record as its only parameter. It returns a long, just like MenuSelect.

For resource-based creation of menus, a new resource type has been added, 'xmnu'. This resource contains the extended menu information for each item in a menu. After creating a menu, GetMenu looks for an 'xmnu' resource with the same ID. The information is set for each menu item. At that point the resource can be purged or released.

#### Menu Bar

The menu bar has been changed to accommodate the new look.

#### Pull Down Menus

These have been changed to accommodate the new look. They handle extended modifier keys and deal with the new extended information mentioned above.

## Dialog Manager

Quite a few new features have been added to the Dialog Manager.

## FEATURE FLAGS

A caller may activate the New Appearance mode of the Dialog Manager on a per-dialog basis by relating a special resource to their DLOG and ALRT resources. The new resource types which hold the new, extended information are the 'dlgx' and 'alrx'. In the new resources, there is a new flag word that is used to determine dialog or alert features. Whenever a dialog is created via GetNewDialog, Alert, StopAlert, CautionAlert, or NoteAlert, after the DLOG or ALRT is read in, we search for a resource type of 'dlgx' or 'alrx', respectively, with the same ID as the DLOG or ALRT. If the resource is found, we read the information and use it to help create the dialog.

Clients creating dialogs without using GetNewDialog will be able to use these features by calling the new NewFeaturesDialog routine, which in addition to the usual NewDialog parameters also takes a flag word parameter to specify the desired features. Following is a rundown of the features that can be set in the extended information.

#### Use Theme Backgrounds

If the kDialogFlagsUseThemeBackground bit is set in the flags, we set the background color to the correct color for the current theme automatically.

#### Use Control Hierarchy

When the kDialogFlagsUseControlHierarchy bit is set, right after the window is created, the CreateRootControl routine is called for the window to establish an embedding hierarchy. This has two effects; first, the hierarchy is established and embedding of controls is possible; second, all dialog items (except user items, for reasons explained later) are controls. This means that if a static text item is in the DITL, a static text control is created instead of the old dialog primitive. This ends up having many advantages, such as homogenous treatment of dialog items, and the ability to disable all items in a dialog, including edit text.

GetDialogItem in this situation still behaves as it always has. To get the control handle for an item, use the new API GetDialogItemAsControl. With a control handle, you can do cool stuff like disable static and edit text items, which was never before possible without great pain.

#### Use Theme Controls

This bit should generally always be set for Appearance-savviness. It tells the dialog manager that when it encounters a push button, check box, or radio button primitive (i.e. dialog items of type kButtonDialogItem, kRadioButtonDialogItem, etc.) to create a new theme-savvy control instead of the classic control. This bit is necessary, otherwise the Dialog Manager won't know the difference, since there is no other way to tell that we want theme controls. The use theme background doesn't have to be set to use this bit. In fact, there are times when that is the desired behavior.

#### Handle Movable Modal

The kDialogFlagsHandleMovableModal bit in the flags tells the Dialog Manger to handle all movable modal behavior if ModalDialog is called with this window frontmost. This only works if the window itself is a movable modal dialog. When told to handle this situation, the Dialog Manager handles window dragging and allows the user to click into another application.

Event filtering is handled a little differently, in that ALL events are passed thru to the application in this mode. This allows the app to handle suspend and resume events, as well as handle Apple Events if it so wished.

### ALERTS

#### Movable Alerts

If the kAlertFlagsHandleMovableModal bit is set, it tells the Dialog Manager whether or not this alert should be movable. If so, a movable modal dialog is used instead of a standard modal one. The behavior is the same as it is for normal movable modals, as mentioned above.

It is also possible, thru the use of a 'alrx' resource, to specify a title for a movable alert.

Another field in the 'alrx' resource tells the dialog manager to directly use the new appearance-savvy defprocs instead of going thru the mapping layer.

It is now possible to specify a refCon for an alert in the 'alrx' also.

### AUTOMATIC SIZING

The Dialog Manager introduces a new routine, AutoSizeDialog, that automatically resizes a dialog to fit all static text contained in it. This is used by the new StandardAlert routine to ensure that all the text of an alert is visible and doesn't get truncated. The DITL is iterated over, looking for static text items. When one is found, the item is resized, the window height is adjusted, and any items below the static text item are moved downward the appropriate amount.

This API only adjusts the height, not the width, of a dialog. It also assumes that items are placed reasonably and formatted correctly to display text in the standard format.

## Window Manager

### COLLAPSING API

The routines for collapsing and uncollapsing a window are exposed to developers. This will allow clients such to control the collapsed state of windows in an intelligent manner. A good example of this might be uncollapsing automatically after double-clicking on an icon to bring its window forward.

There are four routines to do with collapsing: CollapseWindow, CollapseAllWindows, IsWindowCollapsed, and IsWindowCollapsable. These routines only affect windows that advertise that they support the collapsing API, which brings us to window features.

## WINDOW FEATURES

It is possible to determine a window's features thru the GetWindowFeatures API. This is implemented thru a new message, kWindowMsgGetFeatures, which is just like the corresponding version for controls. In response to the GetFeatures message, the window should return a bitfield representing the features it supports. Those features are listed here:

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

## WINDOW DEFINITIONS

#### WindowShade Widget

The new WDEFs in Mac OS 8 support the collapse widget. If a window can be collapsed, a collapse box appears in the title bar of the window. A click on this returns the part code inCollapseBox.

The collapsing behavior is handled automatically by the system. Future releases will allow you to intercept this to handle it yourself if you have special requirements.

#### Document Windows (WDEF 64)

This WDEF draws in the new grayscale look, and supports the new horizontal and vertical zoom boxes. The variants are more straightforward than the old WDEF 0 variants with respect to how grow, zoom, etc. are specified. This WDEF also supports GetWindowFeatures and GetWindowRegion. When called thru the mapper WDEF, this defproc operates in a compatibility mode whereby the grow box is not drawn unless DrawGrowIcon is called. When used directly, the variant alone dictates whether a grow box will be drawn. There is no need to call DrawGrowIcon in this situation.

#### Dialogs (WDEF 65)

The new WDEF for dialogs supports modal, movable modal, plain, and shadow dialog variants. When called from the mapper WDEF, this defproc operates in a compatibility mode. When in this mode, a 3-pixel space exists between the content region and the structure region, as it always did in the past. When used directly, this area is banished and content can finally be run up to the edge of the window. There have been numerous applications which were doing some pretty wild stuff to make this happen in the past.

#### Utility Windows (WDEF 66 & 67)

WDEF 66 is the normal, top-title-bar variant, and 67 is the side title bar variant. The old WDEF was split in two to allow for the new horizontal and vertical zoom boxes. This defproc runs in a compatibility mode when called from the mapper WDEF. When in this mode, the grow box is not drawn until a call to DrawGrowIcon is made. When used directly (no compatibility mode), the presence of a grow box is completely driven thru the variant codes.

## SUPPORT

#### Gestalt Selector

On startup, the extension installs a gestalt selector to indicate that Appearance is running. The result returned is a bit field with the following possible values:

```
enum
{
  gestaltAppearanceExists= 0,
  gestaltAppearanceCompatMode= 1
};
```

The `gestaltAppearanceExists` bit indicates appearance is running. `gestaltAppearanceCompatMode` indicates that we are running in compatibility mode and are using the system 7 defprocs. The `gestaltAppearanceCompatMode` bit indicates that system-wide appearance is currently off.

# Control Manager Reference

This section describes the new routines added to the Control Manager as well as the new behavior of several classic routines.

## Internal Routines

The routines in this section are SPI only and are utilities used by the Control Manager and Dialog Manager.

### SendControlMessage

Use the SendControlMessage to send a low-level message to a control.

```
pascal SInt32 SendControlMessage( ControlRef theControl,
                                  SInt16 message, SInt32 param )
```

DESCRIPTION

The SendControlMessage sends the specified message to a CDEF and gets a response.

### DumpControlHierarchy

This routine dumps the contents of the control hierarchy for the specified window into a file.

```
pascal OSErr DumpControlHierarchy( WindowRef window,
const
                                              FSSpec* file )
```

DESCRIPTION

DumpControlHierarchy dumps a text listing of the current pane hierarchy for the window specified into the file specified, overwriting any existing file.

# Creating Controls

### NewControl

NewControl is adjusted to automatically embed the control into the root control if the root exists. All other aspects of behavior are the same.

# Embedding Controls

The routines in this section allow you to create the root control for a window and also embed controls within others.

### CreateRootControl

Use CreateRootControl to create the root container control for a window and enable embedding in a window.

```
pascal OSErr CreateRootControl( WindowRef window, Con-
trolRef* control )
```

DESCRIPTION

CreateRootControl creates the top-level container control for a window. From that point on, the embedding routines EmbedControl and AutoEmbedControl can be used. If controls were already added to the window when CreateRootControl is called, an error is returned and the root is not created.

### GetRootControl

GetRootControl returns the root container control for the specified window.

```
pascal OSErr GetRootControl( WindowRef window,
                                ControlRef* control )
```

DESCRIPTION

GetRootControl returns the root container control for the window specified. If a hierarchy doesn't exist, an error is returned.

### EmbedControl

Use EmbedControl to place one control inside another.

```
pascal OSErrEmbedControl( ControlRef control,
                                    ControlRef container );
```

DESCRIPTION

EmbedControl is used to place one control inside of another control. You might use this to place a radio button inside of a group box, for example. If the container does not support embedding, or there is no root control for the container's owning window, an error is returned.

