CONTROL MANAGER

Dan Oliver

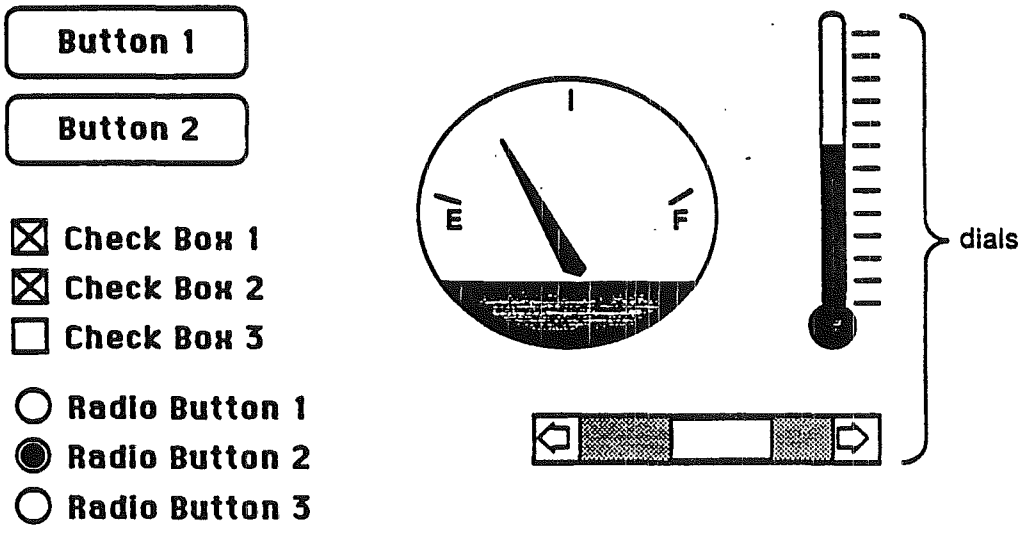
02/20/86	Initial release.
06/14/86	Revised release.
07/1586	Names changes; **BootCtrl, InitCtrlMgr, TermCtrlMgr**, to; **CtrlBootInit, CtrlStartup, CtrlShutDown.** Addition of **CtrlReset, CtrlStatus** and **GetCtrlzpage.**
07/16/86	Replacement for page 27, values for testCnlt and calcCRect were swapped.
08/13/86	Control record changed, CtrlAction added and CtrlHilite removed from CtrlFlag. The path to calling an action routine now includes CtrlAction. Color table colors corrected. **SetCMgrIcons** call added along with a CONTROL MANAGER ICON FONT section.

The Control Manager is the part of the Cortland User Interface Toolbox that deals with controls. A control is an object on the Cortland screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action. Using the Control Manager, your application can:

- display or hide controls
- monitor the user's operation of a control with the mouse and respond accordingly
- read or change the setting or other properties of a control
- change the size, location, or appearance of a control

Your application performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how.

Controls may be of various types, each with its own characteristic appearance on the screen and responses to the mouse. Each individual control has its own specific properties--such as its location, size, and setting--but controls of the same type behave in the same general way.



Certain standard types of controls are predefined for you. Your application can easily use controls of these standard types, and can also define its own "custom" control types. The predefined control types are the following:

- Buttons cause an immediate or continuous action when clicked or pressed with the mouse. They appear on the screen as rounded-corner rectangles with a title centered inside.

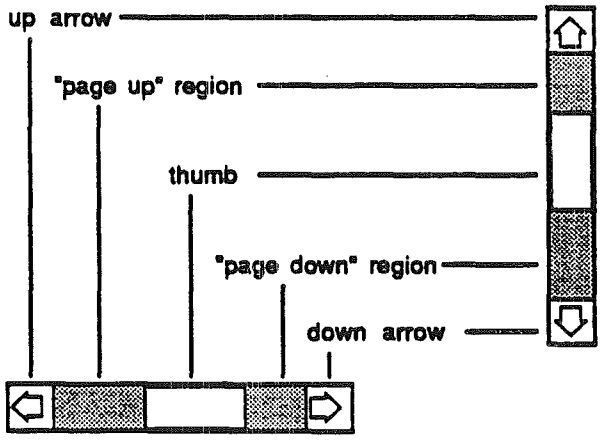
- Check boxes retain and display a setting, either checked (on) or unchecked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title to the left of it; the box is either filled in with an "X" (checked) or empty (unchecked). Check boxes are frequently used to control or midify some future action, instead of causing an immediate action of their own.

- Radio buttons also retain and display an on-or-off setting. They're organized into families, with the property that only one button in the family can be on at a time: clicking any button on turns off all the others in the family, like the buttons on a car radio. Radio buttons are used to offer a choice amoung several alternatives. On the screen, they look like round check boxes; the radio button that's on is filled with a small black circle instead of an "X".

  Note: The Control Manager knows which radio buttons belong in each family from the flag value passed to **NewControl**. Bits 8-14 of flag is the family number. Assign the same number for every member of a family, and different numbers for different families. Zero is an acceptable family number.

- Scroll bars are predefined dials. A dail displays a quantitative setting or value, typically in some pseudonanalog form such as the position of a sliding switch, the reading on a thermometer scale, or the angle of a needle on a gauage; the setting may be displayed digitally as well. The control's moving part that displays the current setting is called the indicator. The user may be able to change a dial's setting by dragging its indicator with the mouse, or the dial may simply display a value not under the user's direct control (such as the amount of free space remaining on a disk).

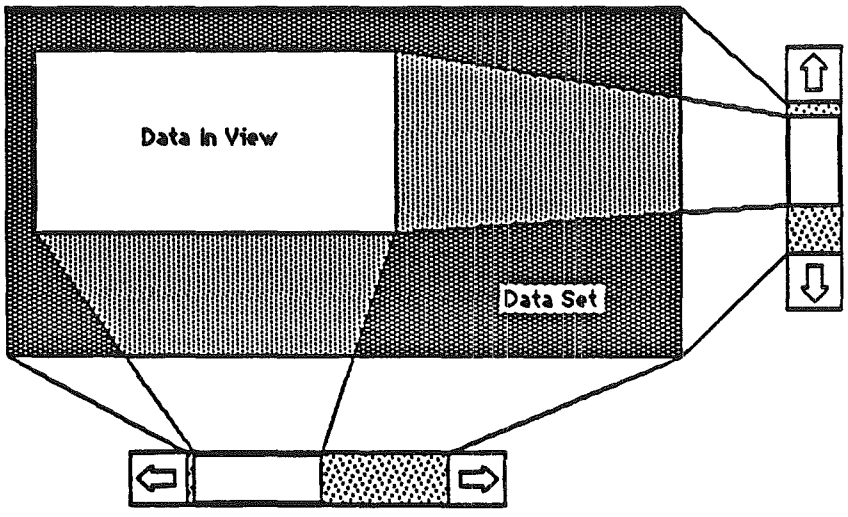
  The following diagram shows the parts of the vertical and horizontal scroll bars.



