19 Version 1.1

CUSTOM CONTROL DEFINITION FUNCTIONS AND VBL TASKS

Includes Demonstration Program CDEFandVBL

Introduction

As stated at Chapter 5 — Controls, the standard controls (buttons, checkboxes, radio buttons, pop-up menus, and scroll bars) may be supplemented with **custom controls**. Generally, the only type of custom control you application might need is some form of **slider control** (see Fig 1). Slider controls graphically represent a range of values that can be set by the user. The current setting is represented by the **indicator**, which is the part of the control that can be moved with the mouse.

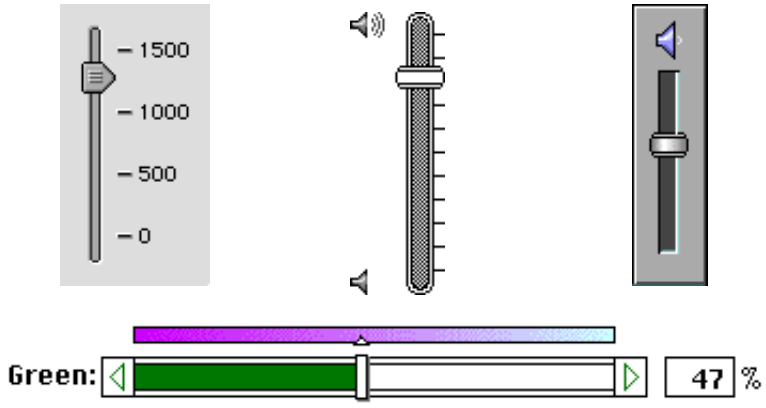


FIG 1 - TYPICAL SLIDER CONTROLS

If your application requires a slider control (or, indeed, any other custom control), you are faced with the task of writing your own **control definition function**.

When the indicator of a slider control is moved with the mouse, an animation process is involved. This animation is achieved by repeatedly erasing the indicator at its current location and then re-drawing it at a new location. One consideration applying to such animation is that, if the erasing and re-drawing of the moved image is performed in the window's graphics port itself, the image will appear to flicker. For that reason, it is essential that the moved image and its background be assembled offscreen and then copied to the window's graphics port. Another key consideration is that the copying of the assembled image to the window's graphics port should not be performed while the video circuitry is somewhere in the middle of its left-to-right/top-to-bottom refreshment of the monitor's screen. Unless the copying action is performed during the **vertical blanking** period (that is, the brief period during which the monitor's electron beam is switched off and returned from the lower right corner to the upper left corner of the monitor's screen), the image can appear to distort while it is moving, more particularly when it is moving fairly rapidly.

The necessary synchronisation of the redrawing of a moving image, such a slider control's indicator, with the monitor's refresh cycle can be achieved using a **vertical blanking (VBL) task**. As will be seen, the achievement of smooth animation is but one of the uses of VBL tasks.

Control Definition Functions (CDEFs)

Declaration

To create a custom control, you must write your own control definition function (CDEF), compile it as a resource type of type `'CDEF'`, and store it in the resource fork of the application that uses it. You must declare your CDEF like this:

```
pascal SInt32  controlDef(SInt16 varCode,ControlHandle theControl,SInt16 message,
                          SInt32 param);
```

`varCode` The variation code for this control. To derive the control definition ID for the control, add this value to the result of 16 multiplied by the resource ID of the `'CDEF'` resource containing this function.

Note: Whenever you create a control, you specify a control definition ID, which the Control Manager uses to determine the CDEF to be used. The control definition ID is an integer which contains the CDEF's resource ID in the upper 12 bits and a variation code in the lower four bits. Thus, for a given resource ID and variation code:

control definition ID = 16 x resource ID + variation code

You can define your own variation codes, which various Control Manager routines pass to your CDEF. This allows you to use one `'CDEF'` resource to handle several variations of the same control.

`theControl` A handle to the control that the operation will affect.

`message` A value which specifies which operation your function must undertake. Possible values are as follows:

Constant	Value	Operation
`drawCntl`	0	Draw the control or its part.
`testCntl`	1	Determine if the mouse-down occurred in a control.
`calcCRgns`	2	Calculate region for control or indicator (24-bit addressing).
`initCntl`	3	Perform any required additional control initialisation.
`dispCntl`	4	Perform any additional control disposal actions.
`posCntl`	5	Move indicator and update control record's `contrlValue` field.
`thumbCntl`	6	Calculate constraints for dragging the indicator.
`dragCntl`	7	Perform custom dragging of the control or its indicator.
`autoTrack`	8	Execute the action procedure specified by your function.
`calcCntlRgn`	10	Calculate region for control (32-bit addressing).
`calcThumbRgn`	11	Calculate region for indicator (32-bit addressing).

`param` A value whose meaning depends on the operation specified in the `message` parameter.

This function is called by the Control Manager in lieu of the standard control definition function when the `contrlDefProc` field of the control's control record points to it.

Default Dragging and Custom Dragging

One of the key decisions you must take before writing a CDEF for a control which is to be draggable, or which contains a part (such as an indicator) which is to be draggable, is whether to use **default dragging** or **custom dragging**[1].

When you specify default dragging, the Control Manager, using certain information provided by your CDEF, handles the dragging operation itself, following the mouse movement with a dotted outline of the control or part. The Control Manager calls your CDEF to redraw the control or part only once, that is, when the mouse button is released. On the other hand, when you specify custom dragging, your CDEF itself must perform the entire dragging operation. For example, if your CDEF supports a fully animated slider control, and the indicator of that control is being dragged, your CDEF must follow the mouse while the mouse button remains down, continually erasing and redrawing the indicator's image and updating the control's value.

An important aspect of your default dragging/custom dragging decision is that, when you elect to use custom dragging, some of the values listed above will never be passed to your CDEF by the Control Manager and others, though passed by the Control Manager, may be ignored. This means, of course, that there is no need for your CDEF to include routines which respond to those particular messages.

Responding to `message` Parameter Values

The Control Manager calls your CDEF under various circumstances, using the `message` parameter to specify the action required to be performed. The action that your CDEF should take, the data passed to it by the Control Manager in the `param` parameter, and the function result that your CDEF should return all depend on the value passed in the `message` parameter.

The following describes how your CDEF should respond to values passed by the Control Manager in the `message` parameter.

`initCntl`

Action Required by Control Manager: Perform any additional control initialisation as required.

Value in `param` : (Not applicable. Ignore)

When creating a new custom control, the Control Manager initialises fields of the control's control record and then passes `initCntl` to your CDEF to give it the opportunity to perform any additional initialisation. For example, you might want to create a control-specific data record and assign a handle to it to the `contrlData` field of the control's control record.

Value to Return: Always return `0`.

`drawCntl`

Action Required by Control Manager: Draw the control or part specified in the `param` parameter.

Value in `param` : The low-order word[2] contains either `0` (meaning the entire control), `129` (meaning the indicator), or some other value, (indicating a part code[3]).

This message is sent by the Control Manager when your application calls `UpdateControls` or `DrawControls` in its update event handling routine. In addition, `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` may call your CDEF to redraw the indicator.

---

[1]The ability to drag (that is, reposition) a control as a whole is something that few applications require. Ordinarily, therefore, a CDEF's dragging operations relate only to parts of a control, such as the indicator of a slider control.

[2]Note that, in the case of the `drawCntl` message, the high-order word may contain undefined data; therefore, evaluate only the low word.

[3]Do not use a part code 128 (reserved) or 129 (which, as stated, the Control Manager uses to signify an indicator which must be moved).

If the specified control is invisible (that is, if the `contrlVis` field of the control's control record is set to `0`), your CDEF should do nothing.

If the control is visible (`contrlVis` field set to 255), your CDEF should draw the control or the part, as specified in the `param` parameter, within the control's rectangle (stored in the `contrlRect` field of the control record). When drawing the control or its part, take into account the current value in the `contrlHilite`[4] field of the control's control record.

Part codes received in `param` reflect the part codes you assign to a part in your response to the `testCntl` message (see below). Note, however, that, since `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` have no way of knowing what part code you chose for your indicator, they all pass `129` in `param`.

Value to Return: Always return `0`.

testCntl

Action Required by Control Manager: Determine whether the point passed in the `param` parameter is inside a control part and, if so, in which part.

Value in `param`: Specifies a point in local coordinates. The high-order word contains the point's vertical coordinate and the low-order word contains the horizontal coordinate.

This message is sent by `FindControl`, which returns whatever is returned by your CDEF to the application.

Value to Return: Your part code for the part that contains the specified point, or `0` if the point is outside the control or the control is inactive.

dragCntl

Action Required by Control Manager: Advise the Control Manager whether default dragging or custom dragging is being used. Also, if custom dragging is being used, perform the dragging operation.

Value in `param`: Specifies whether the user is dragging the indicator or the whole control. `0` means the user is dragging the entire control. Any non-zero value means that the user is dragging the indicator.