### AutoEmbedControl

Use AutoEmbedControl to have a control find its best embedding container.

```
pascal OSErrAutoEmbedControl( ControlRef control,
                                        WindowRef window )
```

DESCRIPTION

The AutoEmbedControl automatically finds the 'best fit' container for a control. It essentially searches for the smallest embedder control that contains the given control and automatically embeds the control in there. The Dialog Manager uses this to automatically assume the embedding hierarchy from the DITL. If there is no root control for the window, an error is returned.

## Drawing Controls

### DrawOneControl

DrawOneControl has been changed to draw all controls contained within a control if the control passed in is an embedder and the window has a root control. If the root control for a window is passed in, the result is the same as if DrawControls was called.

### DrawControls

If a root control is present, DrawControls uses the hierarchy to determine drawing order and draws using that information, else it draws it in the classic manner.

### UpdateControls

If a root control is present, UpdateControls uses the hierarchy to determine drawing order and draws using that information, else it draws it in the classic manner.

### DrawControlInCurrentPort

Use DrawControlInCurrentPort to tell a control to draw in the current port and not in its owner's port.

```
pascal void DrawControlInCurrentPort( ControlRef control
);
```

DESCRIPTION

DrawControlInCurrentPort draws a control in whatever the current port is at the time. This is unlike DrawOneControl (or DrawControls / UpdateControls) in that controls normally are forced to draw in their owner's port. The Control Manager sees to this. This routine is designed to allow for offscreen drawing. All system controls support this type of functionality. For a custom control to work right with this, it just needs to assume that the right port is always set up for it, and not set the port to its owner. If the control has subcontrols, they are drawn as well.

## Testing and Changing Control Settings

The routines in this section allow you to manipulate controls and check their state.

### IsControlActive

Use IsControlActive to tell whether a control is currently active.

```
pascal Boolean IsControlActive( ControlRef control );
```

DESCRIPTION

IsControlActive is used to tell whether the given control is active, that is, it is not disabled or pending disabled (latent).

## IsControlVisible

Use IsControlVisible to tell whether a control is visible.

```
pascal Boolean IsControlVisible( ControlRef control );
```

DESCRIPTION

IsControlVisible returns true if the given control is currently visible.

## SetControlVisibility

Use SetControlVisibility to make a control visible or hidden.

```
pascal Boolean SetControlVisibility( ControlRef control,
Boolean visible, Boolean draw );
```

DESCRIPTION

SetControlVisibility is very useful when you want to hide or show a control. Unlike the HideControl and ShowControl APIs, SetControlVisibility allows you to control whether drawing occurs on screen. By passing false into the draw parameter, you can set the control's visibility without any unsightly drawing.

## ActivateControl

Use ActivateControl to activate a control and any subcontrols.

```
pascal OSErr ActivateControl( ControlRef control );
```

DESCRIPTION

ActivateControl activates the given control. If the control is an embedder and embedding is on, this activates all subcontrols that are currently latent. Passing the root control into this routine will activate all controls in the root's window. You can use this routine in that manner to activate all controls in a window when the window becomes active. If a control supports activate events, it will receive an activate event before getting a draw call to update its appearance.

You should always use this routine instead of HiliteControl( 0 ) to activate a control when a root control is present. It doesn't hurt to use it other times as well.

## DeactivateControl

Use ActivateControl to deactivate a control and any subcontrols.

```
pascal OSErr DeactivateControl( ControlRef control );
```

DESCRIPTION

DeactivateControl deactivates the given control. If the control is an embedder and embedding is on, this deactivates all subcontrols as well. Any subcontrols that are enabled become latent. Passing the root control into this routine will deactivate all controls in the root's window. You can use this routine in that manner to deactivate all controls in a window when the window becomes inactive.If a control supports activate events, it will receive an activate event before getting a draw call to update its appearance.

Calling this routine when a window is inactive is the only way to guarantee that the item will truly get disabled when a root control is present. Calling HiliteControl( 255) will short-circuit because the hilite is already 255. You should generally always use this routine instead of HiliteControl( 255).

## SetControlFontStyle

Use SetControlFontStyle to give a control a special font style.

```
pascal OSErr SetControlFontStyle( ControlRef control,
                                  ControlFontStylePtr style
);
```

DESCRIPTION

SetControlFontStyle sets the font style of the given control to that specified in style. Normally a control uses the System font unless directed to use the window font via a variant. This routine allows you to override that and force the control to use a special font style. Not all controls support this feature. To clear a style in effect, simply pass in a style record with a cleared flags field. The CDEF is expected to respond by falling back to using the old system / window font logic.

### ShowControl

If embedding is enabled for a window, this call will show any subcontrols that are embedded within the control passed in. Passing the root control into this routine will show all items in a window, if they were previously hidden.

### HideControl

If embedding is enabled for a window, this call will hide any subcontrols that are embedded within the control passed in. Passing the root control into this routine will hide all items in a window, if they were previously hidden. Hiding will save the states of all subpanes so that when the control is later shown, all panes that were visible when it was originally hidden will be displayed.

### MoveControl

If embedding is enabled for the control's window, this call will move the control and any subcontrols it might have.

### HiliteControl

If embedding is enabled for the control's window, this call does the following:

• If the part code passed in is 0, the control and all subcontrols are activated

• If the part code passed in is 255, the control and all subcontrols are deactivated.

• If the part code is any other value, the control's hilite value is set, and:

• If the control is inactive, it remains inactive, but will take on the new hilite when activated.

• If the control is active, it will be drawn in its new hilite state.

In addition, if a control is caused to become active/inactive, it will call the control with an activate message if the CDEF supports it.

If an embedding hierarchy is not present, this routine behaves as it always has.

# Handling Mouse Events in Controls

## FindControl

FindControl is changed to use the hierarchy to determine what control the mouse went down in before calling TestControl. If no hierarchy is present, it uses the control list as usual.

## FindControlUnderMouse

Use FindControlUnderMouseto to locate a control under the given point, regardless if any parts of the control are hit.

```
pascal ControlRef FindControlUnderMouse( Point where,
WindowRef window, SInt16* part )
```

DESCRIPTION

FindControlUnderMouse is a variation of FindControl that, unlike FindControl, actually returns the ControlRef for the control currently under the given point. FindControl only returns the ControlRef if a part was hit. This can be used to help adjust the cursor, etc. when over particular items. FindDialogItem uses this when a control hierarchy is present for a dialog.

## HandleControlClick

Use HandleControlClick to handle a mouse click on a control.

```
pascal SInt16 HandleControlClick( ControlRef control,
Point where, SInt16 modifiers, ControlActionUPP action )
```

DESCRIPTION

Like TrackControl, this routine tracks a control until the mouse is released. All that applies to TrackControl applies here as well. The difference, however, is that this routine allows modifier keys to be passed in so that the control may use these if the control is set up to handle its own tracking.

### SetControlSupervisor

Use SetControlSupervisor to route mouse down events from one control to another.

```
pascal OSErr SetControlSupervisor( ControlRef control,
                                   ControlRef supervisor
)
```

DESCRIPTION

This routine is used to make sure that things like list box controls work correctly. List boxes control their scroll bars in an intimate way, and handle the tracking in LClick. Because the new hierarchy is in place. When these controls are created, they get their own panes and report that they are hit (as they rightfully should). This presents a problem in that the list box will never know it got hit (after all, we hit the scroll bar, right?), and LClick will never be called. This routine alleviates this problem by routing the event to the supervisory control, in this case the list box.

## Handling Keyboard Events in Controls

### HandleControlKey

Use HandleControlKey to send a keyboard event to a control.

```
pascal SInt16 HandleControlKey( ControlRef control,
SInt16 keyCode, SInt16 charCode, SInt16 modifiers );
```

DESCRIPTION

HandleControlKey is used when a control supports focus. It sends the necessary information, keyCode, charCode, and modifiers into the CDEF so that it can process it as it wished. This routine returns the part code that the control considers 'hit' by the keyboard event.

# Idle Processing for Controls

### IdleControls

Use IdleControls to give idle time to controls in a window.

```
pascal void IdleControls( WindowRef window );
```

DESCRIPTION

IdleControls calls each control in a window who wants idle events with an idle event so it can do its idle-time processing. The Chasing Arrows CDEF uses this time to perform its animation.

# Determining Features of Controls

### GetControlFeatures

Use GetControlFeatures to find out what messages a control supports.

```
pascal UInt32 GetControlFeatures( ControlRef control )
```

DESCRIPTION

GetControlFeatures returns a 32-bit bitfield which represents the different features that a control supports.

### GetBestControlRect

Use GetBestControlRect to find out what a control's favorite size is.

```
pascal OSErr GetBestControlRect( ControlRef control,
Rect* rect,
                                        SInt16* baseLineOffset
)
```

DESCRIPTION

GetBestControlRect is implemented on top of the kControlMsgCalcBestRect control message. It allows an application to find out what the optimal control size is and where text should be placed in relation to the control's bottom coordinate. You should generally pass in an empty rect (0, 0, 0, 0). This routine will call the CDEF that drives the specified control to fill out the right and bottom sides of the rectangle, so you can determine its metrics for correct placement, etc. This allows you to autosize some controls based on their text, such as Push Buttons. The StandardAlert routine uses this call to help its button placement algorithm. The baseLineOffset parameter returns where the text baseline should be in relation to the bottom of the control rectangle. It is a negative value.

# Handling Focus for Controls

The routines in this section allow you to manage keyboard focus.