The parts of the scroll bars can be generalized into three regions; arrows, paging, and thumb (or thumber). The arrows scroll data a line at a time, paging regions scroll a "page" at a time, and the thumb can be dragged to any position within the scroll area. Although they may seem to behave like individual controls, these are all parts of a single control, the

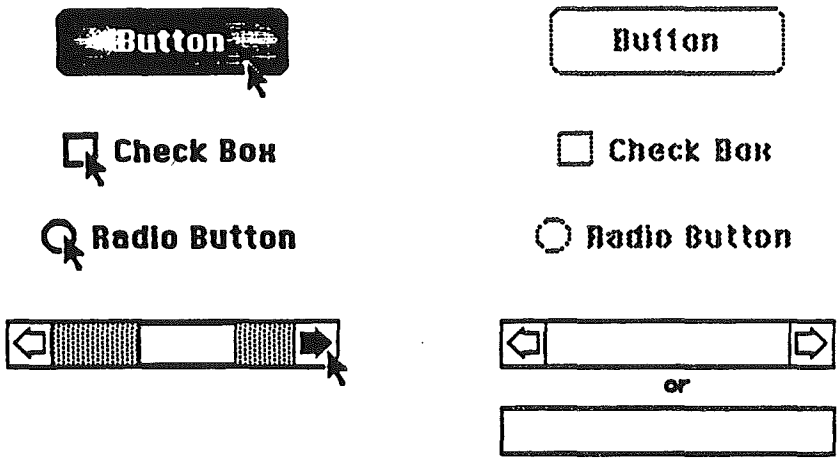
scroll bar type of dial. You can define other dials of any shape or complexity for yourself if your application needs them.

Standard scroll bars are porportional, that is they show the relationship between the total amount of data and the amount viewed, and where the view is in the data.



HIGHLIGHTING AND ACTIVE CONTROLS

When clicked or pressed, a control is usually highlighted. It's also possible for just a part of a control to be highlighted: for example, when the user presses the mouse button inside a scroll arrow in a scroll bar, the arrow, not the whole scroll bar, becomes highlighted.



Highlighted Active Controls Inactive Controls

A control may be active or inactive. Active controls respond to the user's mouse actions; inactive controls don't. A control is made inactive when it has no meaning or effect in the current context, such as an "Open" button when no document has been selected to open, or a scroll bar when there's currently nothing to scroll to. An inactive is highlighted in some special way, depending on its control type. For example, the title of an inactive button, check box, or radio button is dimmed.

There are two ways a scroll bar can be made inactive. In the diagram above the top scroll is made inactive by making the data size equal to or smaller than the view size. This type of inactive state happens automatically when the data size equals or exceeds the view size. The bottom inactive scroll bar is made by passing 255 to **HiliteControl** and is inactive in the same sense that the other controls are inactive.

There is one more way in which controls can be made inactive, make them invisible. Invisible controls are inactive in the sense that it can not be selected. However its 'highlighting in some special way' is an extreme.

CONTROLS AND WINDOWS

Every control "belongs" to a window: When displayed, the control appears within that window's content region; when manipulated with the mouse, it acts on that window. All coordinates pertaining to the control (such as those describing its location) are given in its window's local coordinate system. Even the state of the control can be tied to the state of the window. A bit in wFrame of the window's record can be set so the controls in the window will be considered inactive if the the window is inactive. See the Window Manager ERS.

Warning: In order for the Control Manager to draw a control properly, the control's window must have the top left corner of its grafPort's portRect as coordinates (0,0). If you change a window's local coordinate system for any any reason (with the QuickDraw procedure **SetOrigin**), be sure to change it back-so that the top left corner is again at (0,0)-before drawing any controls. Since almost all of the Control Manager routines can (at least potentially) redraw a control, the safest policy is simply to change the coordinate system back before calling any Control Manager routine.

However: If you would like to have controls in a window scroll with the content region there is a way. Before call the Control Manager make sure the origin of the control's window is set to its scrolled value

PART CODES

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar may have two scroll arrows, two paging regions, and a thumb. To allow different parts of a control to respond to the mouse in different ways, many of the Control Manager routines accept a part code as a parameter or return one as a result.

A part code is a number between 1 and 253 that stands for a particular part of a control. Each type of control has its set of part codes. Some of the Control Manager routines need to give special treatment to the indicator of a dial (such as the thumb of a scroll bar). To allow the Control Manager to recognize such indicators, they always have part codes greater than 127.

The part codes are assigned as follows:

0	No part.
1	Reserved for internal use.
2	Simple button.
3	Check box.
4	Radio button.
5	Up arrow.
6	Down arrow.
7	Page up.
8	Page down.
9	Reserved for internal use.
10	Grow box icon.
11-31	Reserved for internal use.
32-127	Reserved for application's use.
128	Reserved for internal use.
129	Thumb.
130-159	Reserved for internal use.
160-253	Reserved for application's use.
254-255	Reserved for internal use.

USING THE CONTROL MANAGER

This section discusses how the Control Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in CONTROL MANAGER ROUTINES, and examples are found in PROGRAMMING EXAMPLES.

To use the Control Manager, you must have previously called InitGraf to initialize QuickDraw and InitFonts to initialize the Font Manager if you are going to use controls with text in them.

> **Note:** For controls in dialogs or alerts, the Dialog Manager makes some of the basic Control Manager calls for you. Also, the Window Manager will make Control Manager calls concerning standard window controls.

Where appropriate in your program, use **NewControl** to add any controls you need. **NewControl** will set the control's owner to the window pointer passed and add the control to the head of the window's control list. When you no longer need a control, call **DisposeControl** to remove it from its window's control list and erase it from the screen. To dispose of all a window's controls at once, use **KillControls**.

> **Note:** The Window Manager procedure **CloseWindow** will automatically dispose of all the controls associated with the given window.

When the Event Manager function **GetNextEvent** reports that an update event has occurred for a window, the application should call **DrawControls** to redraw the window's controls as part of the process of updating the window.

After receiving a mouse-down event from **GetNextEvent**, do the following:

1. First call **FindWindow** to determine which part of which window the mouse button was pressed in. If it was in the content region of the active window, use that window's control list.

2. If the event did occur in a content area call **FindControl** with the pointer to the window to find out whether the event occurred on an active control.

3. Finally, if **FindControl** returns a control handle, call **TrackControl** to handle user interaction with the control. **TrackControl** will handle the highlighting of the control and determines whether the mouse is still in the control when the mouse button is released. It also handles the dragging of the thumb in a scroll bar and responds to presses or clicks in the other parts of a scroll bar. When **TrackControl** returns the part code for valid control, the application must do whatever is appropriate as a response.