By passing `dragCntl`, the Control Manager is providing your CDEF with the opportunity to advise the Control Manager whether it will be using its own method for dragging a control or its indicator (custom dragging) or whether it wants to use the Control Manager's method (default dragging). The following explains the requirements of the two methods:

- **Default Dragging**. If you use default dragging, you should call `DragControl` to reposition the entire control in the window (something that the vast majority of applications never do) or the Window Manager function `DragGrayRegion` to drag the control's indicator only. As the mouse moves, `DragControl` moves a dotted outline of the control and `DragGrayRegion`, using information already in the control record, moves a dotted outline of the specified (indicator) region. Accordingly, default dragging is not suitable if you require fully animated indicator movement.
- **Custom Dragging.** If you use custom dragging, your CDEF must itself drag the specified control or indicator, following the cursor until the user releases the mouse button, as follows:

[4]Recall from Chapter 5 — Controls that the `contrlHilite` field specifies whether and how the control is to be displayed, indicating whether it is active or inactive The value `0` signifies an active control. The value `255` signifies that the control is to be made inactive and drawn accordingly.

- If the user drags the entire control, your CDEF should use `MoveControl` to reposition the control to its new location after the user releases the mouse button.

- If the user drags the indicator, your CDEF must follow the mouse while the mouse button remains down, continually redrawing the control in its new location and updating the `contrlValue` field in the control's control record.

Note that, when custom dragging is specified, `TrackControl` always returns `0` regardless of whether or not the cursor is still within the control when the button is released.

If you specify custom dragging, your CDEF can ignore `posCntl` and `thumbCntl` messages. In addition, note that the `calcCRgns`, `calcCntlRgn`, and `calcThumbRgn` messages will never be sent to your CDEF when custom dragging is specified.

Value to Return: To advise the Control Manager that default dragging is being used, return 0. To advise the Control Manager that custom dragging is being used, return a non-zero result.

`dispCntl`

Action Required by Control Manager: Perform any additional disposal actions, as required.

Value in `param`: (Not applicable. Ignore)

`DisposeControl` passes `dispCntl` to your CDEF to give it the opportunity to perform any additional actions when disposing of a control, such as freeing up any memory allocated by your CDEF. For example, the standard CDEF for scroll bars releases the memory occupied by the scroll box region, whose handle is kept in the `contrlData` field of the control's control record.

Value to Return: Always return `0`.

`posCntl`

Action Required by Control Manager: Erase the indicator, redraw it in the new position specified in `param`, and update the `contrlValue` field of the control's control record.

Value in `param`: A point (in local coordinates) specifying the vertical and horizontal offset, in pixels, by which your CDEF should move the indicator from its current position. The vertical offset is in the high-order word and horizontal offset is in the low-order word.

This message is received whether you specify default dragging or custom dragging. However, your CDEF can ignore it if you have specified custom dragging; accordingly, the following is relevant only to default dragging.

`TrackControl` passes `posCntl` when a mouse-up event occurs in the indicator of your control. Typically, the value in `param` is the offset between the points where the user pressed and released the mouse button while dragging the indicator.

Your CDEF should calculate the control's new setting based on the given offset and then, to reflect the new setting, redraw the control and update the `contrlValue` field of the control's control record.

Note that `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` do not call your CDEF with the `posCntl` message. Instead, they pass the `drawCntl` message.

Value to Return: Always return 0.

`thumbCntl`

Action Required by Control Manager: Calculate constraints for dragging the indicator.

Value in `param`: A pointer to this data structure:

```
struct
{
  Rect   limitRect;
  Rect   slopRect;
  short  axis;
} thumbCntlParms;
```

This message is received whether you specify default dragging or custom dragging. If you have specified custom dragging, however, your CDEF should ignore `thumbCntl` and implement its own dragging constraints within its response to the `dragCntl` message; accordingly, the following is relevant only to default dragging.

On entry, the field `limitRect.topLeft` contains the point where the mouse-down event first occurred. Your CDEF should calculate values (analogous to the `limitRect`, `slopRect`, and `axis` parameters of `DragControl`) which constrain indicator dragging. Your CDEF should store the appropriate values into the fields of the record pointed to by `param`. (Those fields are, incidentally, analogous to the similarly-named parameters to the Window Manager function `DragGrayRgn`.)

Value to Return: Always return 0.

`calcCRgns`, `CalcCntlRgn`, and `CalcThumbRgn`

Action Required by Control Manager: Calculate the control or indicator region, as specified.

Value in `param`: A QuickDraw region handle. It is the QuickDraw region that you calculate. (Note that the low three bytes of `param` contain the handle in 24-bit addressing mode. All four bytes are used in 32-bit addressing mode.)

These messages will never be sent by the Control Manager if you use custom dragging.

If your CDEF specifies default dragging, the Control Manager passes `calcCRgns` when the 24-bit Memory Manager is in operation. When the 32-bit Memory Manager is in operation, the Control Manager passes either `calcCntlRgn` or `calcThumbRgn`. Your CDEF should respond to all three constants.

When `calcCRgns` is passed, if the high-order bit of `param` is set, the region requested is that of the control's indicator; otherwise the region requested is that of the entire control. Your CDEF should clear the high bit of the region handle before calculating the region.

When `calcCntlRgn` is passed, your CDEF should calculate the region occupied by the control. When `calcThumbRgn` is passed, your CDEF should calculate the region occupied by the indicator. Your CDEF should express the region in local coordinates.

Value to Return: Always return `0`.

`autoTrack`

Action Required by Control Manager: Execute an **action procedure** specified by your CDEF.

Value in `param`: In the low-order word[5], the part code of the part where the mouse-down event occurred.

The `autoTrack` message allows you to locate any custom action procedure used by your control in your CDEF rather than in your application. You will need an action procedure if, for example, your control has arrow boxes and a repetitive action needs to be performed while the mouse button remains down in one of those boxes.

[5]The high-order word may contain undefined data; therefore, evaluate only the low word.

Your CDEF will be sent the `autoTrack` message when your application passes `(ProcPtr)-1` in the `actionProc` parameter of the `TrackControl` function and the `contrlAction` field of the control's control record also contains `(ProcPtr)-1`.

Action procedures were first introduced at Chapter 5 — Controls, and the demonstration program Controls2 defines an action procedure which is called repeatedly when the mouse button is held down while the cursor is in the scroll arrows or gray areas of a vertical scroll bar.

Vertical Blanking (VBL) Tasks

VBL Tasks and the Vertical Retrace Manager

The video circuitry in a Macintosh refreshes the screen at regular intervals, the exact interval depending on the video hardware. For built-in monitors, the refresh rate is 60.15 times a second. To refresh the screen, the monitor's electron beam draws in horizontal lines, starting at the upper left corner, finishing at the lower right corner, and then jumping back to the upper left corner. When the electron beam returns from the lower right corner to the upper left corner, the video circuitry generates a **vertical retrace interrupt** or **vertical blanking (VBL) interrupt**.

The Vertical Retrace Manager schedules tasks, known as **VBL tasks**, for execution during the vertical retrace interrupt. The Operating System itself uses the Vertical Retrace Manager to perform certain housekeeping operations, such as updating the global variable `Ticks` and the position of the cursor (every interrupt) and checking whether a disk has been inserted (every 30 interrupts).

You can also use the Vertical Retrace Manager to install your own recurrent tasks which, for some reason, you do not want to execute in your main event loop. Be aware, however, that:

- The Vertical Retrace Manager is useful only for small, repetitive tasks which do not allocate or release memory.
- The Vertical Retrace Manager is not an absolute timing device. Its operations are always relative to the VBL interrupt, which is sometimes disabled — for example, during disk access. (This latter explains the jerky cursor movement experienced during disk operations.)

VBL tasks installed by the Operating System, incidentally, are not maintained in the same queue as that used by application-defined VBL tasks.

Types of VBL Tasks

There are two general types of VBL tasks:

- **Slot-Based VBL Tasks.** Slot-based VBL tasks are linked to an external video monitor. Because different monitors have different refresh rates, and hence execute VBL tasks at different intervals, a separate task queue is maintained for each attached video device. When a VBL interrupt occurs for one of these devices, the tasks in the queue relating to the slot holding that device's video card are executed. A slot-based VBL task is installed using `SlotVInstall`.
- **System-Based VBL Tasks.** System-based VBL tasks apply to Macintoshes which have only a built-in monitor (such as the Macintosh Classic). On such machines, there is no need to isolate VBL tasks into separate queues. System-based VBL tasks are installed using `VInstall`.

To maintain compatibility on modular Macintoshes for software which uses `VInstall`, the Operating System generates a special interrupt at a frequency identical to the retrace rate on compact Macintoshes. This ensures that application tasks installed using the `VInstall` function, as well as the periodic system tasks previously described, are performed as usual.

VBL Task Rules

A VBL task which violates any of the following rules may cause a system crash:

- A VBL task must not allocate, move, or purge memory, or call any Toolbox routines which may do so.
- A VBL task cannot call a routine from any other code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
- A VBL task cannot access your application's global variables unless it sets up the application's A5 world properly.
- A VBL task's code, and any data accessed during the execution of the task, must be locked into physical memory if virtual memory is in operation.

VBL Tasks and Foreground/Background Switching

Some VBL tasks may be intended to perform services which are useful only to the application, and which should therefore cease execution if the application is switched to the background. Others may be intended to continue to execute even when the application is no longer in the foreground.

System-Based VBL Tasks

If the address of a system-based VBL task (not the same thing as the address of the VBL task record) is anywhere in the partition of the application that installed it, the Process Manager automatically disables that task when it is sent to the background. Then, when the application regains control of the processor (through either a minor or major switch), the task is re-enabled. This does not apply if the address of a system-based VBL task is in the system partition[6].