### GetKeyboardFocus

Use GetKeyboardFocus to get the current keyboard focus for a window.

```
pascal OSErr GetKeyboardFocus( WindowRef window,
                               ControlRef* control );
```

DESCRIPTION

The GetKeyboardFocus returns the ControlRef of the control which currently is the keyboard focus of the window specified.

### SetKeyboardFocus

Use SetKeyboardFocus to set the current keyboard focus for a window.

```
pascal OSErr SetKeyboardFocus( WindowRef window,
                           ControlRef control, FocusPart part
)
```

DESCRIPTION

The SetKeyboardFocus routine is used to set the current keyboard focus to the specified control. The part parameter tells the control what part to focus on. This parameter can be a positive part code or one of the constants, kFocusNoPart, kFocusNextPart, or kFocusPrevPart. These values tell the control to clear, advance, or reverse, its focus. If the control cannot become the focus for some reason, an error is returned. Using this routine, it is possible to set the focus to a disabled or invisible control. You might need to do this when preparing a dialog while hidden.

## AdvanceKeybordFocus

Use AdvanceKeyboardFocus to move the keyboard focus forward.

```
pascal OSErr AdvanceKeyboardFocus( WindowRef window );
```

DESCRIPTION

AdvanceKeyboardFocus attempts to advance forward to the next focusable item in a window and make it the current focus. It skips over disabled and hidden items.

## ReverseKeyboardFocus

Use ReverseKeyboardFocus to move the keyboard focus backwards.

```
pascal OSErr ReverseKeyboardFocus( WindowRef window );
```

DESCRIPTION

ReverseKeyboardFocus attempts to advance backwards to the next focusable item in a window and make it the current focus. It skips over disabled and hidden items.

## ClearKeyboardFocus

Use ClearKeyboardFocus to clear any keyboard focus that exists in a window.

```
pascal OSErr ClearKeyboardFocus( WindowRef window );
```

DESCRIPTION

Clear keyboard focus tells any control that might be the current focus to clear its focus. After the successful execution of this routine, nothing in a window has the keyboard focus.

## Getting and Setting Control Data

The routines in this section allow you to get and set values in a control's private data. You might use this to get the text from an edit text or static text control, or set the indeterminate flag of a progress indicator.

### SetControlData

Use SetControlData to set a piece of data for of a control.

```
pascal OSErr SetControlData( ControlRef control,
                        ControlPartCode part, ResType tag,
                        Size dataSize, Ptr dataPtr );
```

DESCRIPTION

The SetControlData routine is used to set the data represented by `tag` of the specified control to the data pointed to by dataPtr. The part parameter indicates which part of the control should get the data.

Passing kControlEntireControl in for part indicates it doesn't belong to any specific part, but the control as a whole. For some pieces of data, part may not make sense and is ignored by the CDEF.

### GetControlData

Use GetControlData to get a piece of data from a control.

```
pascal OSErr GetControlData( ControlRef control,
                        ControlPartCode part, ResType tag,
                        Size bufferSize, Ptr buffer, Size*
                        actualSize );
```

DESCRIPTION

The GetControlPartText is used to get the data represented by tag in the specified control. The part parameter indicates which part of the control the data should come from. The actual size of the data is returned in actualSize. You can pass nil in this parameter to avoid getting the size back. Calling this routine will a nil buffer pointer is functionally equivalent to calling GetControlDataSize.

Passing kControlEntireControl in for part indicates it doesn't belong to any specific part, but the control as a whole. For some pieces of data, part may not make sense and is ignored by the CDEF.

### GetControlDataSize

Use GetControlDataSize to set the size of a data member of a control.

```
pascal OSErr GetControlDataSize( ControlRef control,
                        ControlPartCode part, ResType tag,
                        Size* size );
```

DESCRIPTION

The GetControlDataSize routine is used to get the size of a specific piece of data the specified control owns. The part parameter indicates which part of the control should be checked for the data.

Passing kControlEntireControl in for part indicates it doesn't belong to any specific part, but the control as a whole. For some pieces of data, part may not make sense and is ignored by the CDEF.

## Iterating Over the Control Hierarchy

The routines in this section allow you to walk the control hierarchy of a window.

### CountSubControls

CountSubControls returns the number of controls embedded within a control.

```
pascal OSErr CountSubControls( ControlRef control,
                                        SInt16* numChildren );
```

DESCRIPTION

The CountSubControls routine returns the number of controls that are inside of the given control. If the control does not support embedding, or embedding is not enabled in its window, an error is returned.

### GetIndexedSubControl

GetIndexedSubControl returns a specific control embedded within another control.

```
pascal OSErr GetIndexedSubControl( ControlRef control,
                                   SInt16 index, ControlRef*
child );
```

DESCRIPTION

The GetIndexedSubControl routine returns the control at the index specified within the control passed in. If the control does not support embedding, or embedding is not enabled in its window, an error is returned. If the index passed in is invalid, an error is returned.

### GetSuperControl

GetSuperControl returns the parent of a control.

```
pascal OSErr GetSuperControl( ControlRef control,
                                        ControlRef* daddy );
```

DESCRIPTION

The GetSuperControl routine returns the parent control of the given control. If the control does not support embedding, or embedding is not enabled in its window, an error is returned.

## RemovingControls

### DisposeControl

DisposeControl is changed to remove any subcontrols that might be embedded within it. Passing the root control into this routine is the same as calling KillControls. In fact, this is what KillControls does.

### KillControls

KillControls gets the root control for a window and if it exists, it disposes of it and all subcontrols via a call to DisposeControl. If a root control does not exist, it does the same thing it always has.

## Application-Defined Routines

This section describes routines that an application can provide to hook into the new architecture.

### MyKeyFilter

Controls that support keyboard focus often have the ability to allow filtering of keystrokes. This is accomplished by a key filter proc.

```
pascal KeyFilterResult MyKeyFilter(ControlRef theCon-
trol,
          SInt16* keyCode, SInt16* charCode, SInt16* mod-
ifiers);
```

`theControl` the control we are dealing with

`keyCode` the key code of the key that was pressed

`charCode` the character code of the key that was pressed

`modifiers` the modifiers that were down when the key was pressed

This callback should be called from a CDEF when its receives a key hit message. The callback can change the keystroke in any way they see fit, leave it alone, or completely block the CDEF from getting it. This does rely on the CDEF implementing this correctly. There are two results the key filter can return: kKeyFilterPassKey or kKeyFilterBlockKey to allow keystrokes thru or to block them, respectively.

## UserPane Callbacks

When using a UserPane control, you can hook callback procedures into it to have it call you back to draw, perform hit testing, etc. In its most basic form, it is just like an old-style UserItem. Essentially a UserPane is a real control which just calls you back to do all the fun stuff.

### MyUserPaneDrawProc

To handle drawing, you can attach a draw proc to a user pane control.

```
pascal void MyUserPaneDrawProc( ControlRef control,
SInt16 part );
```

`control` the control to draw

`part` the part to draw, 0 = everything

### MyUserPaneHitTestProc

To handle hit testing in a user pane, you can attach a hit testing procedure.

```
pascal ControlPartCode MyUserPaneHitTestProc( ControlRef
control,
                                                    Point
where );
```

`control` the control to test

`where` the point where the mouse went down, in local coordinates

When called with this message, your routine should determine what part, if any, the mouse hit in your control and return that part code as its result.

### MyUserPaneTrackProc

To handle tracking in a user pane, you can attach a tracking procedure. This routine will only get called if you've specified the HandlesTracking bit of the control features, which get passed into the value of the control on creation.

```
pascal ControlPartCode MyUserPaneTrackProc( ControlRef
control,
                    Point startPt, ControlActionUPP ac-
tionProc );
```

`control`the control to track

`startPt`the point where the mouse went down, in local coordinates

`actionProc`the address of a routine to call during tracking.

When called with this message, your routine should track your control, calling actionProc repeatedly until the mouse is released. The value of actionProc can be a valid procPtr, nil, or -1. -1 indicates the control should do what it wants to if it actually has some special autoTrack behavior it wants to add. Most of the time, -1 would probably be treated like nil, i.e. do nothing. When the mouse is released, the part the mouse was released on should be returned to indicate a successful tracking session.

### MyUserPaneIdleProc

To handle idle processing in a user pane, you can attach an idle procedure. This routine will only get called if you've specified the WantsIdle bit of the control features, which get passed into the value of the control on creation.

```
pascal void MyUserPaneIdleProc( ControlRef control );
```

`control`the control to idle

You can use this to take advantage of control idle time to do some animation, etc.

### MyUserPaneKeyDownProc

To handle keyboard event processing in a user pane, you can attach an key-down procedure. This routine will only get called if you've specified the SupportsFocus bit of the control features, which get passed into the value of the control on creation.

```
pascal ControlPartCode MyUserPaneKeyDownProc( ControlRef
control,
                SInt16 keyCode, SInt16 charCode, SInt16
modifiers );
```

`control` the control that received the key event

`keyCode` the key code of the key that was pressed

`charCode` the character that the key generated

`modifiers` the modifiers that were held down during the keypress

When called with this message, your routine should do whatever is right for your special item, returning the part code of the item that was hit, if you wish. The standard EditText control, for example, returns kControlEditTextPart so that DialogSelect will return the itemHit when a keystroke is pressed.

### MyUserPaneActivateProc

To handle activate / deactivate events in a user pane, you can attach an activate procedure. This routine will only get called if you've specified the WantsActivate bit of the control features, which get passed into the value of the control on creation.

```
pascal void MyUserPaneActivateProc( ControlRef control,
                                            Boolean acti-
vating );
```

`control` the control that is becoming active / inactive

`activating` true if the control is becoming active, false otherwise.

