CONTROL MANAGER

Dan Oliver

Initial release 02/20/86

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 alongside 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.

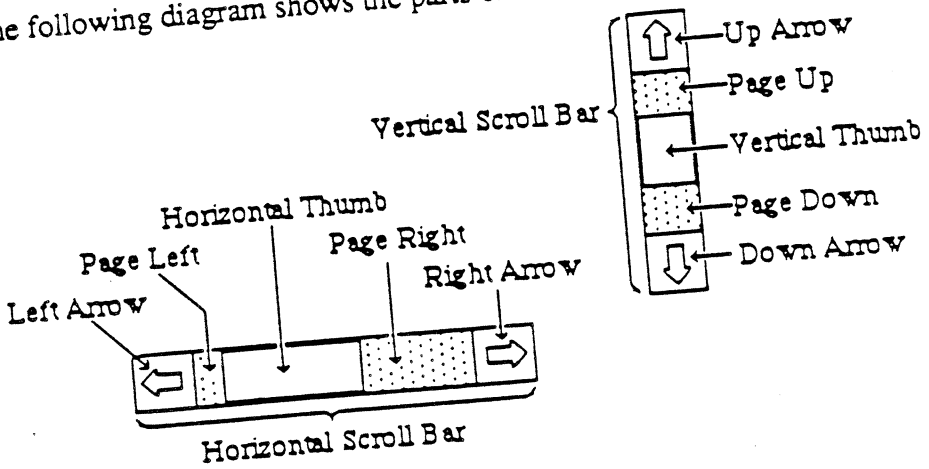
- Radio buttons also retain and display an on-or-off setting. They're organized into groups with the property that only one button in the group can be on at a time: clicking any button on turns off all the others in the group, like the buttons on a car radio. Radio buttons are used to offer a choice 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 doesn't know how radio buttons are grouped, and doesn't automatically turn one off when the user clicks another one on: it's up to your program to handle this.

- 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

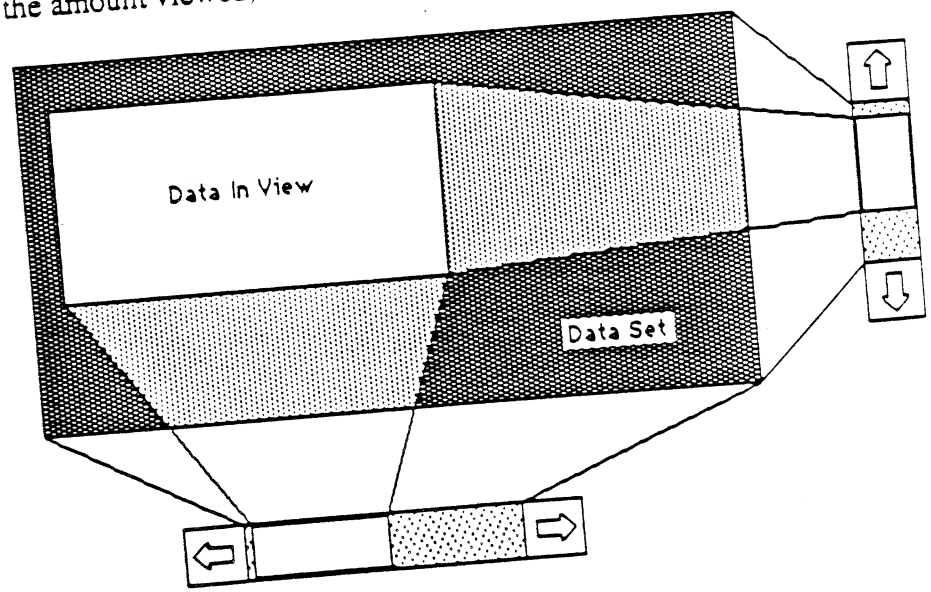
mouse, or the dial may simply display a value not under the user's direct control (such as the amount of free space remaining on a disk).

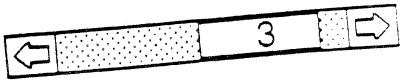
The following diagram shows the parts of the vertical and horizontal scroll bars.



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.

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.





When clicked or pressed, a control is usually highlighted. Standard button controls are inverted, but some control types may use other forms of highlighting, such as making the outline heavier. 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 or the thumb in a scroll bar, the arrow or the thumb (not the whole scroll bar) becomes highlighted.

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 control remains visible, but 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.

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.

However, even though controls belong to windows, it is not necessary for the Window Manager to be installed. Your application can create a window record to pass when adding a control to a control list. The Control Manager needs the window record as a place for the head of the control list and the window's origin is used with the control's position to come up with the screen position. This feature is included for applications that do not need windows, but would like to use controls without losing memory the Window Manager would take up.