Note that, in the case of the address of the system-based task being in the application's partition, the task is re-enabled when the application receives processing time, which can occur without the application necessarily returning to foreground. For that reason, you may want to disable a system-based VBL task manually. This can be done using the same procedure as that applying to the disabling of a slot-based VBL task (see below).

Slot-Based VBL Tasks

By contrast, the Process Manager never disables a slot-based VBL task, no matter where the task is located. Accordingly, if you want a slot-based VBL task to be disabled when your application is in the background, you must do it yourself, either by removing the task record from the VBL queue or by setting the `vblCount` field of the task record (see below) to `0`. You can do this in response to a suspend event. Then, when your application receives a resume event, you can re-enable the task by re-installing the task record or by re-setting the `vblCount` field of the VBL task record (see below) to the appropriate value.

Installing and Removing a VBL Task

You use the Vertical Retrace Manager to install and remove **VBL task records** in and from system-based or slot-based vertical retrace queues. Before you call `VInstall` or `SlotVInstall` to install a task record, you must first fill in the last four of the VBL task record's fields.

[6]You load a system-based task's task record into the system partition when you want the task to be a **persistent VBL task**, that is, a task that continues to be executed even when the application which installed it is no longer in control of the CPU. (Note that slot-based VBLs are always persistent no matter where you put the task record.)

The VBL Task Record

The VBL task record is defined by the `VBLTask` data type:

```
struct VBLTask
{
  QElemPtr  qLink;
  short     qType;
  VBLUPP    vblAddr;
  short     vblCount;
  short     vblPhase;
};

typedef struct VBLTask VBLTask,*VBLTaskPtr;
```

Field Descriptions

`qLink` Pointer to the next entry in the queue. (This field is not set by the application. It is set by the Vertical Retrace Manager.)

`qType` The queue type. This must be set to `vType`.

`vblAddr` Pointer to the procedure that the Vertical Retrace Manager is to execute.

`vblCount` The number of interrupts before the routine first executes.

The Vertical Retrace Manager lowers this number by 1 during each interrupt. If the value in `vblCount` is `0`, the task will not execute. If, when `vblCount` contains `0`, ou want the procedure to be executed again, you must reset the `vblCount` field to the required value.

Setting this field to `0` is one way of disabling a task. A more common approach is to remove the task record from its queue by calling `VRemove` or `SlotVRemove`, although this should not be done by the task itself.

`vblPhase` The phase count of the VBL task.

In most cases, you can set this field to `0` . However, if you install multiple tasks with the same `vblCount` at the same time, you can assign them different `vblPhase` values so that the tasks are not executed during the same interrupt. The value in the `vblPhase` field must be less than the value in the `vblCount` field.

Installing a VBL Task

For any particular VBL task, you must first decide whether to install it as a system-based VBL task or as a slot-based VBL task. The following considerations apply:

- **Slot-Based VBL Tasks.** You need to install a task as a slot-based VBL task only if the execution of the task needs to be synchronised with the retrace rate of a particular external monitor. This will be the case, for example, if you want the repetitive re-drawing of a moving image, such as a slider control's indicator, to occur only during that particular monitor's vertical blanking period.

- **System-Based VBL Tasks.** If the task performs no processing likely to affect the appearance of the screen, and no processing that depends on the state of an external monitor, you can install it as a system-based VBL task.

The next steps are to define the VBL task itself (so as be able to assign its address to the `vblAddr` field of the VBL task record) and, in the case of slot-based VBL tasks, call `LMGetMainDevice` and `GetDCtlEntry` to find the slot number of the video device to whose retrace the VBL task is to be synchronised. The final step is to fill in a VBL task record and install it into the appropriate queue.

Accessing a Task Record

Recall that, if a VBL task is to be executed recurrently, it must reset the `vblCount` field of the task record each time it is executed. A repetitive VBL task must therefore be able to access its task record so that it can reset the `vblCount` field.

When the Vertical Retrace Manager executes the VBL task, it places the address of the VBL task into the A0 register. The following defines an in-line function which moves that value onto the stack:

```
pascal SInt32  GetVBLRec(void) = 0x2E88;
```

This in-line function, which returns a long integer specifying the address of the task record, should be called only from a VBL task. It will not work if called from the main program. In addition, the call should be the first line of your VBL task, because other processing could change the value in A0.

Accessing Application Global Variables

Recall from Chapter 1 that the boundary between the current application's global variables and its application parameters are stored in the microprocessor's A5 register. Since all applications share this register, the Process Manager keeps track of the address of your application's A5 world when a major or minor switch yields control of the microprocessor to another application. Then, when your application regains access to the CPU, the Process Manager restores that address to the A5 register.

Because VBL tasks are interrupt routines, they could well execute when the value in the A5 register does not point to your application's A5 world. As a result, if you need to access your application's global variables in a VBL task, you need to set the A5 register to its correct value when your VBL task begins executing and restore the previous value upon exit.

To achieve this, your application should save its A5 using `SetCurrentA5`. Then, at interrupt time, the VBL task can begin by calling `SetA5` to, firstly, set the A5 register to this saved value and, secondly, save the value that was in the A5 register immediately prior to the call. The VBL task should end with another call to `SetA5`, this time to restore the initial value.

The only memory location that a VBL task has access to is the address of the task record. Accordingly, if your application stores its A5 directly following the task record, it can locate this value by first locating the task record. To store the A5 value directly following the task record, define a new data type whose first field contains the VBL task record and whose second field will hold the value in the A5 register retrieved by a call to `SetCurrentA5`:

```
typedef struct
{
  VBLTask  vblTaskRec;  // The VBL task record.
  long     vblA5        // Saved value of A5.
} VBLRec, *VBLRecPtr;
```

Relevant Control Manager Constants

Constants for `message` Parameter in Control Definition Function

```
drawCntl           = 0    Draw the control or its part.
testCntl           = 1    Determine if the mouse-down occurred in a control.
calcCRgns          = 2    Calculate region for control or indicator (24-bit addressing).
initCntl           = 3    Perform any required additional control initialisation.
dispCntl           = 4    Perform any additional control disposal actions.
posCntl            = 5    Move indicator and update control record's contrlValue field.
thumbCntl          = 6    Calculate constraints for dragging the indicator.
dragCntl           = 7    Perform custom dragging of the control or its indicator.
autoTrack          = 8    Execute the action procedure specified by your function.
calcCntlRgn        = 10   Calculate region for control (32-bit addressing).
calcThumbRgn       = 11   Calculate region for indicator (32-bit addressing).
drawThumbOutline   = 12   Draw indicator outline.
```

Vertical Retrace Manager Data Types and Routines

Data Types

VBL Task Record

```
struct VBLTask
{
  QElemPtr  qLink;
  short     qType;
  VBLUPP    vblAddr;
  short     vblCount;
  short     vblPhase;
};

typedef struct VBLTask VBLTask,*VBLTaskPtr;
```

Routines

Slot-Based Installation and Removal Routines

```
OSErr   SlotVInstall(QElemPtr vblBlockPtr,short theSlot);
OSErr   SlotVRemove(QElemPtr vblBlockPtr,short theSlot);
```

System-Based Installation and Removal Routines

```
OSErr   VInstall(QElemPtr vblTaskPtr);
OSErr   VRemove(QElemPtr vblTaskPtr);
```

Utility Routines

```
OSErr   AttachVBL(short theSlot);
OSErr   DoVBLTask(short theSlot);
QHdrPtr GetVBLQHdr(void);
```