Your routine should do whatever is proper to become active or inactive, such as calling LActivate, etc.

### MyUserPaneFocusProc

To handle focus events in a user pane, you can attach an focus procedure. This routine will only get called if you've specified the SupportsFocus bit of the control features, which get passed into the value of the control on creation.

```
pascal ControlPartCode MyUserPaneFocusProc( ControlRef
control,
                                                    FocusPart
part );
```

`control` the control in question

`part` the part code to focus

This routine is called in response to a change in focus. The part code passed in can mean many different things:

| | |
|---|---|
| `kFocusNoPart` | Clear your focus, return kFocusNoPart |
| `kFocusNextPart` | Focus on the next item. If nothing is in focus now, focus the first item. If there are no more items, clear your focus and return kFocusNoPart. |
| `kFocusPrevPart` | Focus on the previous item. If nothing is in focus now, focus the last item. If there are no more items, clear your focus and return kFocusNoPart. |
| <part code> | Focus on this part. You can interpret this in any way you wish. |

It is very important that your return the right part code for what you consider to be focused after you are called with this. By returning kFocusNoPart, you are telling the Control Manager to go onto another control, or that you can't be focused right now and go bother someone else.

# Control Manager Summary

## Constants

```
/* New part codes returned by FindControl/TestControl/FindControlUn-
derMouse*/
enum
{
```

```
  kControlEditTextPart= 5,/* an edit text field was hit */
  kControlPicturePart= 6,/* a picture control was hit */
  kControlIconPart= 7,/* an icon control was hit */
  kControlClockPart= 8,/* a clock control was hit */
  kControlListBoxPart= 24,/* a list box was clicked */
  kControlListBoxDoubleClickPart= 25/* a list box was double-
clicked*/
};

/* values for focusing */
enum
{
  kFocusNoPart= 0,/*Lose focus or returned to mean focus lost*/
  kFocusNextPart= -1,/*Focus on next part, if any*/
  kFocusPrevPart= -2/*Focus on previous part, if any*/
};
typedef SInt16 FocusPart;

/* return results for key filters */
enum
{
  kKeyFilterBlockKey= 0,/* allow keypress to go thru to control */
  kKeyFilterPassKey= 1/* stop keypress from going to control */
};
typedef SInt16 KeyFilterResult;

/* Error codes */
enum
{
  errMessageNotSupported= -30580,
  errDataNotSupported= -30581,
  errControlDoesntSupportFocus= -30582,
  errWindowDoesntSupportFocus= -30583,
  errPaneNotFound     = -30584,
  errCouldntSetFocus= -30585,
```

```
  errNoRootControl= -30586,
  errRootAlreadyExists= -30587,
  errInvalidPartCode= -30588,
  errControlsAlreadyExist= -30589,
  errControlIsNotEmbedder= -30590,
  errDataSizeMismatch= -30591,
  errControlHiddenOrDisabled= -30592
};

/* Feature bits to be returned when a CDEF is called with a 'get features' msg*/
enum
{
  kControlSupportsGhosting= 1 << 0,
  kControlSupportsEmbedding= 1 << 1,
  kControlSupportsFocus= 1 << 2,
  kControlWantsIdle= 1 << 3,
  kControlWantsActivate= 1 << 4,
  kControlHandlesTracking= 1 << 5,
  kControlSupportsDataAccess= 1 << 6,
  kControlHasSpecialBackground= 1 << 7,
  kControlGetsFocusOnClick= 1 << 8,
  kControlSupportsCalcBest= 1 << 9,
  kControlSupportsLiveFeedback= 1 << 10
};

/* New control messages */
enum
{
  kControlMsgDrawGhost= 13,/* Draw a ghost image of the indicator*/
  kControlMsgCalcBestRect= 14,/* Calculate and return the best bounds*/
  kControlMsgHandleTracking= 15,/* Handles tracking */
  kControlMsgFocus= 16,/* Focus on a part, or clear focus */
  kControlMsgKeyDown= 17,/* Handle a keyboard event */
```

```
  kControlMsgIdle     = 18,/* Do some idle processing */
  kControlMsgGetFeatures= 19,/* Return 32-bit field of features */
  kControlMsgSetData= 20,/* Set a piece of private data */
  kControlMsgGetData= 21,/* Get a piece of private data */
  kControlMsgActivate= 22,/* Handle activate/deactivate */
  kControlMsgSetUpBackground= 23,/* Set up background color, etc */
  kControlMsgCalcValueFromPos= 26

};

/* These constants are meta-font values used in ControlFontStyleRecs*/
enum
{
  kControlFontBigSystemFont= -1,/* force to big system font */
  kControlFontSmallSystemFont= -2,/* force to small system font */
  kControlFontSmallBoldSystemFont= -3/* force to small bold system
font */
};

/* bits to set in flags of ControlFontStyleRec to control what to set
*/
enum
{
  kUseFontMask= 0x0001,/* Set the font */
  kUseFaceMask= 0x0002,/* Set the face */
  kUseSizeMask= 0x0004,/* Set the size */
  kUseForeColorMask= 0x0008,/* Set the foreground color */
  kUseBackColorMask= 0x0010,/* Set the background color */
  kUseModeMask= 0x0020,/* Set the text mode */
  kUseJustMask= 0x0040,/* Set the justification */
  kUseAllMask     = 0x00FF,/* Set all of the above */
  kAddFontSizeMask= 0x0100/* size represents value to add */
                                      /* to current font size */
};
```

```
/* some common data tags */
enum
{
  kControlFontStyleTag= 'font',/* font style (ControlFontStyleRec)*/
  kControlKeyFilterTag= 'fltr'/* key filter (ControlKeyFilterUPP)*/
};
```

## Data Types

```
/* This structure is passed to CDEFs when called via HandleControl-
Click, */
/* provided that the control does its own tracking */
struct ControlTrackingRec
{
  Point   startPt;
  SInt16  modifiers;
  ControlActionUPPaction;
};
typedef struct ControlTrackingRec ControlTrackingRec, *ControlTrack-
ingPtr;

/* This structure is passed to the CDEF for keyboard events */
struct ControlKeyDownRec
{
  SInt16  modifiers;
  SInt16  keyCode;
  SInt16  charCode;
};
typedef struct ControlKeyDownRec ControlKeyDownRec, *ControlKeyDownP-
tr;

/* this structure is passed to CDEFs for the Get/SetData message */
struct DataAccessRec
{
  ResType     tag;    /* 'name' of the data we are specifying */
  ControlPartCodepart;/* part of the control this tag refers to */
```

```
  Size        size;   /* size of the data or buffer */
  Ptr             dataPtr;/* pointer to the data or buffer */
};
typedef struct DataAccessRec DataAccessRec, *DataAccessPtr;

/* this is used by many controls to set a special font style */
struct ControlFontStyleRec
{
  SInt16  flags;      /* which pieces should we set */
  SInt16  font;       /* the font to set to (can be meta-font) */
  SInt16  size;       /* the size of the type */
  SInt16  style;      /* the style (bold, italic, etc.) */
  SInt16  mode;       /* text mode (srcOr, etc.) */
  SInt16  just;       /* justification */
  RGBColorforeColor;/* foreground color */
  RGBColorbackColor;/* background color */
};
typedef struct ControlFontStyleRec ControlFontStyleRec, *ControlFont-
StylePtr;
```

## Control Manager Routines

### Internal Routines

```
pascal SInt32 SendControlMessage( ControlRef theControl, SInt16 mes-
                                        sage, SInt32 param );
pascal OSErr GetControlDialogItemNo( ControlRef window, SInt16* item-
                                        No );
pascal OSErr SetControlDialogItemNo( ControlRef window, SInt16 itemNo
                                        );
pascal OSErr DumpControlHierarchy( WindowRef window, const FSSpec*
                                        file );
```

### Embedding Routines

```
pascal OSErr CreateRootControl( WindowRef window, ControlRef* control
                                        );
pascal OSErr GetRootControl( WindowRef window, ControlRef* control );
pascal OSErr EmbedControl( ControlRef control, ControlRef container);
pascal OSErr AutoEmbedControl( ControlRef control, WindowRef window
                                        );
```

### Drawing Controls

```
pascal void DrawControlInCurrentPort( ControlRef control );
```

### Testing and Changing Control Settings

```
pascal Boolean IsControlActive( ControlRef control );
pascal Boolean IsControlVisible( ControlRef control );
pascal OSErr ActivateControl( ControlRef control );
pascal OSErr DeactivateControl( ControlRef control );
pascal OSErr SetControlFontStyle( ControlRef control,ControlFontSty-
                                        lePtr style );
```

### Handling Mouse Events in Controls

```
pascal ControlRef FindControlUnderMouse
      ( Point where, WindowRef window, SInt16* part );
pascal SInt16 HandleControlClick( ControlRef control, Point where,
                                        SInt16 modifiers, Contro-
                                        lActionUPP action );
pascal OSErr SetControlSupervisor( ControlRef control, ControlRef
                                        boss );
```

### Handling Keyboard Events in Controls

```
pascal SInt16 HandleControlKey( ControlRef control, SInt16 keyCode,
                                        SInt16 charCode, SInt16
                                        modifiers );
```

### Idle Processing for Controls

```
pascal void IdleControls( WindowRef window )
```

### Handling Focus for Controls

```
pascal OSErr GetKeyboardFocus( WindowRef window, ControlRef* control
                                        );

pascal OSErr SetKeyboardFocus( WindowRef window, ControlRef control,
                                        FocusPart part );

pascal OSErr AdvanceKeyboardFocus( WindowRef window );

pascal OSErr ReverseKeyboardFocus( WindowRef window );
```