PART CODES

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar has 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 0 and 255 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 for the predefined controls are as follows:

0 = No part.
10 = Simple button.
11 = Check box.
12 = Radio button.
20 = Up arrow in vertical scroll bar.
21 = Down arrow in vertical scroll bar.
22 = Left arrow in horizontal scroll bar.
23 = Right arrow in horizontal scroll bar.
24 = Page up in vertical scroll bar.
25 = Page down in vertical scroll bar.
26 = Page left in horizontal scroll bar.
27 = Page right in horizontal scroll bar.
128 = Reserved.
129 = Reserved.
130 = Thumb in vertical scroll bar.
131 = Thumb in horizontal scroll bar.
254 = Reserved.
255 = Reserved.

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.

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 procedures **DisposeWindow** and **CloseWindow** automatically dispose of all the controls associated with the given window.

When the Toolbox 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. If you are using windows, 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 first control in the list to find out whether the event occurred on an active control.

3. Finally, if **FindControl** returns a pointer to a control, 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, via your action procedure, the response to presses or clicks in the other parts of a scroll bar. When **TrackControl** returns the part code for a button, check box, or radio button, the application must do whatever is appropriate as a response to a click of that control. When **TrackControl** returns the part code for the thumb of a scroll bar, the application should scroll to the corresponding relative position in the document.

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 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. The **General Control Record** follows:

NextCtrl	LONG	Pointer to next control, zero = last control.
CtrlFlag	BYTE	Bit 7 1 = active. 0 = inactive (dimmed).
	Bit 6	1 = visible. 0 = invisible.
	Bit 5	1 = highlighted (selected). 0 = normal.
	Bits 4-0	control ID.
CtrlOwner	LONG	Pointer to window this control belongs to.
CtrlRect	RECT	Enclosing rectangle.

The following are predefined records:

Simple Button Control Record:

NextCtrl	LONG	Pointer to next control, zero = last control.
CtrlFlag	BYTE	Bits 4-0, 0 = thick outline, 1 = thin outline.
CtrlOwner	LONG	Pointer to window this control belongs to.
CtrlRect	RECT	Button coordinates.
CtrlNColor	BYTE	Normal color. Low nibble = color of text and button outline. High nibble = button's interior color.
CtrlIColor	BYTE	Inverted color. Low nibble = color of text when selected. High nibble = button's interior color when selected. Low = High to use special highlight.
CtrlTitle	STRG	Button's title.

A simple button can be drawn with one of two outlines and has two ways of being highlighted.

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.

The special highlight feature is provided for buttons that may be just a solid color. When a button is just a color it can't be inverted, or its color changed to highlight, and still show what it represents. So, a alternate method is provided, and should only be used in these cases.

Check Box Control Record:

NextCtrl	LONG	Pointer to next control, zero = last cor
CtrlFlag	BYTE	Bits 4-0 = 1, check box control ID.
CtrlOwner	LONG	Pointer to window this control belongs
CtrlRect	RECT	Check box coordinates.
CtrlNColor	BYTE	Normal color. Low nibble = color of tex High nibble = button's int
CtrlIColor	BYTE	Inverted color. Low nibble = color of X. High nibble = button's inte
CtrlTitle	STRG	Title to right of check box.

Radio Button Control Record:

NextCtrl	LONG	Pointer to next control, zero = last control.
CtrlFlag	BYTE	Bits 4-0 = 2, radio button control ID.
CtrlOwner	LONG	Pointer to window this control belongs to.
CtrlRect	RECT	Button coordinates.
CtrlNColor	BYTE	Normal color. Low nibble = color of text and High nibble = button's interior
CtrlIColor	BYTE	Inverted color. High nibble = button's interior c
CtrlTitle	STRG	Title to right of check box.

Scroll Bar Control Record:

NextCtrl	LONG	Pointer to next control, zero = last control.
CtrlFlag	BYTE	Bits 4-0, 3 = horizonal scroll bar, 4 = vertical scroll bar.
CtrlOwner	LONG	Pointer to window this control belongs to.
CtrlRect	RECT	Coordinates of scroll bar.
Thumb	RECT	Coordinates of thumber box.
ScrollCur	WORD	Current value.
ScrollMin	WORD	Minimum value.
ScrollMax	WORD	Maximum value.
SBarColor	WORD	High BYTE = pattern: 0 = solid. 1 = dither. 2 = dotted. Low BYTE = color: High nibble = pattern color. Low nibble = color of background.
ThumbColor	BYTE	High NIBBLE = interior of thumber box when normal. Low NIBBLE = interior of thumber box when inverted.
ArrowColor	BYTE	High NIBBLE = interior color of arrow box. Low NIBBLE = interior of arrow when inverted.