Demonstration Program

```
// ##############################################################################################
// CDEFandVBL.c
// ##############################################################################################
//
// This program opens a window containing a slider control panel.  The slider control
// panel contains two radio button controls and a slider control.  The radio buttons
// activate and deactivate the slider control.
//
// The slider control uses a custom control definition function (CDEF).  The CDEF
// utilises a VBL task to delay the drawing of a moved indicator in the graphics port
// until the vertical blank period is entered.  The radio buttons also use a custom CDEF.
// On colour or grayscale displays, the appearance of the controls conforms to that
// specified in the document Apple Grayscale Appearance for System 7.5 published by Apple
// Computer, Inc.
//
// This program also includes a demonstration of an animated cursor which utilises a
// system-based VBL task to increment the frames of the animation.  This demonstration
// is invoked by choosing the VBL Task Animated Cursor item in the Demonstration menu
// and may be cancelled by pressing any key.
//
// The program utilises the following resources:
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Demonstration
//    menus (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially visible) and a 'wctb' resource (purgeable)
//    for the window containing the slider control panel.
//
// •  'CNTL' resources  (purgeable) for the radio button and slider controls.
//
// •  The 'CDEF' code resources (non-purgeable).
//
// •  An 'acur' resource (purgeable) and 'CURS' resources (purgeable) for the animated
```

```
//    cursor.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
//    is32BitCompatible flags set.
//
// ############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>
#include <Gestalt.h>
#include <Resources.h>
#include <Retrace.h>
#include <Palettes.h>
#include <LowMem.h>

// ……… defines

#define mApple            128
#define  iAbout           1
#define mFile             129
#define  iQuit            11
#define mDemonstration    131
#define  iVBLAnimCursor   1
#define rMenubar          128
#define rWindow           128
#define rFingersCursor    128
#define rStartRadioButton 128
#define rStopRadioButton  129
#define rSliderControl    130
#define MAXLONG           0x7FFFFFFF

// ……… typedefs

typedef struct
{
  SInt16      numberOfFrames;
  SInt16      whichFrame;
  CursHandle  frame[];
} animCurs, *animCursPtr, **animCursHandle;

typedef struct
{
  VBLTask vblTaskRec;
  SInt32  thisApplicationsA5;
} VBLRec, *VBLRecPtr;

// ……… global variables

Boolean         gColorQuickDrawPresent = false;
Boolean         gColourDisplay         = false;
Boolean         gDone;
SInt32          gSleepTime;
Boolean         gInBackground;
WindowPtr       gWindowPtr;
animCursHandle  gAnimCursHdl;
VBLRec          gVBLRec;
SInt16          gVBLCount;
Boolean         gAnimatedCursorActive  = false;
RGBColor        gWindowColour          = { 0xDDDD, 0xDDDD, 0xDDDD };
ControlHandle   gSliderControlHdl;
ControlHandle   gStartControlHdl;
ControlHandle   gStopControlHdl;

// ……… function prototypes

void    main                    (void);
void    doInitManagers          (void);
void    doEvents                (EventRecord *);
void    doMouseDown             (EventRecord *);
void    doUpdate                (EventRecord *);
```

```
111 void    doActivate              (EventRecord *);
112 void    doActivateWindow        (Boolean);
113 void    doOSEvent               (EventRecord *);
114 void    doInContent             (EventRecord *,WindowPtr);
115 void    doMenuChoice            (SInt32 menuChoice);
116 void    doGetSliderControlSuite (void);
117 void    doDrawControlsPanel     (void);
118 void    doStartAnimCursor       (void);
119 Boolean doGetAnimCursor         (SInt16);
120 void    doInstallSystemVBLTask  (void);
121 void    animCursVBLTask         (void);
122 void    doStopAnimCursor        (void);
123
124 // …… in-line glue for GetVBLRec
125
126 pascal SInt32  GetVBLRec (void) = 0x2E88;
127
128 // ############################################################################## main
129
130 void  main(void)
131 {
132   OSErr         osErr;
133   SInt32        response;
134   GDHandle      mainDeviceHdl;
135   SInt16        bitsPerPixel;
136   Handle        menubarHdl;
137   MenuHandle    menuHdl;
138   EventRecord   eventRec;
139
140   // …… initialise managers
141
142   doInitManagers();
143
144   // …… check for Color QuickDraw
145
146   osErr = Gestalt(gestaltQuickdrawVersion,&response);
147   if(response >= gestalt8BitQD)
148   {
149     gColorQuickDrawPresent = true;
150
151     mainDeviceHdl = LMGetMainDevice();
152     bitsPerPixel = (*(*mainDeviceHdl)->gdPMap)->pixelSize;
153     if(bitsPerPixel > 1)
154       gColourDisplay = true;
155   }
156
157   // …… set up menu bar and menus
158
159   menubarHdl = GetNewMBar(rMenubar);
160   if(menubarHdl == NULL)
161     ExitToShell();
162   SetMenuBar(menubarHdl);
163   DrawMenuBar();
164
165   menuHdl = GetMenuHandle(mApple);
166   if(menuHdl == NULL)
167     ExitToShell();
168   else
169     AppendResMenu(menuHdl,'DRVR');
170
171   // …… open window
172
173   if(gColorQuickDrawPresent)
174   {
175     if(!(gWindowPtr = GetNewCWindow(rWindow,NULL,(WindowPtr)-1)))
176       ExitToShell();
177   }
178   else
179   {
180     if(!(gWindowPtr = GetNewWindow(rWindow,NULL,(WindowPtr)-1)))
181       ExitToShell();
182   }
183
184   SetPort(gWindowPtr);
185
186   // …… get slider control suite
187
```

```
188     doGetSliderControlSuite();
189
190     // ……… enter eventLoop
191
192     gDone = false;
193     gSleepTime = MAXLONG;
194
195     while(!gDone)
196     {
197       if(WaitNextEvent(everyEvent,&eventRec,gSleepTime,NULL))
198         doEvents(&eventRec);
199     }
200   }
201
202   // ########################################################################## doInitManagers
203
204   void  doInitManagers(void)
205   {
206     MaxApplZone();
207     MoreMasters();
208
209     InitGraf(&qd.thePort);
210     InitFonts();
211     InitWindows();
212     InitMenus();
213     TEInit();
214     InitDialogs(NULL);
215
216     InitCursor();
217     FlushEvents(everyEvent,0);
218   }
219
220   // ############################################################################### doEvents
221
222   void  doEvents(EventRecord *eventRecPtr)
223   {
224     switch(eventRecPtr->what)
225     {
226       case mouseDown:
227         doMouseDown(eventRecPtr);
228         break;
229
230       case keyDown:
231       case autoKey:
232         if(gAnimatedCursorActive)
233           doStopAnimCursor();
234         break;
235
236       case updateEvt:
237         doUpdate(eventRecPtr);
238         break;
239
240       case activateEvt:
241         doActivate(eventRecPtr);
242         break;
243
244       case osEvt:
245         doOSEvent(eventRecPtr);
246         HiliteMenu(0);
247         break;
248     }
249   }
250
251   // ########################################################################## doMouseDown
252
253   void  doMouseDown(EventRecord *eventRecPtr)
254   {
255     SInt16      partCode;
256     WindowPtr   windowPtr;
257     MenuHandle  menuHdl;
258
259     partCode = FindWindow(eventRecPtr->where,&windowPtr);
260     menuHdl = GetMenuHandle(mDemonstration);
261
262     switch(partCode)
263     {
264       case inMenuBar:
```

```
      if(gAnimatedCursorActive)
        DisableItem(menuHdl,iVBLAnimCursor);
      else
        EnableItem(menuHdl,iVBLAnimCursor);
      doMenuChoice(MenuSelect(eventRecPtr->where));
      break;

    case inSysWindow:
      SystemClick(eventRecPtr,windowPtr);
      break;

    case inContent:
      if(windowPtr != FrontWindow())
        SelectWindow(windowPtr);
      else
        doInContent(eventRecPtr,windowPtr);
      break;

    case inDrag:
      DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
      break;
  }
}

// ############################################################################## doUpdate

void  doUpdate(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  windowPtr = (WindowPtr) eventRecPtr->message;

  BeginUpdate(windowPtr);

  SetPort(windowPtr);
  UpdateControls(windowPtr,windowPtr->visRgn);
  doDrawControlsPanel();

  EndUpdate(windowPtr);
}

// ############################################################################ doActivate

void  doActivate(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  Boolean   becomingActive;

  windowPtr = (WindowPtr) eventRecPtr->message;

  becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
  doActivateWindow(becomingActive);
}

// ######################################################################## doActivateWindow

void  doActivateWindow(Boolean becomingActive)
{
  SInt16 controlVal;

  if(becomingActive)
  {
    controlVal = GetControlValue(gStartControlHdl);
    if(controlVal == 1)
      HiliteControl(gSliderControlHdl,0);
    HiliteControl(gStartControlHdl,0);
    HiliteControl(gStopControlHdl,0);
  }
  else
  {
    HiliteControl(gSliderControlHdl,255);
    HiliteControl(gStartControlHdl,255);
    HiliteControl(gStopControlHdl,255);
  }

  doDrawControlsPanel();
}