### Determining Features of Controls

```
pascal UInt32 GetControlFeatures( ControlRef control );
```

### Getting and Setting Control Data

```
pascal OSErr SetControlData( ControlRef control, ControlPartCode
                                        part, ResType tagName,
                                        Size size, Ptr dataPtr );

pascal OSErr GetControlData( ControlRef control, ControlPartCode
                                        part, ResType tagName,
                                        Size bufferSize, Ptr buff-
                                        erPtr, Size* actualSize);

pascal OSErr GetControlDataSize( ControlRef control, ControlPartCode
                                        part, ResType tagName,
                                        Size* size );
```

### Iterating Over the Control Hierarchy

```
pascal OSErr CountSubControls( ControlRef control, SInt16* numChil-
                                        dren );

pascal OSErr GetIndexedSubControl( ControlRef control, SInt16 index,
                                        ControlRef* child );

pascal OSErr GetSuperControl( ControlRef control, ControlRef* parent
                                        );
```

### Application-Defined Routines

```
pascal KeyFilterResult MyKeyFilter(ControlRef theControl, SInt16*
                                        keyCode,

                                        SInt16* charCode, SInt16*
                                        modifiers);

pascal void MyUserPaneDrawProc( ControlRef control, SInt16 part );

pascal ControlPartCode MyUserPaneHitTestProc

                                    ( ControlRef control, Point
                                        where );

pascal ControlPartCode MyUserPaneTrackProc
```

```
                                        ( ControlRef control, Point
                                          startPt, ControlActionUPP
                                          actionProc );
pascal void MyUserPaneIdleProc( ControlRef control );
pascal ControlPartCode MyUserPaneKeyDownProc
                                        ( ControlRef control, SInt16
                                          keyCode,
                                          SInt16 charCode, SInt16
                                          modifiers );
pascal void MyUserPaneActivateProc( ControlRef control, Boolean acti-
                                          vating );
pascal ControlPartCode MyUserPaneFocusProc
                                        ( ControlRef control, Focus-
                                          Part part );
```

# Dialog Manager Reference

This section describes the new routines added to the Dialog Manager as well as how some routines have been altered when running with a hierarchy.

## Creating Dialogs And Alerts

### NewFeaturesDialog

Call NewFeaturesDialog to create a dialog while specifying features for the dialog.

```
pascal DialogRef NewFeaturesDialog( void *wStorage,
          const Rect *boundsRect, ConstStr255Param title,
          Boolean visible, SInt16 procID, WindowRef be-
hind,
          Boolean goAwayFlag, SInt32 refCon,
          Handle itmLstHndl, SInt32 flags );
```

DESCRIPTION

This new routine allows the creation of a dialog while specifying options, such as theme savvyness, when the dialog is created.

## Presenting Dialogs

### StandardAlert

Call StandardAlert to use a system-supplied default alert template.

```
pascal OSErr StandardAlert(
                AlertType type,
                StringPtr error,
                StringPtr explanation,
                AlertStdAlertParamPtr param,
```

```
                                SInt16* itemHit );
```

DESCRIPTION

The StandardAlert routine is available as an easy to use template for creating alerts. It allows you to set the error text, as well as text to further explain what went wrong and how to fix it. The explanatory text is displayed in the small system font. The button that was hit (you can specify up to 3) is returned in itemHit.

The param parameter is used for special alert customization. You pass the address of a structure which contains information telling the Dialog Manager to make the alert movable, give the alert a filterproc, specify text for buttons, etc.

The alert can be movable by passing true in for the movable field of this structure. If you make your alert movable, you should make sure you pass a modal filter into filterProc. This will allow you to handle update events for window's behind the alert. Be aware that when you are using a movable alert, all events that your application receives are passed to you, i.e. the mask used on GetNextEvent is everyEvent.

You can have up to 4 buttons in the alert: an OK button, a cancel button, an 'other' button, and a help button. The buttons auto-size and autoposition themselves correctly in the alert for you. By default, the rightmost button text is "OK", the button to the immediate left of the OK button (cancel position) defaults to having the text "Cancel", and the 'other' (leftmost) button text is "Don't Save". The 'other' button is always left justified in the alert, and allows you to easily create a save alert. To specify that the default button names should be used, you pass -1 in for the text parameters. Passing nil in for a button text parameter indicates that no button should be displayed for that particular button. The rightmost button cannot be hidden, so passing nil is equivalent to passing -1 in for that parameter.

You can pass true in the param structure for the helpButton field to indicate that a help button is to be displayed.

You can specify which button is the default button, and which is the cancel button in the param struct. This controls which button is 'pressed' when typing return or enter and which button is 'pressed' by typing command-period.

By default, the StandardAlert routine positions the alert in the alert position on the parent window's screen. You can override this by passing another auto-centering constant into the position field of the structure.

Any errors are returned as the function result.

# ModifyingDialogs

The routines in this section allow you to manipulate aspects of a dialog.

## AutoSizeDialog

Use AutoSizeDialog to automatically resize a dialog to make sure all static text is visible.

```
pascal OSErr AutoSizeDialog( DialogPtr dialog );
```

DESCRIPTION

The AutoSizeDialog routine resizes the given dialog enough to show all static text. This is extremely useful in dialogs where the amount of text to be displayed is determined at runtime. Calling this routine iterates over the items in the dialog. For each static text item it finds, it adjusts the bottom of the window to accomodate the amount of text. Any items below a static text field being adjusted are moved down accordingly. If the dialog is visible when this routine is called, it is hidden, resized, and then shown. If the dialog has enough room to show the text as is, no resizing is done.

### MoveDialogItem

Use MoveDialogItem to move an item from one location to another, keeping any control rectangles in sync with the dialog item's rectangle.

```
pascal OSErr MoveDialogItem(DialogPtr dialog,
          SInt16 itemNo, SInt16 horiz, SInt16 vert);
```

DESCRIPTION

The MoveDialogItem should be called when moving any item in a dialog. If the item is a control, it will call MoveControl to move the control to the right place. This routine allows the dialog manager to make sure that the dialog item rectangles always match a control's rectangle. Simply calling MoveControl without adjusting the dialog item's rectangle can confuse the Dialog Manager.

### SizeDialogItem

Use SizeDialogItem to change a dialog item's size, keeping any control rectangles in sync with the dialog item's rectangle.

```
pascal OSErr SizeDialogItem(DialogPtr dialog,
          SInt16 itemNo, SInt16 width, SInt16 height);
```

DESCRIPTION

The SizeDialogItem should be called when resizing any item in a dialog. If the item is a control, it will call MoveControl to move the control to the right place. This routine allows the dialog manager to make sure that the dialog item rectangles always match a control's rectangle. Simply calling MoveControl without adjusting the dialog item's rectangle can confuse the Dialog Manager.

## Routines to Get Information About Dialog Items

The routines in this section allow you to get information about dialog items.

### GetDialogItemAsControl

Use GetDialogItemAsControl to get the actual control handle for a dialog item. This is especially useful when an embedding hierarchy is established.

```
pascal OSErr GetDialogItemAsControl(DialogPtr dialog,
                SInt16 itemNo, ControlHandle *control)
```

DESCRIPTION

GetDialogItemAsControl returns the control handle for the item specified. If the item is not a control, an error is returned. If a dialog is in embedding mode, all items are controls, and this routine will work on any item. This routine is useful when it is necessary to get the control for an edit text or static text item in a dialog.

## Changes To Existing Routines

This section documents new behavior of some of the classic Dialog Manager routines when a dialog is in the Appearance Savvy mode.

### GetNewDialog

GetNewDialog has been changed to check for the presence of a 'dlgx' resource with the same ID as the dialog resource ID passed in. If found, the information is read in and used. The 'dlgx' resource holds information such as the dialog flags for setting features like 'use theme background' and 'use embedding hierarchy'.

### Alert, CautionAlert, StopAlert, NoteAlert

These routines have been changed to check for the presence of a 'alrx' resource esource with the same ID as the alert resource ID passed in. If found, the information is read in and used. The 'alrx' resource holds information such as the alert flags for setting features like 'use theme background' and 'use embedding hierarchy'.

### GetDialogItem

GetDialogItem is changed so that calling it when the dialog has an embedding hierarchy talks to the controls to get the appropriate data. The API still returns the same types of handles as it always has.

### SetDialogItem

SetDialogItem is changed so that calling it when the dialog has an embedding hierarchy has a couple of restrictions: you can't change the type or handle of an item. User item drawing procedures can still be set. If an embedding hierarchy does not exist, it works as it always has in the past. Also, if you set the control rectangle on an item when an embedding hierarchy is present, it will move and resize the item appropriately for you.

### GetDialogItemText

GetDialogItemText is changed such that calling it when the dialog has an embedding hierarchy it will expect a ControlRef in the handle parameter. It will ask the EditText control for the text and return it in the string parameter.

### SetDialogItemText

SetDialogItem is changed so that calling it when the dialog has an embedding hierarchy  it can take either ControlRef or a text handle. The string passed in is set in the Edit Text control.

