Custom Controls Information about defining your own controls

This topic covers how to create custom controls - control types other than push buttons, radio buttons, check boxes, and scroll bars.

The normal procedure is to write a code resource of type 'CDEF'. When the application creates the control, specify a "control type" in which the high 12 bits identify the 'CDEF' resource and the low 4 bits are a variation code (see **NewControl**). When this resource is read in, the low 24 bits of the contrlDefProc field of the ControlRecord is set to a handle to the 'CDEF' code.

Note: During application development, you may wish to keep the control definition code in your application. Refer to Custom Menus for details on how to do this.

Do not use the contrlDefProc to get at the variation code. Use **GetCVariant** which is available on the Mac Plus or higher. Your CDEF should call **SysEnvirons** or **Gestalt** if you want to take advantage of Color QuickDraw.

Your custom routine should be defined as a Pascal-type procedure that returns a 32-bit long:

```
#include <Controls.h>                                   Control Manager

pascal long     MyCtlDefFn(variation, theControl, msg, parm );
short           variation ;     which variation of the control is needed
ControlHandle   theControl ;    handle to ControlRecord of interest
short           msg ;           identifies the request to be handled
long            parm ;          data needed to complete the request
                returns         depends upon input value of msg
```

This function is called in lieu of the standard control definition routine when the contrlDefProc field of the ControlRecord points to it.

variation identifies which variation of the control must be handled.

theControl is a handle leading to a variable-length ControlRecord structure. The structure data contains all the information needed to manage the control.

msg identifies which subfunction is requested. It will be one of the named constants defined in Controls.h as discussed below.

parm is a 4-byte value whose data type can vary, depending upon the value of *msg* (see below).

Returns: a long; return value depends on *msg*.

Notes: While executing the custom handler, you may access the relevant data via:

```
value = (*theControl) -> fieldname;
```

Your custom control definition is normally laid out as a switch statement which handles the following cases:

drawCntl (0) Draw the control or the part specified in *parm* Check

contrlHilite and draw accordingly. If the control has an indicator and *parm* =129, you must erase the indicator from its previous position and redraw the indicator according to contrlValue. Note: only the **LoWord(***parm*)is relevant. The high word of that parameter is undefined.

testCntl (1) Check if the point named by *parm* is inside *theControl*, and if so, in which part. Return a part number (or 0 if it is not in the control).

calcCRgns (2) The low 3 bytes of *parm* are a RgnHandle (or all 4 bytes if in 32 bit mode). Mask the high byte and update that Region so that it encloses the entire control. Do not check or alter *parm* in this way under System 7 when 32-bit addressing is enabled. This will result in your control definition not being 32-bit clean. Two additional messages, calcCntlRgn and calcThumbRgn, both described below, provide a new mechanism for your control definition to calculate control regions under 32-bit addressing. See **Using Window and Control Definition Functions** under **About the Memory Manager** for more information.

initCntl (3) If any special initialization, calculation, or allocations should be done before the control is used, do them upon receiving this.

dispCntl (4) This message comes just before the ControlRecord is trashed via **DisposeControl** or **KillControls**. You should free up all memory allocated by the control definition procedure. Unless the control has a 'cctb' resource associated with it. For more information, see Auxiliary Controls.

posCntl (5) Redraw the indicator and update contrlValue; it has been moved **LoWord(***parm*) dots horizontally and **HiWord(***parm*) dots vertically. Note: You get the posCntl message whether you use default dragging or not (see dragCntl).

thumbCntl (6) *parm* points to a structure which you should fill with thumb motion constraints:

```
struct {
    Rect    limitRect;    /* on entry, topLeft is mouse pt */
    Rect    slopRect;     /* see DragControl */
    short   axis;
} thumbCntlParms;
```

Note: These values are ignored for custom dragging.

dragCntl (7) On entry, *parm* is 0 to drag the entire control or non-zero to drag just the indicator.

For default dragging, just return a 0 (Default dragging uses **DragGrayRgn** and information already in the control record). For custom dragging, follow the mouse around until the user releases the button. Handle all the details, and return a non-zero value.

autoTrack (8) This message comes when **TrackControl**'s final parameter is -1 and the contrlAction field of the ControlRecord is also -1. In that case, *parm* will contain a part code and the routine to handle this message should look like an *actionProc* , as described in **TrackControl**.
Note: Only the **LoWord(***parm*)is relevant. The high word of that parameter is undefined.

undoDev (9) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**.

cutDev (10) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**. You should call **DlgCut** upon receipt of this message.

copyDev (11) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**. You should call **DlgCopy** upon receipt of this message.

pasteDev (12) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**. You should call **DlgPaste** upon receipt of this message.

clearDev (13) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**. You should call **DlgDelete** upon receipt of this message.

cursorDev (14) This message is sent by the Control Panel as a result of the Desk Manager sending an edit message when an application calls **SystemEdit**.

The following two messages will be sent to your control definition under System 7 when 32-bit addressing is enabled.

calcCntlRgn (10) Under 32-bit addressing, this message is sent to a control definition procedure when it would previously have received calcCRgns. If the 24-bit addressing is enabled, calcCRgns will still be used. See **Using Window and Control Definition Functions** under **About the Memory Manager** for more information.

calcThumbRgn (11) Under 32-bit addressing, this message is sent to a control definition procedure when it would previously have received calcCRgns. If the 24-bit addressing is enabled, calcCRgns will still be used. See **Using Window and Control Definition Functions** under **About the Memory Manager** for more information.

Note that if you return 0 after receiving *msg* =7 (dragCntl), then you can ignore cases 5 and 6 (posCntl and thumbCntl). The autoTrack message lets you remove the custom *actionProc* (if any) from the application and put it in the control definition.

Do not dispose of a control that shares a color table that was not built in a resource because **DisposeControl** will free up that memory if the color table is not a resource.

If you implement custom dragging, TrackControl will always return 0.