```

```
342 // ############################################################################## doOSEvent
343
344 void  doOSEvent(EventRecord *eventRecPtr)
345 {
346   switch((eventRecPtr->message >>24) & 0x000000FF)
347   {
348     case suspendResumeMessage:
349       gInBackground = (eventRecPtr->message & resumeFlag) == 0;
350       doActivateWindow(!gInBackground);
351       break;
352
353     case mouseMovedMessage:
354       break;
355   }
356 }
357
358 // ############################################################################ doInContent
359
360 void  doInContent(EventRecord *eventRecPtr,WindowPtr windowPtr)
361 {
362   ControlHandle controlHdl;
363   SInt16        partCode;
364   Rect          theRect;
365   Str255        theString;
366
367   GlobalToLocal(&eventRecPtr->where);
368
369   partCode = FindControl(eventRecPtr->where,windowPtr,&controlHdl);
370
371   if(controlHdl == gSliderControlHdl)
372   {
373     if(partCode == kControlIndicatorPart)
374       TrackControl(controlHdl,eventRecPtr->where,NULL);
375
376     RGBBackColor(&gWindowColour);
377     SetRect(&theRect,253,107,390,119);
378     FillRect(&theRect,&qd.white);
379     MoveTo(255,117);
380     DrawString("\pSlider Control Value: ");
381     NumToString((long) GetControlValue(controlHdl), theString);
382     DrawString(theString);
383   }
384   else if(controlHdl == gStartControlHdl || controlHdl == gStopControlHdl)
385   {
386     if(TrackControl(controlHdl,eventRecPtr->where,NULL))
387     {
388       if(controlHdl == gStartControlHdl)
389       {
390         HiliteControl(gSliderControlHdl,0);
391         SetControlValue(gStartControlHdl,1) ;
392         SetControlValue(gStopControlHdl,0);
393       }
394       else if(controlHdl == gStopControlHdl)
395       {
396         SetControlValue(gSliderControlHdl,0);
397         HiliteControl(gSliderControlHdl,255);
398         SetControlValue(gStartControlHdl,0);
399         SetControlValue(gStopControlHdl,1);
400
401         RGBBackColor(&gWindowColour);
402         SetRect(&theRect,253,107,390,119);
403         FillRect(&theRect,&qd.white);
404       }
405     }
406   }
407 }
408
409 // ############################################################################ doMenuChoice
410
411 void  doMenuChoice(SInt32 menuChoice)
412 {
413   SInt16  menuID, menuItem;
414   Str255  itemName;
415   SInt16  daDriverRefNum;
416
417   menuID = HiWord(menuChoice);
418   menuItem = LoWord(menuChoice);
```

```
419
420   if(menuID == 0)
421     return;
422
423   switch(menuID)
424   {
425     case mApple:
426       if(menuItem == iAbout)
427         SysBeep(10);
428       else
429       {
430         GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
431         daDriverRefNum = OpenDeskAcc(itemName);
432       }
433       break;
434
435     case mFile:
436       if(menuItem == iQuit)
437       {
438         DisposeWindow(gWindowPtr);
439         gDone = true;
440       }
441       break;
442
443     case mDemonstration:
444       if(menuItem == iVBLAnimCursor)
445         doStartAnimCursor();
446       break;
447   }
448
449   HiliteMenu(0);
450 }
451
452 // ############################################################## doGetSliderControlSuite
453
454 void  doGetSliderControlSuite(void)
455 {
456   gSliderControlHdl = GetNewControl(rSliderControl,gWindowPtr);
457   HiliteControl(gSliderControlHdl,255);
458
459   gStartControlHdl = GetNewControl(rStartRadioButton,gWindowPtr);
460   gStopControlHdl = GetNewControl(rStopRadioButton,gWindowPtr);
461
462   doDrawControlsPanel();
463 }
464
465 // ################################################################### doDrawControlsPanel
466
467 void  doDrawControlsPanel(void)
468 {
469   GDHandle  mainDeviceHdl;
470   SInt16    bitsPerPixel;
471   SInt16    fontNum, a;
472   RGBColor  gray8 = { 0x7777,0x7777,0x7777 };
473
474   GetFNum("\pChicago",&fontNum);
475   TextFont(fontNum);
476   TextSize(12);
477
478   mainDeviceHdl = LMGetMainDevice();
479   bitsPerPixel = (*(*mainDeviceHdl)->gdPMap)->pixelSize;
480
481   if(bitsPerPixel > 1)
482     gColourDisplay = true;
483   else
484     gColourDisplay = false;
485
486   for(a=0;a<2;a++)
487   {
488     if(a == 0)
489       ForeColor(whiteColor);
490     else
491     {
492       if(!gInBackground)
493       {
494         if(gColorQuickDrawPresent && gColourDisplay)
495           ForeColor(blackColor);
```

```
        else
        {
          ForeColor(blackColor);
          PenPat(&qd.black);
          TextMode(srcOr);
        }
      }
      else
      {
        if(gColorQuickDrawPresent && gColourDisplay)
          RGBForeColor(&gray8);
        else
        {
          ForeColor(blackColor);
          PenPat(&qd.gray);
          TextMode(grayishTextOr);
        }
      }
    }

    if(a == 0 && !gColourDisplay)
      continue;
    else
    {
      MoveTo(156-a,22-a);
      LineTo(152-a,22-a);
      LineTo(152-a,230-a);
      LineTo(246-a,230-a);
      LineTo(246-a,22-a);
      LineTo(242-a,22-a);

      MoveTo(163-a,26-a);
      DrawString("\pEngine RPM");
    }
  }

  ForeColor(blackColor);

  GetFNum("\pGeneva",&fontNum);
  TextFont(fontNum);
  TextSize(10);
  PenPat(&qd.black);
  TextMode(srcOr);
}

// ########################################################################## doStartAnimCursor

void  doStartAnimCursor(void)
{
  gVBLCount = 30;
  gSleepTime = 0;

  if(doGetAnimCursor(rFingersCursor) == false)
    ExitToShell();

  doInstallSystemVBLTask();

  gAnimatedCursorActive = true;

  MoveTo(40,110);
  DrawString("\pPress any key to");
  MoveTo(30,125);
  DrawString("\pstop animated cursor");
}

// ########################################################################## doGetAnimCursor

Boolean  doGetAnimCursor(SInt16 resourceID)
{
  SInt16  cursorID, a = 0;
  Boolean noError = false;

  if((gAnimCursHdl = (animCursHandle) GetResource('acur',resourceID)))
  {
    noError = true;
    while((a < (*gAnimCursHdl)->numberOfFrames)  && noError)
    {
```

```
573         cursorID = (SInt16) HiWord((SInt32) (*gAnimCursHdl)->frame[a]);
574
575         (*gAnimCursHdl)->frame[a] = GetCursor(cursorID);
576         if((*gAnimCursHdl)->frame[a])
577           a++;
578         else
579           noError = false;
580       }
581     }
582
583     if(noError)
584       (*gAnimCursHdl)->whichFrame = 0;
585
586     return(noError);
587   }
588
589   // ############################################################## doInstallSystemVBLTask
590
591   void  doInstallSystemVBLTask(void)
592   {
593     gVBLRec.vblTaskRec.qType    = vType;
594     gVBLRec.vblTaskRec.vblAddr  = (VBLUPP) animCursVBLTask;
595     gVBLRec.vblTaskRec.vblCount = gVBLCount;
596     gVBLRec.vblTaskRec.vblPhase = 0;
597
598     gVBLRec.thisApplicationsA5 = SetCurrentA5();
599
600     VInstall((QElemPtr) &gVBLRec.vblTaskRec);
601   }
602
603   // ################################################################################ animCursVBLTask
604
605   void  animCursVBLTask(void)
606   {
607     VBLRecPtr vblRecPtr;
608     SInt32    currentA5;
609
610     vblRecPtr = (VBLRecPtr) GetVBLRec();
611     currentA5 = SetA5(vblRecPtr->thisApplicationsA5);
612
613     SetCursor(*((*gAnimCursHdl)->frame[(*gAnimCursHdl)->whichFrame++]));
614     if((*gAnimCursHdl)->whichFrame == (*gAnimCursHdl)->numberOfFrames)
615       (*gAnimCursHdl)->whichFrame = 0;
616
617     vblRecPtr->vblTaskRec.vblCount = gVBLCount;
618
619     currentA5 = SetA5(currentA5);
620   }
621
622   // ################################################################################ doStopAnimCursor
623
624   void  doStopAnimCursor(void)
625   {
626     SInt16    a;
627     Rect      theRect;
628
629     VRemove((QElemPtr) &gVBLRec.vblTaskRec);
630
631     for(a=0;a<(*gAnimCursHdl)->numberOfFrames;a++)
632       ReleaseResource((Handle) (*gAnimCursHdl)->frame[a]);
633
634     ReleaseResource((Handle) gAnimCursHdl);
635
636     gAnimatedCursorActive = false;
637     gSleepTime = MAXLONG;
638
639     SetCursor(&qd.arrow);
640
641     SetRect(&theRect,30,100,150,130);
642     RGBBackColor(&gWindowColour);
643     FillRect(&theRect,&qd.white);
644   }
645
646   // ##############################################################################################
647
648   // ##############################################################################################
649   // CDEF1.c                    Custom control definition function for radio button control
```

```
// ##############################################################################################
//
// This CDEF displays:
//
// •  3D coloured radio buttons on colour displays set to pixel depths greater than 2.
//
// •  Black-and-white buttons on colour displays set to pixel depths less than 4.
//
// •  Black-and-white radio buttons if Color QuickDraw is not present.
//
// This CDEF utilises six 'cicn' resources (purgeable).  The bitmap component, as opposed
// to the pixel map component, of each of these resources is automatically utilised if
// the icon is being displayed on a colour device for which the pixel depth has been set
// to 1 or 2.  The appearance of the coloured radio buttons conforms to the specification