The application's exact response to mouse activity in a control that retains a setting will depend on the current setting of the control, which is available from the **GetCtlValue** function. For controls whose values can be set by the user, the **SetCtlValue** procedure may be called to change the control's setting and redraw the control accordingly. You'll call **SetCtlValue**, for example, when a check box or radio button is clicked, to change the setting and draw or clear the mark inside the control.

Wherever needed in your program, you can call **HideControl** to make a control invisible or **ShowControl** to make it visible. Similarly, **MoveControl**, which simply changes a control's location without pulling around an outline of it, can be called at any time, as can **SizeControl**, which changes its size. For example, when the user changes the size of a document window that contains a scroll bar, you'll call **HideControl** to remove the old scroll bar, **MoveControl** and **SizeControl** to change its location and size, and **ShowControl** to display it as changed. Whenever necessary, you can read various attributes of a control with **GetCTitle**, **GetCtlMinMax**, or **GetCtlState**; you can change them with **SetCTitle**, **SetCtlMinMax**, or **SetCtlState**.

CONTROL MANAGER ICON FONT

The standard control definition procedures use a font to draw some control parts and their highlighted states. If you would like to use different icons you can replace the default font. To replace the icon font, or just get the handle to the current font, call **SetCMgrIcons**. The format of the font is as follows:

Character 0	Check box that is off and not highlighted.
Character 1	Check box that is off and is highlighted.
Character 2	Check box that is on and not highlighted.
Character 3	Check box that is on and not highlighted.
Character 4	Radio button that is off and not highlighted.
Character 5	Radio button that is off and is highlighted.
Character 6	Radio button that is on and not highlighted.
Character 7	Radio button that is on and is highlighted.
Character 8	Right arrow that is not highlighted.
Character 9	Right arrow that is highlighted.
Character 10	Left arrow that is not highlighted.
Character 11	Left arrow that is highlighted.
Character 12	Up arrow that is not highlighted.
Character 13	Up arrow that is highlighted.
Character 14	Down arrow that is not highlighted.
Character 15	Down arrow that is highlighted.
Character 16	Grow icon.

CONTROL RECORDS

Every control has the same front end to its control record. Additional data can then be appended to the end of the general control record. For example, **NewControl** will call the control's definition procedure to find out the size of the record to allocate, before the record is actually allocated. The **General Control Record** follows:

CtrlNext	LONG	Handle to next control, zero = last control.
CtrlOwner	LONG	Pointer to window the control belongs to.
CtrlRect	RECT	Enclosing rectangle.
CtrlFlag	BYTE	Flags that define the control, bit 7 = 0 if visible, 1 if invisible.
CtrlHilite	BYTE	Currently highlighted part.
CtrlValue	WORD	Current value.
CtrlProc	LONG	Address of control's definition procedure.
CtrlAction	LONG	Address of control's default action procedure.
CtrlData	LONG	Data used by definition procedure.
CtrlRefCon	LONG	Reserved for application's use only.
CtrlColor	LONG	Pointer to control's color table, zero for default table.

CtrlNext is a handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beinning in the wcontrol field of the window record and chained together through the CtrlNext fields of the individual control records. The end of the list is marked by a zero value; as new controls are created, they're added to the beginning of the list.

CtrlOwner is a pointer to the window port that this control belongs to.

CtrlRect is the rectangle that completely encloses the control, in the local coordinates of the control's window.

CtrlFlag is a bit vector that further describes the control. The only bit that is used for all controls is bit 7. Bit 7 is 0 if the control is visible, and 1 if invisible. Bits 0-6 are reserved for the control's definition procedure.

CtrlHilite specifies whether and how the control is to be highlighted, indicating whether it's active or inactive. The value is zero if the control is active and has no highlighted parts. The value is 255 if the control is inactive. If the value is between 1 and 254, it's the part code of a highlighted part of the control. Therefore, only one part on a control can be highlighted at any one time, and no part can be highlighted on an inactive control. See **HiliteControl** for more information.

CtrlValue is the control's current setting. For check boxes and radio buttons, 0 means the control is off and non-zero means it's on. For scroll bars, the value is between 0 and data size less view size. Custom controls can use the field as they see fit.

CtrlProc is the address of the control definition procedure for this type of control. Standard controls do not use an address in this field. Instead, bits 0-23 are zero and only the high-order byte is used to determine which standard control the control is. Values for standard controls are:

$00000000	Simple button.
$02000000	Check box.
$04000000	Radio button.
$06000000	Scroll bar.

CtrlData is reserved for use by the control definition procedure, typically to hold additional information specific to a particular control type. For example, the standard definition procedure for scroll bars uses the low-order word as view size, and the high-order word as data size. The standard definition procedures for simple buttons, check boxes, and radio buttons store the address of the control's title here.

CtrlAction is a address of the control's default action procedure, if any. The procedure **TrackControl** may call the default action procedure to respond to the user's draaging the mouse inside the control. See **TrackControl** for more information.

CtrlRefCon is the control's reference value field, which the application may store into and access for any purpose.

CtrlColor is a pointer to the control's color table, which is used by the control's definition procedure to draw the control.

More fields can be added to the end of the control record to further define the control. See the control record of a standard scroll bar as an example.

The following are how control record fields are used by standard controls:

Simple Button:

CtrlFlag, bit 0 = 1 for bold outline, 0 for single outline.

CtrlValue is always zero.

CtrlProc equals $00000000.

CtrlData is a pointer to the button's title string.

CtrlColor is a pointer to control's color table, or zero for default table. The simple button color table is defined as:

color1 = outline color when normal, in high nibble.
color2 = interior color when normal.
color3 = interior color when selected.
color4 = text color when normal.
color5 = text color when selected.
color6 = special highlight color.
color7 = thick outline color.

A simple button can be drawn with one of two outlines. The thick outline should be used with buttons that would be selected by the user pressing Return on the keyboard. This would be a default key and should never cause a the destruction of something, like a default button "Delete File". The thin outline should be used for all other simple buttons.

Check Box:

CtrlValue is 0 if not checked, non-zero if checked.

CtrlProc equals $02000000.

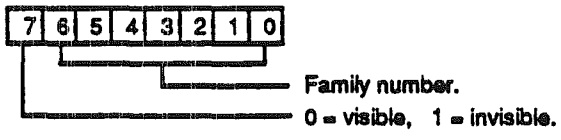
CtrlData is a pointer to the check boxe's title string.

CtrlColor is a pointer to control's color table, or zero for default table. The check box color table is defined as:

color1 = not used.
color2 = color of check box when not highlighted.
color3 = color of check box when highlighted.
color4 = color of title.

Radio Button:

CtrlFlag is:



CtrlValue is 0 if off, non-zero if on.

CtrlProc equals $04000000.