# Summary of the Dialog Manager

## Constants

```
/* Flags for NewFeaturesDialog, as well as dlgx and alrx resources */
enum
{
  kDialogFlagsUseThemeBackground = 1,
  kDialogFlagsUseControlHierarchy= 2,
  kDialogFlagsHandleMovableModal= 4,
  kDialogFlagsUseThemeControls = 8
};
```

## Creating Dialogs and Alerts

```
pascal DialogRef NewFeaturesDialog( void *wStorage, const Rect
                                     *boundsRect,
                                     ConstStr255Param title,
                                     Boolean visible, SInt16
```

```
                                        procID, WindowRef behind,
                                        Boolean goAwayFlag, SInt32
                                        refCon, Handle itmLstHndl,
                                        SInt32 flags );


pascal void AutoSizeDialog( DialogPtr dialog );

pascal OSErr StandardAlert( AlertType type,StringPtr error, StringP-
                                        tr explanation, Boolean
                                        movable, ModalFilterUPP
                                        filterProc, StringPtr de-
                                        faultText, StringPtr can-
                                        celText, tringPtr
                                        otherText, const FSSpec*
                                        agFileSpec, SInt16 agSe-
                                        quenceID, SInt16* itemHit
                                        );
```

# Window Manager Reference

This section describes the new routines added to the Window Manager.

## Window Collapsing Support

A new part code is introduced to represent the Collapse Box:

```
enum
{
      inCollapseBox = 9
};
```

Normally this is hidden from an application and taken care of by our SystemEvent patch. We are working on trying to establish a mechanism whereby apps can signal us that they want to receive these events themselves, bypassing the automatic behavior.

A new message has been created for getting the features of a window definition function:

```
enum
{
      kWindowMsgGetFeatures= 7
};
```

When sent this message, the WDEF should respond by filling out a 32-bit response field and returning it as the result of the definition function. The values that are currently valid are:

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

When a WDEF supports the collapsing, it knows to calculate its regions in its collapsed state by testing to see whether IsWindowCollapsed returns true. If so, it should calculates its structure region based on the collapsed state. If not, it should do its normal structure calculation.

# Collapsing Routines

## CollapseWindow

Call CollapseWindow to collapse a window. A window typically collapses to its title bar.

```
pascal OSErr CollapseWindow( WindowRef window, Boolean
collapse );
```

DESCRIPTION

This routine will either tell a window to collapse or uncollapse a window, depending on the value of the collapse parameter. If a window does not support collapsing thru the new mechanism, an error is returned.

## CollapseAllWindows

Call CollapseAllWindows to collapse or uncollapse all windows.

```
pascal OSErr CollapseAllWindows( Boolean collapseEm );
```

DESCRIPTION

This routine will either tell all windows that are in the current layer to collapse or uncollapse a window, depending on the value of the collapse parameter. If a window does not support collapsing thru the new mechanism, an error is returned.

## IsWindowCollapsed

Call IsWindowCollapsed to check to see whether a window is in its collapsed state.

```
pascal Boolean IsWindowCollapsed( WindowRef window );
```

DESCRIPTION

This routine will return true or false depending on the collapse state of the window. If the window does not support collapsing, false is returned.

# Routines to Get Window Information

## GetWindowFeatures

Use GetWindowFeatures to determine what features a window supports, as well as what type of window it is.

```
pascal OSStatus GetWindowFeatures(WindowPtr window,
                                  UInt32 *features)
```

DESCRIPTION

This routine is used to determine what features a window supports, such as collapsing, as well as getting what widgets are shown and what type of window you are dealing with (modal, for example). The features are returned in the features parameter. The bits are defined below:

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

## GetWindowRegion

Use GetWindowRegion to get a specific region of a window.

```
pascal OSStatus GetWindowRegion(WindowPtr window,
                    WindowRegionCode regionCode,
                    RgnHandle winRgn);
```

DESCRIPTION

GetWindowRegion allows you to get specific regions of a window, such as the grow box region, or the close box or title region.

# Summary of the Window Manager

## Constants

```
/* Part codes returned by FindWindow */
enum
{
  inCollapseBox = 9  /* Collapse box of a window was hit */
};

/* Window definition function task codes */
enum
{
      kWindowMsgGetFeatures= 7
};

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

enum {
  kWindowTitleBarRgn= 0,
  kWindowTitleTextRgn= 1,
  kWindowCloseBoxRgn= 2,
  kWindowZoomBoxRgn= 3,
  kWindowDragRgn  = 5,
  kWindowGrowRgn  = 6,
```

```
  kWindowCollapseBoxRgn= 7,
  kWindowStructureRgn= 32,
  kWindowContentRgn= 33
};


/* Window feature bits */
enum
{
  kWindowCanGrow  = (1 << 0),
  kWindowCanZoom  = (1 << 1),
  kWindowCanCollapse= (1 << 2),
  kWindowIsModal  = (1 << 3),
  kWindowIsMovableModal= (1 << 4)
};
```

## Collapsing Windows

```
pascal OSErr CollapseWindow( WindowRef window, Boolean collapse );
pascal OSErr CollapseAllWindows( Boolean collapse );
pascal Boolean IsWindowCollapsed( WindowRef window );
pascal OSStatus GetWindowFeatures(WindowPtr window, UInt32 *features)
pascal OSStatus GetWindowRegion(WindowPtr window, WindowRegionCode
                                        regionCode, RgnHandle win-
                                        Rgn)
```

# Menu Manager Reference

This section describes the new routines added to the Menu Manager. It has been extended to allow for more modifier keys to be used, such as shift and option. We have also added the ability to set a command ID for a menu item an other information. The routines in this section only function when the system supplied MDEF is used.

## Handling Keyboard Events

### MenuEvent

Call MenuEvent instead of MenuKey to determine if a keyboard equivalent for a menu item has been pressed when using the extended modifiers.

```
pascal UInt32 MenuEvent( EventRecord* event );
```

DESCRIPTION

MenuEvent is used to determine if a keyboard equivalent has been pressed by the user when using the new extended set of modifiers. The charCode and modifiers are normally taken from an EventRecord's message and modifiers fields.

## Getting and Setting Menu Item Data

### SetMenuItemModifiers

Call SetMenuItemModifiers to set the modifier keys to use for a specific menu item.

```
pascal OSErr SetMenuItemModifiers( MenuRef menu, SInt16
item,
                                   SInt16 modifiers );
```

DESCRIPTION

This routine will set the modifiers field of a menu item. The Command key is always implied to be set; however, it is possible to set a modifier sequence without the command key using the kMenuNoCommand flag in modifiers.

### GetMenuItemModifiers

Call GetMenuItemModifiers to set the modifier keys to use for a specific menu item.

```
pascal OSErr SetMenuItemModifiers( MenuRef menu, SInt16
item,
                                        SInt16* modifiers );
```

DESCRIPTION

This routine will get the modifiers field of a menu item.

### SetMenuItemCommandID

Call SetMenuItemCommandID to set the command ID for a specific menu item.

```
pascal OSErr SetMenuItemCommandID( MenuRef menu, SInt16
item,
                                        UInt32 commandID );
```

DESCRIPTION

This routine will set the command ID of a menu item. You can use the command ID as a position independent method of signaling a specific action in an application. After a successful call to MenuSelect, MenuKey, or ExtendedMenuKey, you can call GetMenuItemCommandID to get the command of the item and do the appropriate thing.

### GetMenuItemCommandID

Call GetMenuItemCommandID to get the command ID for a specific menu item.

```
pascal OSErr GetMenuItemCommandID ( MenuRef menu, SInt16
item,
                                        UInt32* commandID );
```

DESCRIPTION

This routine will get the command ID of a menu item. You can use the command ID as a position independent method of signaling a specific action in an application. After a successful call to MenuSelect, MenuKey, or ExtendedMenuKey, you can call GetMenuItemCommandID to get the command of the item and do the appropriate thing.

## SetMenuItemTextEncoding

Call SetMenuItemTextEncoding to set the script code to use for a specific menu item.

```
pascal OSErr SetMenuItemTextEncoding( MenuRef menu,
                          SInt16 item, TextEncoding encoding
);
```

DESCRIPTION

This routine will set the script code of a menu item. You can use this routine instead of the older method of using $1C in the command key equivalent field, which uses up that field as well as the icon field, which would hold the script code. Using this new method allows you to gain those fields back for your use. If a menu item has a command code of $1C when this routine is called, the command and icon fields are cleared, in favor of the new setting passed in and stored with the extended information for this item.

## GetMenuItemTextEncoding

Call GetMenuItemTextEncoding to get the script code for a specific menu item.

```
pascal OSErr GetMenuItemTextEncoding( MenuRef menu,
                SInt16 item, TextEncoding* encoding );
```

DESCRIPTION

This routine will get the script code of a menu item. If the script code is set using the old method ($1C in the key equivalent field), the script code is extracted from the icon field and returned. In general, when running Appearance, you should use the new SetMenuItemScript routine instead of the older method.

## SetMenuItemIconHandle

Call SetMenuItemIconHandle to set an icon to use for a specific menu item.

```
pascal OSErr SetMenuItemIconHandle( MenuRef menu, SInt16
item,
                                    MenuIconType type,
                                    Handle icon );
```

DESCRIPTION

This routine will set the icon of a menu item with an icon handle instead of an ID. This call allows you to set icons of type ICON, cicn, SICN, and icon suites. The menu will not dispose of any icons, it is up to the application to do so.

## GetMenuItemIconHandle

Call GetMenuItemIconHandle to get the handle of an icon you've set using SetMenuItemIconHandle.

```
pascal OSErr GetMenuItemIconHandle( MenuRef menu, SInt16
item,
                                    MenuIconType* type,
                                    Handle* suite );
```

DESCRIPTION

This routine will return the icon handle and the type of icon. If there is no icon for this item, nil is returned for the icon handle and kMenuNoIcon is returned for the type.