// for radio button controls contained in the document Apple Greyscale Appearance for
// System 7.5 published by Apple Computer Inc.
//
// ##############################################################################################

// ……… includes

#include <Gestalt.h>
#include <ToolUtils.h>
#include <A4Stuff.h>
#include <Fonts.h>
#include <Palettes.h>
#include <LowMem.h>

// ……… defines

#define rActiveDeselect   128
#define rActiveSelect     129
#define rInActiveDeselect 130
#define rInactiveSelect   131
#define rFeedbackDeselect 132
#define rFeedbackSelect   133
#define partCode          1

// ……… global variables

QDGlobals   *gQDGlobalsPtr;
Boolean     gColorQuickDrawPresent = false;
Boolean     gColourDisplay         = false;
CIconHandle gActiveDeselectHdl     = NULL;
CIconHandle gActiveSelectHdl       = NULL;
CIconHandle gInactiveDeselectHdl   = NULL;
CIconHandle gInactiveSelectHdl     = NULL;
CIconHandle gFeedbackDeselectHdl   = NULL;
CIconHandle gFeedbackSelectHdl     = NULL;

// ……… function prototypes

pascal SInt32  main (short varCode,ControlHandle controlHdl,short message,SInt32 param);

void    doInitMessage   (void);
void    doDrawMessage   (ControlHandle);
void    drawColour      (ControlHandle,SInt16,Rect);
void    drawMono        (ControlHandle,SInt16,Rect);
SInt32  doTestMessage   (ControlHandle,SInt32);

// ########################################################################## main

pascal SInt32  main(short varCode,ControlHandle controlHdl,short message,SInt32 param)
{
  SInt32  returnValue;

  EnterCodeResource();

  switch(message)
  {
    case initCntl:
      doInitMessage();
      break;

    case drawCntl:
      if((*controlHdl)->contrlVis)
      {
```

```
727           doDrawMessage(controlHdl);
728         }
729         returnValue = 0;
730         break;
731
732       case testCntl:
733         returnValue = doTestMessage(controlHdl,param);
734         break;
735
736       default:
737         returnValue = 0;
738     }
739
740     ExitCodeResource();
741     return returnValue;
742   }
743
744   // ############################################################################## doInitMessage
745
746   void  doInitMessage(void)
747   {
748     OSErr     osErr;
749     SInt32    response;
750
751     gQDGlobalsPtr = (QDGlobals*) (*((SInt32*) SetCurrentA5()) - (sizeof(QDGlobals) -
752                                                                  sizeof(GrafPtr)));
753     osErr = Gestalt(gestaltQuickdrawVersion,&response);
754     if(response >= gestalt8BitQD)
755       gColorQuickDrawPresent = true;
756   }
757
758   // ############################################################################## doDrawMessage
759
760   void  doDrawMessage(ControlHandle controlHdl)
761   {
762     WindowPtr windowPtr;
763     GrafPtr   oldPort;
764     SInt16    oldFont, oldSize, oldTextMode, controlValue, fontNum;
765     PenState  oldPenState;
766     Rect      controlRect;
767     GDHandle  mainDeviceHdl;
768     SInt16    bitsPerPixel;
769
770     windowPtr = (WindowPtr) (*controlHdl)->contrlOwner;
771
772     GetPort(&oldPort);
773     oldFont = windowPtr->txFont;
774     oldSize = windowPtr->txSize;
775     oldTextMode = windowPtr->txMode;
776     GetPenState(&oldPenState);
777
778     SetPort(windowPtr);
779
780     controlValue = GetControlValue(controlHdl);
781
782     controlRect = (*controlHdl)->contrlRect;
783     controlRect.right = controlRect.left + 12;
784     controlRect.bottom = controlRect.top + 12;
785
786     GetFNum("\pChicago",&fontNum);
787     TextFont(fontNum);
788     TextSize(12);
789
790     if(gColorQuickDrawPresent)
791     {
792       if(gActiveDeselectHdl == NULL)
793         gActiveDeselectHdl = GetCIcon(rActiveDeselect);
794       if(gActiveSelectHdl == NULL)
795         gActiveSelectHdl = GetCIcon(rActiveSelect);
796       if(gInactiveDeselectHdl == NULL)
797         gInactiveDeselectHdl = GetCIcon(rInActiveDeselect);
798       if(gInactiveSelectHdl == NULL)
799         gInactiveSelectHdl = GetCIcon(rInactiveSelect);
800       if(gFeedbackDeselectHdl == NULL)
801         gFeedbackDeselectHdl = GetCIcon(rFeedbackDeselect);
802       if(gFeedbackSelectHdl == NULL)
803         gFeedbackSelectHdl = GetCIcon(rFeedbackSelect);
```

```
804     }
805
806     mainDeviceHdl = LMGetMainDevice();
807     bitsPerPixel = (*(*mainDeviceHdl)->gdPMap)->pixelSize;
808     if(bitsPerPixel > 1)
809       gColourDisplay = true;
810     else
811       gColourDisplay = false;
812
813     if(gColorQuickDrawPresent && gColourDisplay)
814       drawColour(controlHdl,controlValue,controlRect);
815     else
816       drawMono(controlHdl,controlValue,controlRect);
817
818     SetPenState(&oldPenState);
819     TextMode(oldTextMode);
820     TextFont(oldFont);
821     TextSize(oldSize);
822     SetPort(oldPort);
823   }
824
825   // ############################################################################## drawColour
826
827   void  drawColour(ControlHandle controlHdl,SInt16 controlValue, Rect controlRect)
828   {
829     RGBColor  oldForeColour;
830     RGBColor  blackColour   = { 0x0000, 0x0000, 0x0000 };
831     RGBColor  gray8         = { 0x7777, 0x7777, 0x7777 };
832
833     GetForeColor(&oldForeColour);
834
835     if((*controlHdl)->contrlHilite == 255)
836     {
837       if(gColourDisplay)
838         RGBForeColor(&gray8);
839       else
840         TextMode(grayishTextOr);
841
842       if(controlValue == 1)
843         PlotCIcon(&controlRect,gInactiveSelectHdl);
844       else
845         PlotCIcon(&controlRect,gInactiveDeselectHdl);
846     }
847     else if((*controlHdl)->contrlHilite == 0)
848     {
849       RGBForeColor(&blackColour);
850       TextMode(srcOr);
851
852       if(controlValue == 1)
853         PlotCIcon(&controlRect,gActiveSelectHdl);
854       else
855         PlotCIcon(&controlRect,gActiveDeselectHdl);
856     }
857     else if((*controlHdl)->contrlHilite == partCode)
858     {
859       if(controlValue == 1)
860         PlotCIcon(&controlRect,gFeedbackSelectHdl);
861       else
862         PlotCIcon(&controlRect,gFeedbackDeselectHdl);
863     }
864
865     MoveTo(controlRect.left + 17,controlRect.top + 10);
866     DrawString((*controlHdl)->contrlTitle);
867
868     RGBForeColor(&oldForeColour);
869   }
870
871   // ############################################################################## drawMono
872
873   void  drawMono(ControlHandle controlHdl,SInt16 controlValue, Rect controlRect)
874   {
875     ForeColor(blackColor);
876     BackColor(whiteColor);
877
878     PenNormal();
879
880     if((*controlHdl)->contrlHilite == 255 || (*controlHdl)->contrlHilite == 0)
```

```
881     {
882       if((*controlHdl)->contrlHilite == 255)
883       {
884         PenPat(&gQDGlobalsPtr->gray);
885         TextMode(grayishTextOr);
886       }
887       else
888       {
889         PenPat(&gQDGlobalsPtr->black);
890         TextMode(srcOr);
891       }
892
893       FrameOval(&controlRect);
894
895       InsetRect(&controlRect,1,1);
896       FillOval(&controlRect,&gQDGlobalsPtr->white);
897       InsetRect(&controlRect,-1,-1);
898
899       if(controlValue == 1)
900       {
901         InsetRect(&controlRect,3,3);
902         if((*controlHdl)->contrlHilite == 255)
903           FillOval(&controlRect,&gQDGlobalsPtr->gray);
904         else
905           FillOval(&controlRect,&gQDGlobalsPtr->black);
906         InsetRect(&controlRect,-3,-3);
907       }
908     }
909     else if((*controlHdl)->contrlHilite == partCode)
910     {
911       InsetRect(&controlRect,1,1);
912       FrameOval(&controlRect);
913       InsetRect(&controlRect,-1,-1);
914     }
915
916     MoveTo(controlRect.left + 17,controlRect.top + 10);
917     DrawString((*controlHdl)->contrlTitle);
918   }
919
920   // ########################################################################## doTestMessage
921
922   SInt32  doTestMessage(ControlHandle controlHdl,SInt32 param)
923   {
924     Rect  controlRect;
925     Point mouseXY;
926
927     controlRect = (*controlHdl)->contrlRect;
928
929     mouseXY.v = HiWord(param);
930     mouseXY.h = LoWord(param);
931
932     if(PtInRect(mouseXY,&controlRect))
933       return partCode;
934     else
935       return (SInt32) 0;
936   }
937
938   // ############################################################################################
939
940   // ############################################################################################
941   // CDEF2.c                          Custom control definition function for slider control
942   // ############################################################################################
943   //
944   // This CDEF displays:
945   //
946   // •  A 3D coloured slider control on colour displays set to pixel depths greater than 1.
947   //
948   // •  A black-and-white slider control on colour displays set to pixel depths less than
949   //    2.
950   //
951   // •  A black-and-white slider control if Color QuickDraw is not present.
952   //
953   // The CDEF utilises two 'PICT' resources (purgeable).  One resource contains the colour
954   // version of the slider control components.  The other comprises the black and white