INITIALIZATION AND TERMINATION

InitCtrlMgr Call # (not completed)

input: None.

output: None.

function: The Control Manager doesn't need to keep very much variable an entire zero page will not be needed. The Control Manager w allocate a relocatable block of memory for its needs.

TermCtrlMgr Call # (not completed)

input: None.

output: None.

function: Frees any allocated memory.

NewControl Call # (not completed)

input: theWindow:LONG Pointer to window owner.
theControl:LONG Pointer to control.

output: None.

function: Adds control to head of window's control list. If the control is visible it will be drawn in theWindow. **NewControl** will set NextCtrl, CtrlOwner, and Thumb for scroll bars. It's important that the control list pointer in the window record is zero when **NewControl** is called for the first control of theWindow.

Note: The control is not drawn using the standard window updating mechanism, but instead is drawn immediately in the window.

Call # (not completed)

input: theControl:LONG Pointer to control.

output: None.

function: Calls **HideControl** to erases control from screen and of the window's control list.

KillControl

Call # (not completed)

input: theWindow:LONG Pointer to window.

output: None.

function: Calls **DisposeControl** for every control in theWindow's cont

SetCTitle

input: title:LONG — Address of new title.
theControl:LONG — Pointer to control.
Call # (not completed)

output: None.

function: Sets theControl's title to the given string and redraws th

GetCTitle

input: title:LONG — Address of where to put title.
theControl:LONG — Pointer to control.
Call # (not completed)

output: None.

function: TheControl's title is moved into title.

HideControl

input: theControl:LONG — Pointer to control.
Call # (not completed)

output: None.

function: Makes theControl invisible. It fills the control's CtrlRect with its window's content background pattern and color and adds CtrlRect to the window's update region. If the control is already invisible, **HideControl** has no effect.

ShowControl

input: theControl:LONG — Pointer to control.
Call # (not completed)

output: None.

function: 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

Call # (not

input: drawNum:WORD — Num
theControl:LONG — Poin

output: None.

function: Draws any number of controls,
you want every control in the w
the first control as theControl a
are drawn in reverse order of cr
earliest-created controls appear

Note: Window Manager routines such a
and **BringToFront** do not automaticall
window's controls. They just add the ap
update region, generating an update ever
DrawControls explicitly upon receivin
contains controls.

HiliteControl

Call # (not c

input: hiliteState:WORD — Opera
theControl:LONG — Pointe

output: None.

function: Control part is redrawn using hili

High BYTE: $00 = unhig
$01 = highl
Low BYTE: part number.

ControlState

Call # (not co

input: CtrlState:WORD — TRUE
FALSE

theControl:LONG — Pointer

output: None.

function: Control is redrawn in CtrlState.

MOUSE LOCATION

FindControl Call # (not completec

input: thePoint:LONG — Point, in local
theControl:LONG — Pointer to list.

output: FoundCtrl:LONG — Pointer to cont

function: Find the control, if any, thePoint is on. T
control found, or zero if no control is fou

Because the control list is searched from
address of a control so you have more tha
FindControl will always return the last
controls that overlap. However, if you fc
first control found you could continue the
control as theControl.

TestControl Call # (not completed

input: thePoint:LONG — Point, in local c
theControl:LONG — Pointer to contr

output: PartCode:WORD — Part thePoint is

function: Find what part of a control thePoint is on.
coordinates. PartCode is:

Zero if thePoint is not on control.
High BYTE = $00 if control is activ
= $FF if control is inac
Low BYTE = part number.

TrackControl Call # (not completed)

input:
StartPt:LONG Starting position of mouse.
StartState:WORD Starting button state, 1 = down.
Action:LONG Address of routine, or zero.
theControl:LONG Pointer to control.

output:
PartCode:WORD Selected part when button was released.

function:
If StartPt is over a selectable part of theControl, and the control is active, the part is highlighted. The mouse's position is then tracked until the button is released (or pressed and then released). While tracking the mouse **TrackControl** will inform the user graphically as appropriate for the control. When the button is relased the part will be unhighlighted if it was highlighted, and return the part number affected. If the button was released outside of the original part, zero will be returned.

StartState is provided to support hot controls. Here are the possibilities:

StartState	Mouse Tracked Until
$0001	Button is released, used for all predefined controls.
$0000	Button is down and then released, button may be down when called.
$FFFF	Mouse leaves part.