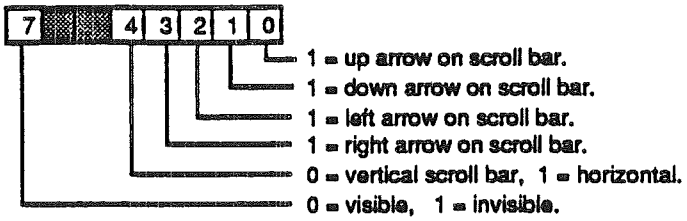
CtrlData is a pointer to the radio button's title string.

CtrlColor is a pointer to control's color table, or zero for default table. The radio button color table is defined as:

color1 = not used.
color2 = color of radio button when not highlighted.
color3 = color of radio button when highlighted.
color4 = color of title.

Scroll Bar:

CtrlFlag is defined as:



CtrlValue equals a number between zero and data size less view size.

CtrlProc equals $06000000.

CtrlData low-order WORD equals view size, and high-order word equals data size.

CtrlColor is a pointer to control's color table, or zero for default table. The radio button color table is defined as:

color1 = outline color.
color2 = arrow color when normal.
color3 = arrow color when selected.
color4 = arrow box interior color.
color5 = thumber interior color when normal.
color6 = thumber interior color when selected.
color7 = page region's color, low nibble = background.
color8 = inactive color.

Additional data fields appended to the end of the control record:

Thumb	RECT	Thumber rectangle.
PageRegion	RECT	Page region, thumb's bounds.

CONTROL MANAGER ROUTINES

INITIALIZATION AND TERMINATION

CtrlBootInit

input: None.

output: None.

Called only by the loader when loaded.

CtrlVersion

input: None.

output: Version:WORD Version number of the Control Manager.

CtrlReset

input: None.

output: None.

Called on system reset.

CtrlStatus

input: None.

output: status:WORD - TRUE if Control Manager is active, FASLE if not.

CtrlStartup

input: yourID:WORD — Your ID number, used for memory allocation.
zeroPage:WORD — Zero page Control Manager can use.

output: None.

InitCtrlMgr allows the Control Manager to perform startup initialization. YourID will be used by the Control Manager when it allocates memory. ZeroPage is an address of a page (256 bytes) in bank zero that your application makes available to the Control Manager for its use. The page does not have to be page aligned, but the Control Manager will operate faster if it is.

CtrlShutDown

input: None.

output: None.

Deactivates the Control Manager. No controls are disposed of, CloseWindow in the Window Manager disposes of all controls in a window. Therefore, the Control Manager should not be shutdown until after the Window Manager has been shutdown.

CtrlNewRes

input: None.

output: None.

Call CtrlNewRes after you have changed the video mode. This routine will reinitialize resolution and mode dependencies.

NewControl

input:	theWindow:LONG	Pointer to window owner.
	boundsRect:LONG	Pointer to enclosing RECT.
	title:LONG	Pointer to title string (CtrlData).
	flag:WORD	Bit vector of flags.
	value:WORD	Control's starting value.
	param1:WORD	Additional parameter (view size for scroll bars).
	param2:WORD	Additional parameter (date size for scroll bars).
	defProc:LONG	Address of definition procedure, or standard.
	refCon:LONG	Any value you want, application reserved.
	colorTable:LONG	Pointer to control's color table.
output:	ControlHandle:LONG	Control's handle, zero if error.

NewControl creates a control, adds it to the beginning of theWindow's control list, and returns a handle to the new control. The values passed as parameters are stored in the corresponding fields of the control record, as described below. The field that determines highlighting is set to 0 (no highlighting).

> **Note:** The control definition function may do additional initialization, including changing any of the fields of the control record. The only standard control for which additional initialization is done is the scroll bar; its control definition procedure computes the thumber and page region from boundsRect and flag.

TheWindow is the window the new control will belong to. All coordinates pertaining to the control will be interpreted in this window's local coordinate system.

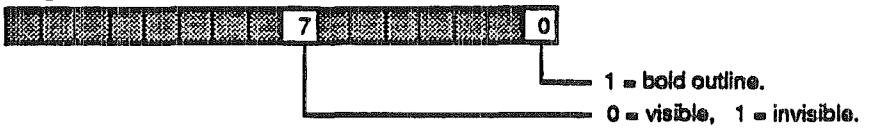
BoundsRect, given in theWindow's local coordinates, is the rectangle that encloses the control and thus determines its size and location. Note the following about the enclosing rectangle for the standard controls:

- Simple buttons are drawn to fit the rectangle exactly. (The control definition function calls the QuickDraw procedure FrameRoundRect.) To allow for the tallest characters in the system font, there should be at least a 20-point difference between the top and bottom coordinates of the rectangle.
- For check boxes and radio buttons, there should be at least a 16-point difference between the top and bottom coordinates.
- A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and thumb.

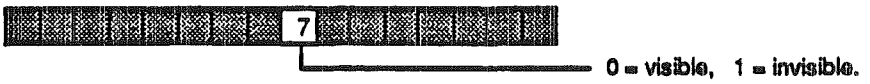
Title is the control's title, if any (if none, you can just pass the empty string as the title). Be sure the title will fit in the control's enclosing rectangle; if it won't it may not be completely erase with **HideControl**, along with other possible side effects.

Flag is a bit vector that further defines the control. Bit 7 is a visible/invisiable flag for every kind of controll. Bits 8-15 can be set to $FFxx to make the control inactive, but should be normally set to zero for an active control. Bits 0-6 are defined by each type of control. The bit vectors are defined below for standard controls.

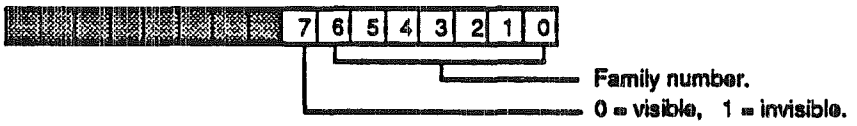
Simple button flag:



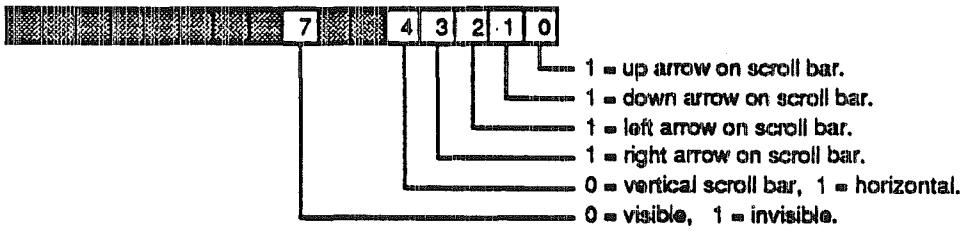
Check box flag:



Radio button flag:



Scroll bar flag:



Grow box flag:

The min and max parameters define the control's range of possible settings; the value parameter gives the initial setting. For controls that don't retain a setting, such as buttons, the values you supply for these parameters will be passed to the definition procedure, and it may or may not store them in the control's record. So it doesn't matter what values you give for those controls—0 for all three parameters will do. For dials, you can specify whatever values are appropriate for min, max, and value. For standard scroll bars the min is the size of the view, and max is the total data size. The standard scroll bar definition procedure will store the value of min in the CtrlData field, and max in CtrlData+2 field.