955   // version.  The appearance of the coloured slider conforms to the specification for
956   // slider controls contained in the document Apple Greyscale Appearance for System 7.5
957   // published by Apple Computer Inc.
```

```
//
// ##############################################################################################

// ……… includes

#include <ToolUtils.h>
#include <QDOffscreen.h>
#include <Resources.h>
#include <Gestalt.h>
#include <Fonts.h>
#include <Retrace.h>
#include <Traps.h>
#include <Devices.h>
#include <A4Stuff.h>
#include <LowMem.h>

// ……… defines

#define kInactive         255
#define kIndicatorHeight  16
#define rTrackPict        128

// ……… typedefs

typedef struct
{
  VBLTask   vblTaskRec;
  Boolean   inVBlankPeriod;
  SInt32    thisApplicationsA5;
} VBLRec, *VBLRecPtr;

typedef struct
{
  GWorldPtr offScreenPort;
  Rect      offScreenPortRect;
  Rect      trackActiveRect;
  Rect      trackInactiveRect;
  Rect      indicatorActiveRect;
  Rect      indicatorPressedRect;
  Rect      indicatorInactiveRect;
  Rect      compositeRect;
  GWorldPtr currentPort;
  GDHandle  currentDevice;
} SliderDataRec, *SliderDataPtr, **SliderDataHdl;

// ……… global variables

Boolean   gColorQuickDrawPresent = true;
SInt16    gMainSlotNumber;
Boolean   gSlotVInstallPresent;
Boolean   gDragMessageFlag       = false;
Boolean   gVBLInstallFail        = true;
VBLRec    gVBLRec;

// ……… function prototypes

pascal SInt32   main(SInt16 varCode,ControlHandle theControl,SInt16 message,SInt32 param);

void      doInitMessage              (ControlHandle);
void      doDrawMessage              (ControlHandle);
SInt32    doTestMessage              (ControlHandle,SInt32);
SInt32    doDragMessage              (ControlHandle);
void      doDisposeMessage           (ControlHandle);
void      createOffScreenGWorld      (ControlHandle);
void      pixelDepthCheck            (ControlHandle);
void      drawControlActive          (ControlHandle);
void      drawControlInactive        (ControlHandle);
Rect      calcIndicatorRect          (ControlHandle);
OSErr     installVBLTask             (void);
void      removeVBLTask              (void);
void      theVBLTask                 (void);
Boolean   checkSlotVInstallAvailable (void);
Boolean   checkTrapAvailable         (SInt16);

// ……… in-line glue for GetVBLRec

pascal SInt32  GetVBLRec (void) = 0x2E88;
```

```
1035
1036  // ############################################################################## main
1037
1038  pascal SInt32  main(SInt16 varCode,ControlHandle theControl,SInt16 message,SInt32 param)
1039  {
1040    PenState  oldPenState;
1041    SInt32    returnValue;
1042
1043    EnterCodeResource();
1044
1045    GetPenState(&oldPenState);
1046
1047    switch(message)
1048    {
1049      case initCntl:
1050        doInitMessage(theControl);
1051        returnValue = 0;
1052        break;
1053
1054      case drawCntl:
1055        if((*theControl)->contrlVis)
1056          doDrawMessage(theControl);
1057        returnValue = 0;
1058        break;
1059
1060      case testCntl:
1061        returnValue = doTestMessage(theControl,param);
1062        break;
1063
1064      case dragCntl:
1065        returnValue = doDragMessage(theControl);
1066        break;
1067
1068      case dispCntl:
1069        doDisposeMessage(theControl);
1070        returnValue = 0;
1071        break;
1072
1073      default:
1074        returnValue = 0;
1075    }
1076
1077    SetPenState(&oldPenState);
1078
1079    ExitCodeResource();
1080
1081    return returnValue;
1082  }
1083
1084  // ########################################################################## doInitMessage
1085
1086  void  doInitMessage(ControlHandle theControl)
1087  {
1088    OSErr       osErr;
1089    SInt32      response;
1090    GDHandle    mainDeviceHdl;
1091    SInt16      mainDeviceRefNum;
1092    DCtlHandle  deviceCtlEntryHdl;
1093
1094    osErr = Gestalt(gestaltQuickdrawVersion,&response);
1095    if(response < gestalt8BitQD)
1096      gColorQuickDrawPresent = false;
1097
1098    HLock((Handle) theControl);
1099
1100    (*theControl)->contrlData = NewHandle(sizeof(SliderDataRec));
1101    if((*theControl)->contrlData != NULL)
1102      createOffScreenGWorld(theControl);
1103
1104    HUnlock((Handle) theControl);
1105
1106    gSlotVInstallPresent = checkSlotVInstallAvailable();
1107    if(gSlotVInstallPresent)
1108    {
1109      mainDeviceHdl     = LMGetMainDevice();
1110      mainDeviceRefNum  = (*mainDeviceHdl)->gdRefNum;
1111      deviceCtlEntryHdl = GetDCtlEntry(mainDeviceRefNum);
```

```
1112      gMainSlotNumber   = (SInt16) (*((AuxDCEHandle) deviceCtlEntryHdl))->dCtlSlot;
1113    }
1114  }
1115
1116  // ########################################################################## doDrawMessage
1117
1118  void  doDrawMessage(ControlHandle theControl)
1119  {
1120    if(gColorQuickDrawPresent)
1121      pixelDepthCheck(theControl);
1122
1123    if((*theControl)->contrlHilite == kInactive)
1124      drawControlInactive(theControl);
1125    else
1126      drawControlActive(theControl);
1127  }
1128
1129  // ########################################################################## doTestMessage
1130
1131  SInt32  doTestMessage(ControlHandle theControl,SInt32 param)
1132  {
1133    Rect  indicatorRect;
1134    Point mouseXY;
1135
1136    indicatorRect = calcIndicatorRect(theControl);
1137
1138    mouseXY.v = HiWord(param);
1139    mouseXY.h = LoWord(param);
1140
1141    if(PtInRect(mouseXY,&indicatorRect))
1142    {
1143      gDragMessageFlag = true;
1144      drawControlActive(theControl);
1145      gDragMessageFlag = false;
1146      return kControlIndicatorPart;
1147    }
1148    else
1149      return 0;
1150  }
1151
1152  // ########################################################################## doDragMessage
1153
1154  SInt32  doDragMessage(ControlHandle theControl)
1155  {
1156    Rect      indicatorRect, slopRect, trackRect;
1157    SInt16    indicatorHeight, indicatorHalfHeight, indicatorCentre, trackHeight;
1158    Point     startMouseXY, currentMouseXY;
1159    SInt16    controlValueRange, differenceMouseY;
1160    float     ratio;
1161    WindowPtr windowPtr;
1162    OSErr     osErr;
1163
1164    gDragMessageFlag = true;
1165
1166    HLock((Handle) theControl);
1167
1168    indicatorHeight = kIndicatorHeight;
1169    indicatorHalfHeight = indicatorHeight / 2;
1170
1171    trackRect = (*theControl)->contrlRect;
1172    InsetRect(&trackRect,0,indicatorHalfHeight + 4);
1173    trackRect.bottom += 1;
1174    trackHeight = trackRect.bottom - trackRect.top;
1175
1176    controlValueRange = (*theControl)->contrlMax - (*theControl)->contrlMin;
1177    ratio = (float) ((float) controlValueRange) / ((float) trackHeight);
1178
1179    windowPtr = (*theControl)->contrlOwner;
1180    slopRect = windowPtr->portRect;
1181
1182    osErr = installVBLTask();
1183    if(osErr == noErr)
1184      gVBLInstallFail = false;
1185    else
1186      gVBLInstallFail = true;
1187
1188    indicatorRect = calcIndicatorRect(theControl);
```

```
1189
1190    GetMouse(&startMouseXY);
1191
1192    while(StillDown())
1193    {
1194      GetMouse(&currentMouseXY);
1195      differenceMouseY = startMouseXY.v - currentMouseXY.v;
1196
1197      if(differenceMouseY != 0 && PtInRect(currentMouseXY,&slopRect))
1198      {
1199        indicatorRect.top -= differenceMouseY;
1200        indicatorRect.bottom -= differenceMouseY;
1201
1202        indicatorCentre = indicatorRect.top + indicatorHalfHeight;
1203
1204        (*theControl)->contrlValue = (trackRect.bottom - indicatorCentre) * ratio;
1205
1206        if((*theControl)->contrlValue > (*theControl)->contrlMax)
1207          (*theControl)->contrlValue = (*theControl)->contrlMax;
1208        if((*theControl)->contrlValue < (*theControl)->contrlMin)
1209          (*theControl)->contrlValue = (*theControl)->contrlMin;
1210
1211        drawControlActive(theControl);
1212
1213        startMouseXY = currentMouseXY;
1214      }
1215    }
1216
1217    if(!gVBLInstallFail)
1218      removeVBLTask();
1219
1220    gDragMessageFlag = false;
1221    drawControlActive(theControl);
1222
1223    HUnlock((Handle) theControl);
1224
1225    return 1;
1226  }
1227
1228  // ########################################################################## doDisposeMessage
1229
1230  void  doDisposeMessage(ControlHandle theControl)
1231  {
1232    Rect  theRect;
1233
1234    theRect = (*theControl)->contrlRect;
1235    theRect.right = theRect.right + (theRect.right - theRect.left);
1236    EraseRect(&theRect);
1237
1238    if((**((SliderDataHdl) (*theControl)->contrlData)).offScreenPort != NULL)
1239      DisposeGWorld((**((SliderDataHdl) (*theControl)->contrlData)).offScreenPort);
1240
1241    if((*theControl)->contrlData != NULL)
1242      DisposeHandle((*theControl)->contrlData);
1243  }
1244
1245  // ############################################################### createOffScreenGWorld
1246
1247  void  createOffScreenGWorld(ControlHandle theControl)
1248  {
1249    SliderDataHdl sliderDataHdl;
1250    SInt16        resourceOffset = 0;
1251    PixMapHandle  pixMapHdl;