If Action is nonzero, it is the address of a routine in your application that will be repeatedly called while the original part is selected. Your routine will be called as quickly as possible, there is no delay factor other than to determine the part is still selected. This function is useful when the user selects an arrow on a scroll bar and keeps the button pressed until desired data scrolls into view.

Input to your routine:

PartCode:WORD Part number selected.
theControl:LONG Pointer to the control.

No output.

MOVING AND SIZING

MoveControl

Call # (not completed)

input: NewX:WORD New X origin of control.
NewY:WORD New Y origin of control.
theControl:LONG Pointer to control.

output: None.

function: The control is erase from the screen if visible and redrawn at position. NewY and NewY are given in local coordinates and the new top and left side of CtrlRect, the bottom and right sid changed to keep the same height and width of CtrlRect.

DragControl

Call # (not completed)

input: StartPt:LONG Starting position of mouse.
Axis:WORD Constraining axis.
Limit:RECT Limiting area of movement.
theControl:LONG Pointer to control

output: None.

function: Draws an XOR frame of CtrlRect and moves the box as the mouse moves until the button is released, or pressed and released, and then calls **MoveControl**. The box will only be drawn inside Limit and only move according to Axis. Axis can be:

$0000 No constraint.
$0001 Move only horizontally.
$FFFF Move only vertically.

SizeControl

Call # (not completed)

input: NewWidth:WORD New width of control.
NewHeight:WORD New height of control.
theControl:LONG Pointer to control.

output: None.

function: Erases theControl, adds NewWidth to CtrlRect's left side to get new right side, adds NewHeight to CtrlRect's top to get new bottom, and redraws.

CONTROL RECORD ACCESS

SetCtlState Call # (not completed)

input:	CtrlState:WORD	Control's new CtrlFlag.
	theControl:LONG	Pointer to control.
output:	None.	
function:	Sets control's CtrlFlag and redraws control if active, visible or highlighted state changes.	

GetCtlState Call # (not completed)

input:	theControl:LONG	Pointer to control.
output:	CtrlState:WORD	Control's CtrlFlag.
function:	Returns theControl's CtrlFlag.	

SetCtlValue Call # (not completed)

input:	CurValue:WORD	Current value of control.
	theControl:LONG	Pointer to control.
output:	None.	
function:	The new values are set, and the control redrawn. For scroll bars ScrollCur = CurValue. For check box and button controls CtrlFlag bit 5 = CurValue (TRUE or FLASE).	

GetCtlValue Call # (not completed)

input:	theControl:LONG	Pointer to control.
output:	CurValue:WORD	Control's current value.
function:	For scroll bars CurValue = ScrollCur. For check box and button controls CtrlFlag bit 5 is returned in CurValue.	

SetCtlMinMax Call # (not completed)

input:	MaxValue:WORD	Maximum value of control.
	MinValue:WORD	Minimum value of control.
	theControl:LONG	Pointer to control.
output:	None.	
function:	The new values are set, and the control redrawn. For scroll bars ScrollMin = MinValue, and ScrollMax = MaxValue. Nothing is done with check box and button controls.	

GetCtlMinMax Call # (not completed)

input:	theControl:LONG	Pointer to control.
output:	MinMax:LONG	Minimum value in high WORD, Maximum value in low WORD.
function:	Returns the minimum and maximum values of a control. Zero is returned for check box and button controls.	

FindButton Call # (not completed)

input:	theControl:LONG	Pointer to control.
output:	RadioButton:LONG	Highlighted radio button, or zero.
function:	This call searches the control list for the first active, visible, highlighted radion button. A zero is returned a button is not found. This call is provided as one way to unhighlight one radio button when another has been selected.	

DEFINING YOUR OWN CONTROLS

In addition to predefined controls, you can also define "c
need a three-way selector switch, a memory-space indica
thruster control for a spacecraft simulator--whatever your
indicators may occupy regions of any shape.

To define your own type of control, you write a control d
The Control Manager stores this address in the CtrlPimp
needs to perform a type-dependent action on the control, i

Keep in mind that the calls your application makes to use
definition function. Just as you need to know how to call
controls, each custom control type will have a particular ca
the control to work properly.

THE CONTROL DEFINITION FUNCTION

You can give your control definition function any name yo
one named **MyControl**:

MyControl	input:	message:WORD	Desired
		param:LONG	Differen
		theControl:LONG	Pointer t
	output:	RetValue:LONG	Depends

The message parameter identifies the desired operation. It h

initCntl	= 0	Do any additional contr
dispCntl	= 1	Take any additional dis
getTitle	= 2	Return address of cont
drawCntl	= 3	Draw the control (or co
testCntl	= 4	Test where mouse butt
setValue	= 5	Set control's current va
getValue	= 6	Return control's curren
setMinMax	= 7	Set control's minimum
getMinMax	= 8	Return control's minimu
calcCRgn	= 9	Set region with control

As described below in the discussions of the routines that per
passed for param depends on the operation. Where it's not m
ignored. Similarly, the control definition function is expected
indicated; in other cases, the function should return 0.