DefProc is the address of the control's definition procedure. DefProcs for custom control types are discussed later under "Defining Your Own Controls". The values for the standard control types are:

$00000000	- Simple button.
$02000000	- Check box.
$04000000	- Radio button.
$06000000	- Scroll bar.

RefCon is the control's reference value, set and used only by your application.

DisposeControl

input: theControl:LONG Handle of control.

output: None.

DisposeControl removes theControl from the screen, deletes it from its window's control list, and releases the memory occupied by the control record and any data structures associated with the control.

KillControls

input: theWindow:LONG Pointer to window.

output: None.

KillControls disposes of all controls associated with theWindow by calling DisposeControl (above) for each control in theWindow's control list.

Note: Remember that the Window Manager procedures CloseWindow automatically dispose of all controls associated with the given window.

CONTROL DISPLAY

These procedures affect the appearance of a control but not its size or location.

SetCTitle

input:	title:LONG	Address of new title.
	theControl:LONG	Handle of control.
output:	None.	

SetCTitle sets theControl's title to the given string and redraws the control.

GetCTitle

input:	theControl:LONG	Handle of control.
output:	title:LONG	Pointer to control's title.

GetCTitle returns the value in theControl's CtrlData field, which, for controls with titles, is the pointer to the control's title string.

HideControl

input:	theControl:LONG	Handle of control.
output:	None.	

HideControl makes theControl invisible. It fills the region the control occupies within its window with the background pattern of the window's grafPort. It also adds the control's enclosing rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, HideControl has no effect.

ShowControl

input:	theControl:LONG	Handle of control.
output:	None.	

ShowControl makes theControl visible. The control is drawn in its window but may be completely or partially obscured by overlapping windows or other objects. If the control is already visible, ShowControl has no effect.

DrawControls

input: theWindow:LONG — Pointer to window, of which the control list is drawn.

output: None.

DrawControls draws all controls currently visible in theWindow. The controls are drawn in reverse order of creation; thus in case of overlap the earliest-created controls appear frontmost in the window.

Note: Window Manager routines such as **SelectWindow**, **ShowWindow**, and **BringToFront** do not automatically call **DrawControls** to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event. Your program should always call **DrawControls** explicitly upon receiving an update event for a window that contains controls.

HiliteControl

input: hiliteState:WORD — Operation to perform.
theControl:LONG — Handle of control.

output: None.

HiliteControl changes the way theControl is highlighted. HiliteState has one of the following values:

- The value 0 means no highlighting and the control is active. Any highlighted part of the control is unhighlighted. If the control is inactive, it's changed to active and redrawn.
- A value between 1 and 253 is interpreted as a part code designating the part of the (active) control to be highlighted.
- The value 255 means that the control is to be made inactive and redrawn accordingly.

Note: The value 254 should not be used; this value is reserved for future use.

HiliteControl calls the control definition function to redraw the control with its new highlighting.

MOUSE LOCATION

FindControl

input:	FoundCtrl:LONG	Address of where to store control handle.
	xPoint:WORD	X coordinate, in global coordinates, to check.
	yPoint:WORD	Y coordinate, in global coordinates, to check.
	theWindow:LONG	Pointer of window to check.
output:	FoundPart:WORD	Part code of found part on control.

When the Window Manager function **FindWindow** reports that the mouse button was pressed in the content region of a window, and the window contains controls, the application should call **FindControl** with theWindow equal to the window pointer and thePoint equal to the point where the mouse button was pressed (in the window's global coordinates). **FindControl** tells which of the window's controls, if any, the mouse button was pressed in:

- If it was pressed in a visible, active control, **FindControl** sets the whichControl parameter to the control handle and returns a part code identifying the part of the control that it was pressed in.
- If it was pressed in an invisible or inactive control, or not in any control, **FindControl** sets whichControl to NIL and returns 0 as its result.

Note: **FindControl** also returns zero for whichControl and zero as its result if the window is invisible or doesn't contain the given point. In these cases, however, **FindWindow** wouldn't have returned this window in the first place, so the situation should never arise.

TestControl

input:	xPoint:WORD	X coordinate, in local coordinates, to check.
	yPoint:WORD	Y coordinate, in local coordinates, to check.
	theControl:LONG	Handle of control.
output:	PartCode:WORD	Part thePoint is over.

If theControl is visible and active, **TestControl** tests which part of the control contains thePoint (in the local coordinates of the control's window); it returns the corresponding part code, or zero if the point is outside the control. If the control is invisible or inactive, **TestControl** returns zero. **TestControl** is called by **FindControl** and **TrackControl**; normally you won't need to call it yourself.

TrackControl

input:	startX:WORD	X coordinate, in global coordinates, of starting point.
	startY:WORD	Y coordinate, in global coordinates, of starting point.
	actionProc:LONG	Address of routine, zero, or a negative number.
	theControl:LONG	Handle of control.
output:	PartCode:WORD	Selected part when button was released.

When the mouse button is pressed in a visible, active control, the application should call **TrackControl** with theControl equal to the control handle and startY and startX are equal to the point where the mouse button was pressed (in the global coordinates). **TrackControl** follows the movements of the mouse and responds in whatever way is appropriate until the mouse button is released; the exact response depends on the type of control and the part of the control in which the mouse button was pressed. If highlighting is appropriate, **TrackControl** does the highlighting, and undoes it before returning. When the mouse button is released, **TrackControl** returns with the part code if the mouse is in the same part of the control that it was originally in, or with zero if not (in which case the application should do nothing).

If the mouse button was pressed in an indicator, **TrackControl** drags a dotted outline of it to follow the mouse. When the mouse button is released, **TrackControl** calls the control definition procedure to reposition the control's indicator. The control definition function for scroll bars responds by redrawing the thumb, calculating the control's current setting based on the new relative position of the thumb, and storing the current setting in the control record. The application must then scroll to the corresponding relative position in the document.

TrackControl may take additional actions beyond highlighting the control or dragging the indicator, depending on the value passed in the actionProc parameter, as described below. The following tells you what to pass for the standard control types; for a custom control, what you pass will depend on how the control is defined.

- If actionProc is zero, **TrackControl** performs no additional actions. This is appropriate for simple buttons, check boxes, radio buttons, and the thumb of a scroll bar.