### SetMenuItemRefCon

Call SetMenuItemRefCon to set an application-specific piece of information for a menu item.

```
pascal OSErr SetMenuItemRefCon( MenuRef menu, SInt16
item,
                                        SInt32 refCon );
```

DESCRIPTION

This routine allows an application to set a piece of application specific data to a menu item.

### GetMenuItemRefCon

Call GetMenuItemRefCon to get an application-specific piece of information for a menu item.

```
pascal OSErr GetMenuItemRefCon( MenuRef menu, SInt16
item,
                                        SInt32* refCon );
```

DESCRIPTION

This routine returns the application specific data set for a menu item with SetMenuItemRefCon.

### SetMenuItemRefCon2

Call SetMenuItemRefCon2 to set an application-specific piece of information for a menu item.

```
pascal OSErr SetMenuItemRefCon2( MenuRef menu, SInt16
item,
                                        SInt32 refCon );
```

DESCRIPTION

This routine allows an application to set a piece of application specific data to a menu item.

## GetMenuItemRefCon2

Call GetMenuItemRefCon2 to get an application-specific piece of information for a menu item.

```
pascal OSErr GetMenuItemRefCon2( MenuRef menu, SInt16
item,
                                        SInt32* refCon );
```

DESCRIPTION

This routine returns the application specific data set for a menu item with SetMenuItemRefCon2.

## SetMenuItemHierarchicalID

Call SetMenuItemHierarchicalID to attach a submenu to a menu item.

```
pascal OSErr SetMenuItemHierarchicalID( MenuRef menu,
SInt16 item,
                                        SInt16 hierID );
```

DESCRIPTION

This routine allows you to attach a hierarchical menu to the given menu item. This is a high-level method than existed in the past, as it allows you to forget about how hierarchicals are attached to menus. Currently, the hierarchical menu ID is still restricted to 0-255, but a future version will allow a full 16-bit integer to be used.

### GetMenuItemHierarchicalID

Call GetMenuItemHierarchicalID to get an application-specific piece of information for a menu item.

```
pascal OSErr GetMenuItemHierarchicalID( MenuRef menu,
SInt16 item,
                                        SInt16* hierID );
```

DESCRIPTION

This routine returns the hierarchical menu ID for the given menu item. If the keyboard equivalent for the item is set to $1B, the menu ID is extracted from the item mark field and returned.

### SetMenuItemFont

Call SetMenuItemFont to set the font for a specific menu item.

```
pascal OSErr SetMenuItemFont( MenuRef menu, SInt16 item,
                                        SInt16 fontNum );
```

DESCRIPTION

This routine allows you to set the font to use when drawing the given menu item. This effectively allows you to set up a font menu with each item being drawn in the actual font.

### GetMenuItemFont

Call GetMenuItemFont get the font used by a specific menu item.

```
pascal OSErr GetMenuItemFont( MenuRef menu, SInt16 item,
                                        SInt16* fontNum );
```

DESCRIPTION

This routine returns the font for the given menu item.

## SetMenuItemKeyGlyph

Call SetMenuItemKeyGlyph to set the glyph to display as the keyboard equivalent for a specific menu item.

```
pascal OSErr SetMenuItemKeyGlyph( MenuRef menu,
                  SInt16 item, SInt16 glyph );
```

DESCRIPTION

This routine allows you to set a different glyph that would be normally displayed for the keyboard equivalent of a menu item. This is needed at times when the character code for some keys (like the delete key - ascii 8) does not map to the correct glyph in the font (which would be ascii 10). This glyph overrides the normal key that would be displayed. If zero is passed in for glyph, it clears the glyph and the menu item displays the actual character.

## GetMenuItemKeyGlyph

Call GetMenuItemKeyGlyph to get the glyph to display as the keyboard equivalent for a specific menu item.

```
pascal OSErr GetMenuItemKeyGlyph( MenuRef menu,
                      SInt16 item, SInt16* glyph );
```

DESCRIPTION

This routine allows you to get the glyph that overrides the keyboard equivalent for a menu item.

# Summary of the Menu Manager

## Constants

```
/* Modifier flags used by SetMenuItemModifiers */
enum
{
  kMenuOptionKey= 1,
  kMenuShiftKey= 2,
  kMenuControlKey= 4,
  kMenuNoCommandKey= 8
};

/* Valid icon types for SetMenuItemIconHandle */
enum
{
  kMenuIcon   = 1,    /* old ICON data */
  kMenuColorIcon= 2,/* cicn format */
  kMenuSmallIcon= 3,/* SICN format */
  kMenuIconSuite= 4,/* Icon Suite */
  kMenuIconRef= 5     /* Icon Ref */
};
```

## Routines

### Handling Keyboard Events

```
pascal SInt32 MenuEvent( EventRecord* event );
pascal OSErr SetMenuItemModifiers( MenuRef menu, SInt16 item, SInt16
                                          modifiers );
```

### Getting and Setting Menu Data

```
pascal OSErr SetMenuItemModifiers( MenuRef menu, SInt16 item, SInt16*
                                          modifiers );
```

```
pascal OSErr SetMenuItemCommandID( MenuRef menu, SInt16 item, UInt32
                                        commandID );

pascal OSErr GetMenuItemCommandID ( MenuRef menu, SInt16 item, UInt32*
                                        commandID );

pascal OSErr SetMenuItemScriptID( MenuRef menu, SInt16 item, Script-
                                        Code script );

pascal OSErr GetMenuItemScriptID( MenuRef menu, SInt16 item,Script-
                                        Code* script );

pascal OSErr SetMenuItemIconHandle( MenuRef menu, SInt16 item, MenuI-
                                        conType type, Handle icon
                                        );

pascal OSErr GetMenuItemIconHandle( MenuRef menu, SInt16 item, MenuI-
                                        conType* type, Handle*
                                        suite );

pascal OSErr SetMenuItemRefCon( MenuRef menu, SInt16 item, SInt32
                                        refCon );

pascal OSErr GetMenuItemRefCon( MenuRef menu, SInt16 item, SInt32*
                                        refCon );

pascal OSErr SetMenuItemRefCon2( MenuRef menu, SInt16 item, SInt32
                                        refCon );

pascal OSErr GetMenuItemRefCon2( MenuRef menu, SInt16 item, SInt32*
                                        refCon );

pascal OSErr SetMenuItemHierarchicalID( MenuRef menu, SInt16 item,
                                        SInt16 hier );
pascal OSErr GetMenuItemHierarchicalID( MenuRef menu, SInt16 item,
                                        SInt16* hier );

pascal OSErr SetMenuItemFont( MenuRef menu, SInt16 item, SInt16 font
                                        );
```

```
pascal OSErr GetMenuItemFont( MenuRef menu, SInt16 item, SInt16* font
                                                    );


pascal OSErr SetMenuItemKeyGlyph( MenuRef menu, SInt16 item, SInt16
                                                    keyGlyph );
pascal OSErr GetMenuItemKeyGlyph( MenuRef menu, SInt16 item, SInt16*
                                                    keyGlyph );
```

### Manipulating the Menu Bar Clock

```
pascal void DrawMenuBarClock( StringPtr text, Handle batteryIconSuite
                                                    );
```

# Appearance Manager Reference

This section describes the routines available as part of the Appearance Manager.

## Registering with Appearance

### RegisterAppearanceClient

Use RegisterAppearanceClient to let the Appearance Manager know you are a client of the new Appearance defprocs and APIs.

```
pascal OSStatus RegisterAppearanceClient(void)
```

DESCRIPTION

This routine should be called at the very beginning of your application if you are adopting Appearance. It tells the system to autoroute calls to the classic defprocs (WDEF 0, CDEF 0, etc.) to the new Appearance-Savvy defprocs automatically. This call is necessary to call to ensure your application behaves correctly with Appearance.

### UnregisterAppearanceClient

Use RegisterAppearanceClient to let the Appearance Manager know you are no longer using new Appearance defprocs and APIs.

```
pascal OSStatus UnregisterAppearanceClient(void)
```

DESCRIPTION

This routine should be called when you want to stop autorouting calls to the classic defprocs (WDEF 0, CDEF 0, etc.) to the new Appearance-Savvy defprocs automatically. This should not normally be called until your application terminates. The only exception would be around calls to plug-ins that might require the classic defprocs.

## Using Patterns and Colors

### SetThemePen

Use SetThemePen to set the foreground color to a specified pattern.

```
pascal OSStatus SetThemePen(ThemeBrush brush,
              SInt16 depth, Boolean colorDevice);
```

DESCRIPTION

SetThemePen simply sets the foreground pattern to the pattern specified in the brush parameter. You also pass the depth and a boolean indicating whether or not you are drawing on a color device. This information helps the Appearance manager know what exact color or pattern to use for the situation. This is typically used inside a DeviceLoop drawing procedure.

## SetThemeBackground

Use SetThemeBackground to set the background pattern of a window.

```
pascal OSStatus SetThemeBackground(ThemeBrush brush,
              SInt16 depth, Boolean colorDevice);
```

DESCRIPTION

SetThemeBackground simply sets the background pattern to the pattern specified in the brush parameter. You also pass the depth and a boolean indicating whether or not you are drawing on a color device. This information helps the Appearance manager know what exact color or pattern to use for the situation. This is typically used inside a DeviceLoop drawing procedure.

## SetThemeTextColor

Use SetThemeTextColor to set the foreground color for drawing text.

```
pascal OSStatus SetThemeTextColor(ThemeTextColor color,
              SInt16 depth, Boolean colorDevice);
```

DESCRIPTION

SetThemeTextColor sets the foreground color to the color specified in the color parameter for drawing text. You also pass the depth and a boolean indicating whether or not you are drawing on a color device. This information helps the Appearance manager know what exact color to use for the situation. This is typically used inside a DeviceLoop drawing procedure.