In some cases, the value of param or the function result is a part code. The part code 128 is reserved for future use and shouldn't be used for parts of your controls. Part codes greater than 128 should be used for indicators; however, 129 has special meaning to the control definition function.

User defined control record is as follows:

NextCtrl	LONG	Pointer to next control, zero = last control.
CtrlFlag	BYTE	Bits 4-0 = 5-31, control ID.
CtrlOwner	LONG	Pointer to window this control belongs to.
CtrlRect	RECT	Control's enclosing rectangle.
CtrlPimp	LONG	Address of application's handling routine.

You may append more data to the control record as you would like.

The following Control Manager routines will call your control definition function with the given inputs, expected results and returns:

NewControl

message = initCntl.
param = zero.

RetValue = doesn't matter.

Called after setting CtrlOwner and linking control into control list. This gives the definition function a chance to perform any type-specific initialization it may require.

DisposeControl

message = dispCntl.
param = zero.

RetValue = doesn't matter.

Called after the control has been removed from the screen and control list. Your function may then carry out any additional actions required when disposing of the control.

SetCTitle

message = getTitle.
param = zero.

RetValue = address of where to store title, or zero if no title.

Expects the control definition function to return an address of where the Control Manager can store a new string for the control's title. Return zero if the control doesn't have a title. If the string is stored, **DrawControls** will be called to draw the new title.

GetCTitle

message = getTitle.
param = zero.

RetValue = address of control's title, or zero if no title.

Expects the control definition function to return the address of the control's title. Return zero if the control doesn't have a title.

DrawControls

message = drawCntl.
param = part code, or zero for entire control.

RetValue = doesn't matter.

The message drawCntl asks the control definition function to draw all or part of the control. The value of param is a part code specifying which part of the control to draw, or 0 for the entire control. If the control is invisible (that is, if CtrlFlag bit 6 is 0), there's nothing to do; if it's visible, the definition function should draw it (or the requested part), taking into account the current values of highlight and active bits in CtrlFlag.

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. There's a special case, though, in which the draw routine has to allow for the fact that the indicator may have moved: This happens when the Control Manager procedures **SetCtlValue** and **SetCtlMinMax** call the control definition function to redraw the indicator after changing the control setting. Since they have no way of knowing what part code you chose for your indicator, they both pass 129 to mean the indicator. The draw routine must detect this part code as a special case, and remove the indicator from its former location before drawing it.

Note: If your control has more than one indicator, 129 should be interpreted to mean all indicators.

TestControl

message = testCntl.
param = Y coordinate in high WORD, X in low WORD.

RetValue = part code.

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 high-order WORD, and the horizontal coordinate is in the low-order WORD of param. 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.

DragControl

message = calcCRgn.
param = part code in high BYTE, region handle in low 3 BYTES.

RetValue = TRUE if region generated, FALSE if error.

This call is to ask for the region of a control, or part, so **DragControl** can drag an outline of it.

SetCtlValue

message = setValue.
param = value to set to.

RetValue = TRUE if value set, FALSE if not set.

Set the control's current value. Control should not be redrawn here, the Control Manager will ask for redraw elsewhere if RetValue was TRUE.

GetCtlValue

message = getValue.
param = zero.

RetValue = control's current value.

Return the control's current value.

SetCtlMinMax

message = setMinMax.
param = Maximum value in high WORD, minimum value in low WORD.

RetValue = TRUE if set, FALSE if not set.

Set the control's minimum and maximum values. Control should not be redrawn here, the Control Manager will ask for redraw elsewhere if RetValue was TRUE.

GetCtlMinMax

message = getMinMax.
param = zero.

RetValue = Maximum value in high WORD, minimum value in low WORD.

Return the control's minimum and maximum values, or zero if the control does not use them.

KillControl Doesn't call the control definition function directly, calls **DisposeControl** for each control in the control list.

HideControl Doesn't call the control definition function.

ShowControl Doesn't call the control definition function directly

HiliteControl Doesn't call the control definition function directly.

ControlState Doesn't call the control definition function.

FindControl Doesn't call the control definition function.

TrackControl Doesn't call the control function directly.

MoveControl Doesn't call the control function directly.

SizeControl Doesn't call the control function directly.

SetCtlState Doesn't call the control function.

GetCtlState Doesn't call the control function.