- ActionProc may be a pointer to an action procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button. (See below for details.)

- If actionProc is a negative number, **TrackControl** will check the CtrlAction field of the control's record. No additional actions will be performed if CtrlAction is zero. If CtrlAction is negative, the control's definition procedure will be called with an autoTrack message. If CtrlAction is neither zero or negative, it will be considered a valid address of an action routine and be called.

The action procedure in the control definition procedure is described in the section "Defining Your Own Controls". The action procedure should be of the form:

```
MyAction

     inputs:  partCode:WORD     Selected part.
              theControl:LONG   Handle of control.

     outputs: None.
```

In this case, **TrackControl** passes the control handle and the part code to the action procedure. (It passes zero in the partCode parameter if the mouse has moved outside the original control part.) As an example of this type of action procedure, consider what should happen when the mouse button is pressed in a scroll arrow or paging region in a scroll bar. For these cases, your action procedure should examine the part code to determine exactly where the mouse button was pressed, scroll up or down a line or page as appropriate, and call **SetCtlValue** to change the control's setting and redraw the thumb.

CONTROL MOVING AND SIZING

MoveControl

input:	NewX:WORD	New X origin of control.
	NewY:WORD	New Y origin of control.
	theControl:LONG	Handle of control.
output:	None.	

MoveControl moves theControl to a new location within its window. The top left corner of the control's enclosing rectangle is moved to the horizontal and vertical coordinates h and v (given in the local coordinates of the control's window); the bottom right corner is adjusted accordingly, to keep the size of the rectangle the same as before. If the control is currently visible, it's hidden and then redrawn at its new location.

DragControl

input:	startX:WORD	X coordinate, in local coordinates, of starting point.
	startY:WORD	Y coordinate, in local coordinates, of starting point.
	limitRect:LONG	Pointer to bounds rectangle.
	slopRect:LONG	Pointer to slop rectangle.
	axis:WORD	Movement constraint.
	theControl:LONG	Handle of control.
output:	None.	

Called with the mouse button down inside theControl, **DragControl** pulls a dotted outline of the control around the screen, following the movements of the mouse until the button is released. When the mouse button is released, **DragControl** calls **MoveControl** to move the control to the location to which it was dragged.

> Note: Before beginning to follow the mouse, **DragControl** calls the control definition function to allow it to do its own "custom dragging" if it chooses. If the definition function doesn't choose to do any custom dragging, **DragControl** uses the default method of dragging described here.

The startX, startY, limitRect, slopRect, and axis parameters have the same meaning as for the procedure **DragRect.**, see **DragRect.**

SizeControl *(not completed)*

input:	NewWidth:WORD	New width of control.
	NewHeight:WORD	New height of control.
	theControl:LONG	Handle of control.
output:	None.	

SizeControl changes the size of theControl's enclosing rectangle. The bottom right corner of the rectangle is adjusted to set the rectangle's width and height to the number of pixels specified by w and h; the position of the top left corner is not changed. If the control is currently visible, it's hidden and then redrawn in its new size.

CONTROL RECORD ACCESS

SetCtlValue

input:	CurValue:WORD	Current value of control.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlValue sets theControl's current setting to theValue and redraws the control to reflect the new setting. For check boxes and radio buttons, the value 1 fills the control with the appropriate mark, and zero clears it. For scroll bars, **SetCtlValue** redraws the thumb where appropriate.

If the specified value is out of range, it's forced to the nearest endpoint of the current range.

GetCtlValue

input:	theControl:LONG	Handle of control.
output:	CurValue:WORD	Control's current value.

GetCtlValue returns theControl's current setting.

SetCtlParams

input:	param2:WORD	Additional control parameter, defined by control.
	param1:WORD	Additional control parameter, defined by control.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlParams is a way of setting new parameters to the control's definition procedure, which will set the values and redraw the control if necessary. Simple buttons, check boxes, and radio buttons, do not use param1 or param2, and no action is performed.

Of the predefined controls, only scroll bars use the parameters. Param1 is used as the scroll bar's view, and param2 the data size. If, for either param1 or param2, a -1 is passed, that parameter will not be changed (this only applicable to predefined scroll bars, custom controls may not support this feature). Example:

> You want to show am editable text document, with a single vertical scroll bar to the right of the text. The text document has 300 lines, of which 30 can be displayed at one time. To set the scroll bar you would pass 30 for param1 and 300 for param2.
>
> If the user enters a line you would want to update the scroll bar. So, you pass -1 for param1 because there was no change in the view (although for predefined scroll bar there is no advantage to passing the view size again rather than -1), and 301 for param2 to show the increased data size.
>
> For this same document there is another approach you could take. You could pass the view and data sizes as pixels. If every line is 10 pixels high, counting leading, and there were 300 lines, of which 30 can be displayed, you would pass 300 for param1 and 3000 for param2. After the line was entered, you'd pass -1 for param1 (or 300 again), and 3010 for param2. Because passing the number of pixels, rather than the number of lines, is proportionally equivalent, the scroll bar will be identical for either method.

GetCtlParams

input:	theControl:LONG	Handle of control.
output:	params:LONG	param1 in high WORD, param2 in low WORD.

GetCtlParams returns theControl's additional parameter settings. See **SetCtlParams** for a description of param1 and param2.

Miscellaneous Routines

DragRect

input:	actionProc:LONG	Address of routine, zero, or a negative number.
	dragPattern:LONG	Address of pattern to use for drag outline.
	startX:WORD	X coordinate, in local coordinates, of starting point.
	startY:WORD	Y coordinate, in local coordinates, of starting point.
	dragRect:LONG	Pointer to rectangle to be dragged.
	limitRect:LONG	Pointer to bounds rectangle.
	slopRect:LONG	Pointer to slop rectangle.
	axis:WORD	Movement constraint.
output:	MoveDelta:LONG	Low WORD is the amount Y changed, High WORD is the amount X changed.

DragRect pulls a dotted outline of dragRect around the screen, following the movements of the mouse until the button is released.

- StartY and startX are assumed to be the point where the mouse button was originally pressed, in the local coordinates of the current port.
- LimitRect limits the travel of the control's outline, and should normally coincide with or be contained within the current port.
- SlopRect allows the user some "slop" in moving the mouse; it should completely enclose limitRect. SlopRect is the limit of mouse movement before the drag outline is snapped back to its starting position. While the cursor is outside of slopRect the drag outline will be at its starting position.
- The axis parameter allows you to constrain the control's motion to only one axis. It has one of the following values:

```
CONST   noConstraint = 0    No constraint.
        hAxisOnly    = 1    Horizontal axis only.
        vAxisOnly    = 2    Vertical axis only.
```

GetCtrlzpage

input: None.

output: CtrlZPage:WORD - Control Manger's direct (zero) page.