### SetThemeWindowBackground

Use SetThemeWindowBackground to set the background color of a window.

```
pascal OSStatus SetThemeWindowBackground(
                  WindowPtr window, ThemeBrush brush,
                  Boolean update)
```

DESCRIPTION

SetThemeWindowBackground is used to set the background color of a window. This is the actual content color that PaintOne will erase to when called, and can be different than the actual background color stored in the grafPort for the window. The color to use is passed in the brush parameter. If update is true, the window is erased and an update event is generated for the entire contents.

## Drawing Theme-Savvy Primitives

### DrawThemeWindowHeader

Call DrawThemeWindowHeader to draw the correct window header for the current theme.

```
pascal OSErr DrawThemeWindowHeader( const Rect* rect,
                                          ThemeDrawState
state );
```

DESCRIPTION

This routine will draw a window header which looks right for the current theme. The header is the same as that used in the Finder. The state parameter indicates which state to draw the header in.

### DrawThemeWindowListViewHeader

Call DrawThemeFinderListViewHeader to draw the correct window header for a list view for the current theme.

```
pascal OSErr DrawThemeFinderListViewHeader( const Rect*
rect,
                                                    ThemeDraw-
State state );
```

DESCRIPTION

This routine will draw a window header for a list view which looks right for the current theme. The header is the same as that used in the Finder. The state parameter indicates which state to draw the header in.

## DrawThemePlacard

Call DrawThemePlacard to draw a placard for the current theme.

```
pascal OSErr DrawThemePlacard( const Rect* rect,
                                        ThemeDrawState state
);
```

DESCRIPTION

This routine will draw a placard which looks right for the current theme. The state parameter indicates which state to draw the header in.

## DrawThemeModelessDialogFrame

Call DrawThemeModelessDialogFrame to draw the right frame for a modeless dialog for the current theme.

```
pascal OSErr DrawThemeModelessDialogFrame ( const Rect*
rect,
                                                    ThemeDraw-
State state );
```

DESCRIPTION

This routine will draw a modeless dialog frame which looks right for the current theme. The state parameter indicates which state to draw the frame in. This call is actually used by the Dialog Manager to draw appearance-savvy dialogs. It is provided for those developers which implement windows that act like dialogs without the use of the Dialog Manager.

### DrawThemeEditTextFrame

Call DrawThemeEditTextFrame to draw an edit text frame in the current theme.

```
pascal OSErr DrawThemeEditTextFrame( const Rect* rect,
                                                    ThemeDraw-
State state );
```

DESCRIPTION

This routine will draw an edit text frame which looks right for the current theme. The state parameter indicates which state to draw the frame in. The frame is can actually be outset from the rectangle you pass in. In practice, you would pass the bounding rectangle of your item. This routine would outset the appropriate amount as specified by the theme and draw the frame.

### DrawThemeListBoxFrame

Call DrawThemeListBoxFrame to draw an edit text frame in the current theme.

```
pascal OSErr DrawThemeListBoxFrame( const Rect* rect,
                                    ThemeDrawState state );
```

DESCRIPTION

This routine will draw a list box frame which looks right for the current theme. The state parameter indicates which state to draw the frame in. The frame is can actually be outset from the rectangle you pass in. In practice, you would pass the bounding rectangle of your item. This routine would outset the appropriate amount as specified by the theme and draw the frame.

### DrawThemeFocusRect

Call DrawThemeFocusRect to draw a rectangular generic focus ring around a rectangle.

```
pascal OSErr DrawThemeFocusRect( const Rect* rect,
                                 Boolean hasFocus );
```

DESCRIPTION

This routine will draw a generic focus ring which looks right for the current theme. The hasFocus parameter indicates whether to draw or erase the ring. The ring is actually outset from the rectangle you pass in. In practice, you would pass the bounding rectangle of your item. This routine would outset the appropriate amount as specified by the theme and draw the ring.

### DrawThemePrimaryGroup

Call DrawThemePrimaryGroup to draw the right frame for a primary group box.

```
pascal OSErr DrawThemePrimaryGroup( const Rect* rect,
                                    ThemeDraw-
State state );
```

DESCRIPTION

This routine will draw a primary group frame which looks right for the current theme. The state parameter indicates which state to draw the header in.

### DrawThemeSecondaryGroup

Call DrawThemeSecondaryGroup to draw the right frame for a secondary group box.

```
pascal OSErr DrawThemeSecondaryGroup( const Rect* rect,
                                      ThemeDraw-
```

```
State state );
```

DESCRIPTION

This routine will draw a secondary group frame which looks right for the current theme. The state parameter indicates which state to draw the header in.

### DrawThemeSeparator

Call DrawThemeSeparator to draw a visual separator for the current theme.

```
pascal OSErr DrawThemeSeparator( const Rect* rect,
                                                ThemeDraw-
State state );
```

DESCRIPTION

This routine will draw a visual separator which looks right for the current theme. The state parameter indicates which state to draw the header in. The orientation of the rect passed in determines whether the line is horizontal or vertical.

# Summary of the Appearance Manager

## Constants

```
enum {
  kThemeActiveDialogBackgroundBrush                 = 1,
  kThemeInactiveDialogBackgroundBrush               = 2,
  kThemeActiveAlertBackgroundBrush                  = 3,
  kThemeInactiveAlertBackgroundBrush                = 4,
  kThemeActiveModelessDialogBackgroundBrush         = 5,
  kThemeInactiveModelessDialogBackgroundBrush       = 6,
  kThemeActiveUtilityWindowBackgroundBrush          = 7,
  kThemeInactiveUtilityWindowBackgroundBrush        = 8,
```

```
  kThemeListViewSortColumnBackgroundBrush            = 9,
  kThemeListViewBackgroundBrush                      = 10,
  kThemeIconLabelBackgroundBrush                     = 11,
  kThemeListViewSeparatorBrush                       = 12,
  kThemeChasingArrowsBrush                           = 13,
  kThemeDragHiliteBrush                              = 14,
  kThemeDocumentWindowBackgroundBrush                = 15,
  kThemeFinderWindowBackgroundBrush                  = 16
};

typedef SInt16 ThemeBrush;

enum {
  kThemeActiveDialogTextColor                        = 1,
  kThemeInactiveDialogTextColor                      = 2,
  kThemeActiveAlertTextColor                         = 3,
  kThemeInactiveAlertTextColor                       = 4,
  kThemeActiveModelessDialogTextColor                = 5,
  kThemeInactiveModelessDialogTextColor              = 6,
  kThemeActiveWindowHeaderTextColor                  = 7,
  kThemeInactiveWindowHeaderTextColor                = 8,
  kThemeActivePlacardTextColor                       = 9,
  kThemeInactivePlacardTextColor                     = 10,
  kThemePressedPlacardTextColor                      = 11,
  kThemeActivePushButtonTextColor                    = 12,
  kThemeInactivePushButtonTextColor                  = 13,
  kThemePressedPushButtonTextColor                   = 14,
  kThemeActiveBevelButtonTextColor                   = 15,
  kThemeInactiveBevelButtonTextColor                 = 16,
  kThemePressedBevelButtonTextColor                  = 17,
  kThemeActivePopupButtonTextColor                   = 18,
  kThemeInactivePopupButtonTextColor                 = 19,
  kThemePressedPopupButtonTextColor                  = 20,
  kThemeIconLabelTextColor                           = 21,
  kThemeListViewTextColor                            = 22
```

```
};

typedef SInt16 ThemeTextColor;

/* States to draw primitives: disabled, active, and pressed (hilited)
*/
enum {
  kThemeStateDisabled= 0,
  kThemeStateActive= 1,
  kThemeStatePressed= 2
};

typedef UInt32 ThemeDrawState;
```

## Routines

### Getting Patterns and Colors

```
pascal OSStatus SetThemePen(ThemeBrush brush, SInt16 depth, Boolean
                                        colorDevice);

pascal OSStatus SetThemeBackground(ThemeBrush brush, SInt16 depth,
                                        Boolean colorDevice);

pascal OSStatus SetThemeTextColor(ThemeTextColor color, SInt16 depth,
                                        Boolean colorDevice);

pascal OSStatus SetThemeWindowBackground(WindowPtr window, ThemeBrush
                                        brush, Boolean update);
```

### Drawing Theme-Savvy Primitives

```
pascal OSErr DrawThemeWindowHeader( const Rect* rect,  ThemeDrawState
                                        state );


pascal OSErr DrawThemeFinderListViewHeader
                                      ( const Rect* rect,  Theme-
                                        DrawState state );


pascal OSErr DrawThemePlacard( const Rect* rect,  ThemeDrawState
                                        state );
```

```
pascal OSErr DrawThemeModelessDialogFrame
                                        ( const Rect* rect,  Theme-
                                          DrawState state );


pascal OSErr DrawThemeEditTextFrame( const Rect* rect,  ThemeDraw-
                                          State state );


pascal OSErr DrawThemeFocusRect( const Rect* rect,  Boolean hasFocus
                                          );


pascal OSErr DrawThemePrimaryGroup( const Rect* rect,  ThemeDrawState
                                          state );


pascal OSErr DrawThemeSecondaryGroup( const Rect* rect,  ThemeDraw-
                                          State state );


pascal OSErr DrawThemeSeparator( const Rect* rect,  ThemeDrawState
                                          state );
```