This call will normally only be made by the Dialog Manager. The Dialog Manager makes this call because the Control and Dialog Managers share a single direct page.

SetCMgrIcons

input: newFont:LONG - handle of new icon font, negative to not set new font.

output: oldFont:LONG - handle of current icon font (before newFont is set).

See CONTROL MANAGER ICON FONT for more information about the icon font.

DEFINING YOUR OWN CONTROLS

In addition to predefined controls, you can also define "custom" controls of your own. Maybe you need a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator--whatever your application needs. Controls and their indicators may occupy regions of any shape.

To define your own type of control, you write a control definition procedure in your application. The Control Manager stores this address in the CtrlProc field of the control record. Later, when it needs to perform a type-dependent action on the control, it calls the control definition procedure.

The Control Definition Procedure

The inputs and output of the definition procedure are:

input:	message:WORD	Desired operation.
	param:LONG	Depends on operation.
	theControl:LONG	Handle of control.
output:	RetValue:LONG	Depends on operation.

The message parameter identifies the desired operation. It has one of the following values:

drawCntl	= 0	Draw the control (or control part).
calcCRect	= 1	Compute the rectangle to drag.
testCntl	= 2	Test where mouse button was pressed.
initCntl	= 3	Do any additional control initialization.
dispCntl	= 4	Take any additional disposal actions.
posCntl	= 5	Move the control's indicator.
thumbCntl	= 6	Compute the parameters for dragging an indicator.
dragCntl	= 7	Drag either a control's indicator, or the whole control.
autoTrack	= 8	Called while dragging if -1 passed to **TrackControl**.
newValue	= 9	Called when control gets new value.
setParams	= 10	Called when control gets new additional parameters.
moveCntl	= 11	Called control moves, compute new position for parts.
recSize	= 12	Return record size of control (in bytes).

As described below in the discussions of the routines that perform these operations, the value passed for param, depends on the operation. Similarly, the control definition procedure is expected to return a function result only where indicated; in other cases, the function should return zero.

The Draw Routine

message = drawCntl.
param = part code - draw part.
= zero - draw entire control.
(Only the low WORD is used, high WORD is undefined.)
RetValue = undefined.

The message drawCntl asks the control definition function to draw all or part of the control within its enclosing rectangle. The low-order WORD of param is a part code specifying which part of the control to draw, or zero for the entire control. If the control is invisible, there's nothing to do; if it's visible, the definition procedure should draw it (or the requested part), taking into account the current highlighting and value.

If param is the part code of the control's indicator, the draw routine can assume that the indicator hasn't moved; it might be called, for example, to highlight the indicator.

The Test Routine

message = testCntl.
param = low-order WORD = y point to check, in window's local coordinates.
= high-order WORD = x point to check, in window's local coordinates.
RetValue = undefined.

The Control Manager function **TestControl** sends the message testCntl to the control definition function when the mouse button is pressed in a visible control. This message asks in which part of the control, if any, a given point lies. The point is passed as the value of param, in the local coordinates of the control's window; the vertical coordinate is in the low-order word of the long integer and the horizontal coordinate is in the high-order word. The control definition function should return the part code for the part of the control that contains the point; it should return zero if the point is outside the control or if the control is inactive.

The Routine to Calculate Indicator Rectangle

message = calcCRect.
param = address of RECT.
RetValue = zero for default RECT, nonzero if RECT is set.

Just before the Control Manager starts to drag a control, or its indicator, it will call the control's definition procedure to determine the coordinates of the control, or its indicator. The highest bit of param will be clear if the whole control is to be dragged, or set if its indicator is to be dragged.

If the definition procedure returns zero, and the whole control is to be dragged, the RECT is set to the control's enclosing rectangle. If the definition procedure returns zero, and the control's indicator is to be dragged, the RECT is set to the thumb rectangle (see Scroll Bar Control Record) .

The Initialize Routine

message = initCntl.
param = low-order WORD is the param1 value passed to NewControl.
= high-order WORD is the param2 value passed to NewControl.
RetValue = undefined.

After allocating and initializing the control record as appropriate when creating a new control, the Control Manager sends the message initCntl to the control definition procedure. This gives the definition procedure a chance to perform any type-specific initialization it may require. For example, the control definition procedure for scroll bars initializes the thumb and page RECTs, and also stores param1 and param2 in the CtrlData field. The initialize routine for standard buttons, check boxes, and radio buttons does nothing.

The Dispose Routine

message = dispCntl.
param = undefined.
RetValue = zero to continue disposal, nonzero to abort disposal.

The Control Manager's DisposeControl procedure sends the message dispCntl to the control definition function, telling it to carry out any additional actions required when disposing of the control. The predefined controls always return zero. If the definition procedure returns zero for RetValue, the control will be erased, taken out of the control list, and its record deallocated.

By returning a nonzero number for RetValue, the definition procedure has a chance to abort the disposal. This feature is provided even though I am unable to provide an example of when this feature might be useful.

The Position Routine

message = posCntl.
param = low-order WORD is the vertical offset (delta y).
high-order WORD is the horizontal offset (delta x)
RetValue = zero for default reposition, nonzero if reposition completed.

When dragging a control's indicator to completed, **TrackControl** calls the control definition procedure with the message posCntl to reposition the indicator and update the control's setting accordingly. The value of param is a point giving the vertical and horizontal offset, in pixels, by which the indicator is to be moved relative to its current position. (Typically, this is the offset between the points where the user pressed and released the mouse button while dragging the indicator.) The vertical offset is given in the low-order word of param and the horizontal offset in the high-order word. The definition procedure should calculate the control's new setting based on the given offset, update the CtrlValue field, and redraw the control within its window to reflect the new setting.

Note: The Control Manager procedures **SetCtlValue** and **SetCtlParams** do not call the control definition procedure with this message; instead, they pass the newValue and setParams message (see below).

The Thumb Routine

message = thumbCntl.
param = pointer to parameter block for dragging an indicator.
RetValue = zero for default reposition, nonzero if reposition completed.

Before the Control Manger begins to drag a control's indicator, it will call the control's definition procedure with the message thumbCntl. The control definition procedure should respond by calculating the limiting rectangle, slop rectangle, axis constraint, and outline pattern to use for dragging the control's indicator. Param is a pointer to the following data structure:

```
limit_blk
   bound_rect:RECT    Limit of drag (not a pointer).
   slop_rect:RECT     Limit of cursor (not a pointer).
   axis_param:WORD    Movement constrain.
   drag_patt:LONG     Pointer to pattern for drag outline.
```

If the definition procedure returns zero, default parameters will be used. The defaults are computed thus:

`bound_rect`	PageRegion (see "Scroll Bar Control Record").
`slop_rect`	PageRegion plus 16 all around.
`axis_param`	2 if bit 12 of CtrlFlag is clear, 1 if set.
`drag_patt`	Pattern generated from color7 in control's color table.

See **DragRect** for more information about the parameters in the limit_blk. The parameters in limit_blk will be passed to **DragRect**.

The Drag Routine

message = dragCntl.
param = part code to drag, zero to drag the entire control.
RetValue = zero to use default dragging, nonzero if dragging is completed.

The message dragCntl asks the control definition procedure to drag the control or its indicator around on the screen to follow the mouse until the user releases the mouse button. Param specifies whether to drag a part or the whole control: zero means drag the whole control, while a nonzero value is the part code of the control part to drag.

The control definition procedure need not implement any form of "custom dragging"; if it returns a result of zero, the Control Manager will use its own default method of dragging (calling **DragControl** to drag the control or **DragRect** to drag its indicator). Conversely, if the control definition procedure chooses to do its own custom dragging, it should signal the Control Manager not to use the default method by returning a nonzero result.

If the whole control is being dragged, the definition function should call **MoveControl** to reposition the control to its new location after the user releases the mouse button. If just the indicator is being dragged, the definition function should execute its own position routine (see below) to update the control's setting and redraw it in its window.

The Track Routine

message = autoTrack.
param = part code, zero if not currently in part.
RetValue = undefined.

You can design a control to have its action procedure in the control definition procedure. To do this, pass -1 for actionProc parameter to **TrackControl**. **TrackControl** will respond by calling the control definition procedure with the message autoTrack. The definition function should respond like an action procedure, as discussed in detail in the description of **TrackControl**. It can tell which part of the control the mouse button was pressed in from param, which contains the part code. The track routine for each of the standard control types does nothing.

The New Value Routine

message = newValue.
param = undefined.
RetValue = number of bytes needed for control's record.

The Control Manager will call the control's definition procedure with the message newValue anytime a control's value changes. First, the Control Manager will store the new value in the CtrlValue field of the control's record. The definition should compute any new parameters affected by the change, like a new thumb position for scroll bars, and then redraw the control (if visible). The definition procedure can assume that control is already drawn is the window, so, in the case of scroll bars, only the thumb has to be erased, and redrawn. Actually, the definition procedure for standard scroll bars only erases the part of the thumb that uncovered the page region, rather than the entire thumb.

The New Parameters Routine

message = setParams.
param = new parameters.
RetValue = undefined.

The Control Manager will call the control's definition procedure with the message setParams anytime a control's additional parameters change. The term 'additional parameters' is defined by the control. The values could be anything, even a pointer to more parameters. The definition should the perform necessary actions the new parameters cause, including redrawing the control if needed. The definition procedure can assume that control is already drawn is the window, unlike when new parameters are sent with the message initCntl (see "The Initialize Routine").

The only predefined control that uses additional parameters is the scroll bar. The low-order WORD is the view value, and the high-order WORD is the data size. Simple buttons, check boxes, and radio button do nothing with addition parameters. The standard scroll bar definiton procedure will store the values in the CtrlData field of the control's record, compute a new thumb, and draw the new thumb in the scroll bar (if visible).

The Move Routine

message = moveCntl.
param = low-order WORD is the change in the vertical axis (delta y),
high-order WORD is the change in the horizontal axis (delta x).
RetValue = undefined.

The Control Manager will call the control's definition procedure with the message moveCntl from **MoveControl**. The Control Manager will first hide the control, with **HideControl**, if it was visible and move the control's enclosing rectangle (CtrlRect field). The definition procedure should compute any other parameters necessary and return. For example, the standard definition procedure for scroll bars will also move the Thumb and PageRegion fields in the control record. Upon return, the Control Manager will do a **ShowControl** if the control was visible on entry, to draw the control at its new position. The definition procedure should not redraw the control here, but should do everything necessary to ensure the control will be drawn properly at its new position.

The Record Size Routine

message = recCntl.
param = undefined.
RetValue = number of bytes needed for control's record.

The Control Manager call the control's definition procedure the message recCntl from **NewControl** before it allocates memory for the control's record. **NewControl** will then allocate how ever many bytes is returned in RetValue for the control's record.

If your control only needs the standard control record, like buttons, check boxes, and radio buttons, return the size of the standard record. If your control needs additional data fields, like a scroll bar, return the size of the standard record, plus the additional size. You should never return a number less than the number of bytes in a standard record.

Note: TheControl, the handle of the control, passed to the definition procedure is not valid in this case. Because the control's record has not been allocated, no access to the record should be performed during this call. After the record has been allocated and initialized by the Control Manager, the definition procedure will be called again with the message initCntl, see "The Initialize Routine" below.

Constants

```
NoPart          0
SimpleButt      2
CheckBox        3
RadioButt       4
UpArrow         5
DownArrow       6
PageUp          7
PageDown        8
GrowBox         10
Thumb           129

SimpleProc      $00000000
CheckProc       $02000000
RadioProc       $04000000
ScrollProc      $06000000

CTRL_VIS        $0080
UP_FLAG         $0001
DOWN_FLAG       $0002
LEFT_FLAG       $0004
RIGHT_FLAG      $0008
DIR_SCROLL      $0010
FAMILY          $007F
BOLD_BUTT       $0001
```

noConstraint	0	No constraint on movement.
hAxisOnly	1	Horizontal axis only.
vAxisOnly	2	Vertical axis only.
drawCtrl	0	Draw control command.
calcCRect	1	Compute drag RECT command.
testCtrl	2	Hit test command.
initCtrl	3	Initialize command.
dispCtrl	4	Dispose command.
posCtrl	5	Move indicator command.
thumbCtrl	6	Compute drag parameters command.
dragCtrl	7	Drag command.
autoTrack	8	Action command.
newValue	9	Set new value command.
setParams	10	Set new parameters command.
moveCtrl	11	Move command.
recSize	12	Return record size command.

Data Types

CtrlNext	0	Handle	Handle of next control.
CtrlOwner	4	Pointer	Pointer to control's window.
CtrlRect	8	RECT	Enclosing rectangle.
CtrlFlag	16	Byte	Bit flags.
CtrlHilite	17	Byte	Highlighted part.
CtrlValue	18	Integer	Control's value.
CtrlProc	20	Pointer	Control's definition procedure.
CtrlAction	24	Pointer	Control's action procedure.
CtrlData	28	LongInt	Reserved for CtrlProc's use.
CtrlRefCon	32	LongInt	Reserved for application's use.
CtrlColor	36	Pointer	Control's color table.
color1	0	Integer	
color2	2	Integer	
color3	4	Integer	
color4	6	Integer	
color5	8	Integer	
color6	10	Integer	
color7	12	Integer	
bound_rect	0	RECT	Drag bounds.
slop_rect	8	RECT	Cursor bounds.
axis_param	16	Integer	Movement constrains.
drag_patt	18	Pointer	Pattern for drag outline.