1252    PicHandle     pictureHdl;
1253    SInt16        currentPortDepth = 1;
1254
1255    sliderDataHdl = (SliderDataHdl) (*theControl)->contrlData;
1256
1257    (*sliderDataHdl)->compositeRect = (*theControl)->contrlRect;
1258    OffsetRect(&(*sliderDataHdl)->compositeRect,-(*sliderDataHdl)->compositeRect.left,
1259               -(*sliderDataHdl)->compositeRect.top);
1260    SetRect(&(*sliderDataHdl)->trackActiveRect,50,0,100,139);
1261    SetRect(&(*sliderDataHdl)->trackInactiveRect,100,0,150,139);
1262    SetRect(&(*sliderDataHdl)->indicatorActiveRect,0,139,16,154);
1263    SetRect(&(*sliderDataHdl)->indicatorPressedRect,16,139,32,154);
1264    SetRect(&(*sliderDataHdl)->indicatorInactiveRect,32,139,48,154);
1265    SetRect(&(*sliderDataHdl)->offScreenPortRect,0,0,150,154);
```

```
1266
1267    GetGWorld(&(*sliderDataHdl)->currentPort,&(*sliderDataHdl)->currentDevice);
1268
1269    NewGWorld(&(*sliderDataHdl)->offScreenPort,0,&(*sliderDataHdl)->offScreenPortRect,
1270              NULL,NULL,0);
1271
1272    HLock((Handle) sliderDataHdl);
1273
1274    pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->offScreenPort);
1275    LockPixels(pixMapHdl);
1276
1277    SetGWorld((*sliderDataHdl)->offScreenPort,nil);
1278
1279    EraseRect(&(*sliderDataHdl)->offScreenPortRect);
1280
1281    if(gColorQuickDrawPresent)
1282    {
1283      pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->currentPort);
1284      currentPortDepth = (*pixMapHdl)->pixelSize;
1285    }
1286
1287    if(!gColorQuickDrawPresent || currentPortDepth < 2)
1288      resourceOffset = 1;
1289
1290    if(pictureHdl = GetPicture(rTrackPict + resourceOffset))
1291    {
1292      HNoPurge((Handle) pictureHdl);
1293      DrawPicture(pictureHdl,&(*sliderDataHdl)->offScreenPortRect);
1294      HPurge((Handle) pictureHdl);
1295    }
1296
1297    SetGWorld((*sliderDataHdl)->currentPort,(*sliderDataHdl)->currentDevice);
1298    UnlockPixels(pixMapHdl);
1299    HUnlock((Handle) sliderDataHdl);
1300  }
1301
1302  // ########################################################################## pixelDepthCheck
1303
1304  void  pixelDepthCheck(ControlHandle theControl)
1305  {
1306    SliderDataHdl sliderDataHdl;
1307    PixMapHandle  pixMapHdl;
1308    SInt16        currentPortDepth, gworldPortDepth;
1309
1310    sliderDataHdl = (SliderDataHdl) (*theControl)->contrlData;
1311
1312    pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->currentPort);
1313    currentPortDepth = (*pixMapHdl)->pixelSize;
1314    pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->offScreenPort);
1315    gworldPortDepth = (*pixMapHdl)->pixelSize;
1316
1317    if(currentPortDepth != gworldPortDepth)
1318    {
1319      DisposeGWorld((*sliderDataHdl)->offScreenPort);
1320      createOffScreenGWorld(theControl);
1321    }
1322  }
1323
1324  // ########################################################################## drawControlActive
1325
1326  void  drawControlActive(ControlHandle theControl)
1327  {
1328    RGBColor      oldForeColour, oldBackColour;
1329    SliderDataHdl sliderDataHdl;
1330    WindowPtr     windowPtr;
1331    PixMapHandle  pixMapHdl;
1332    Rect          indicatorRect;
1333
1334    GetForeColor(&oldForeColour);
1335    GetBackColor(&oldBackColour);
1336
1337    HLock((Handle) theControl);
1338
1339    sliderDataHdl = (SliderDataHdl) (*theControl)->contrlData;
1340    HLock((Handle) sliderDataHdl);
1341
1342    windowPtr = (WindowPtr) (*theControl)->contrlOwner;
```

```
1343    SetPort(windowPtr);
1344
1345    pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->offScreenPort);
1346    LockPixels(pixMapHdl);
1347
1348    ForeColor(blackColor);
1349    BackColor(whiteColor);
1350
1351    CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1352             &((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1353             &(*sliderDataHdl)->trackActiveRect,&(*sliderDataHdl)->compositeRect,
1354             srcCopy,NULL);
1355
1356    indicatorRect = calcIndicatorRect(theControl);
1357    OffsetRect(&indicatorRect,-(*theControl)->contrlRect.left,
1358               -(*theControl)->contrlRect.top);
1359
1360    if(gDragMessageFlag)
1361    {
1362      CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1363               &((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1364               &(*sliderDataHdl)->indicatorPressedRect,&indicatorRect,srcCopy,NULL);
1365    }
1366    else
1367    {
1368      CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1369               &((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1370               &(*sliderDataHdl)->indicatorActiveRect,&indicatorRect,srcCopy,NULL);
1371    }
1372
1373    if(gDragMessageFlag && !gVBLInstallFail)
1374    {
1375      if(gVBLRec.inVBlankPeriod)
1376      {
1377        gVBLRec.inVBlankPeriod = false;
1378
1379        CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1380                 &((GrafPtr) windowPtr)->portBits,&(*sliderDataHdl)->compositeRect,
1381                 &(*theControl)->contrlRect,srcCopy,NULL);
1382      }
1383    }
1384    else
1385    {
1386      CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
1387               &((GrafPtr) windowPtr)->portBits,&(*sliderDataHdl)->compositeRect,
1388               &(*theControl)->contrlRect,srcCopy,NULL);
1389    }
1390
1391    UnlockPixels(pixMapHdl);
1392    HUnlock((Handle) sliderDataHdl);
1393    HUnlock((Handle) theControl);
1394
1395    RGBForeColor(&oldForeColour);
1396    RGBBackColor(&oldBackColour);
1397  }
1398
1399  // ##################################################################### drawControlInactive
1400
1401  void drawControlInactive(ControlHandle theControl)
1402  {
1403    RGBColor      oldForeColour, oldBackColour;
1404    SliderDataHdl sliderDataHdl;
1405    WindowPtr     windowPtr;
1406    PixMapHandle  pixMapHdl;
1407    Rect          indicatorRect;
1408
1409    GetForeColor(&oldForeColour);
1410    GetBackColor(&oldBackColour);
1411
1412    HLock((Handle) theControl);
1413
1414    sliderDataHdl = (SliderDataHdl) (*theControl)->contrlData;
1415    HLock((Handle) sliderDataHdl);
1416
1417    windowPtr = (WindowPtr) (*theControl)->contrlOwner;
1418    SetPort(windowPtr);
1419
```

```
  pixMapHdl = GetGWorldPixMap((*sliderDataHdl)->offScreenPort);
  LockPixels(pixMapHdl);

  ForeColor(blackColor);
  BackColor(whiteColor);

  CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
           &((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
           &(*sliderDataHdl)->trackInactiveRect,&(*sliderDataHdl)->compositeRect,
           srcCopy,nil);

  indicatorRect = calcIndicatorRect(theControl);
  OffsetRect(&indicatorRect,-(*theControl)->contrlRect.left,
             -(*theControl)->contrlRect.top);

  CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
           &((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
           &(*sliderDataHdl)->indicatorInactiveRect,&indicatorRect,srcCopy,nil);

  CopyBits(&((GrafPtr) ((*sliderDataHdl)->offScreenPort))->portBits,
           &((GrafPtr) windowPtr)->portBits,&(*sliderDataHdl)->compositeRect,
           &(*theControl)->contrlRect,srcCopy,nil);

  UnlockPixels(pixMapHdl);
  HUnlock((Handle) sliderDataHdl);
  HUnlock((Handle) theControl);

  RGBForeColor(&oldForeColour);
  RGBBackColor(&oldBackColour);
}

// ########################################################################## calcIndicatorRect

Rect  calcIndicatorRect(ControlHandle theControl)
{
  SInt16  indicatorHeight, indicatorHalfHeight;
  Rect    trackRect, indicatorRect;
  SInt16  trackHeight, controlValue, controlMax, controlMin, indicatorCentre;
  float   ratio;

  indicatorHeight = kIndicatorHeight;
  indicatorHalfHeight = indicatorHeight / 2;

  trackRect = (*theControl)->contrlRect;
  InsetRect(&trackRect,0,indicatorHalfHeight + 4);
  trackRect.bottom += 1;
  trackHeight = trackRect.bottom - trackRect.top;

  controlValue = (*theControl)->contrlValue;
  controlMax = (*theControl)->contrlMax;
  controlMin = (*theControl)->contrlMin;

  ratio = ((float) controlValue) / ((float) (controlMax - controlMin));
  indicatorCentre = trackRect.bottom - (SInt16) (ratio * trackHeight) ;

  SetRect(&indicatorRect,trackRect.left,indicatorCentre - indicatorHalfHeight,
          trackRect.left + 16,indicatorCentre + indicatorHalfHeight - 1);

  return indicatorRect;
}

// ############################################################################ installVBLTask

OSErr installVBLTask(void)
{
  OSErr osErr;

  gVBLRec.inVBlankPeriod = false;

  gVBLRec.vblTaskRec.qType    = vType;
  gVBLRec.vblTaskRec.vblAddr  = (VBLUPP) theVBLTask;
  gVBLRec.vblTaskRec.vblCount = 1;
  gVBLRec.vblTaskRec.vblPhase = 0;

  gVBLRec.thisApplicationsA5 = SetCurrentA5();

  if(gSlotVInstallPresent)
```

```
1497      osErr = SlotVInstall((QElemPtr) &gVBLRec.vblTaskRec,gMainSlotNumber);
1498    else
1499      osErr = VInstall((QElemPtr) &gVBLRec.vblTaskRec);
1500
1501    return osErr;
1502  }
1503
1504  // ########################################################################## removeVBLTask
1505
1506  void  removeVBLTask(void)
1507  {
1508    if(gSlotVInstallPresent)
1509      SlotVRemove((QElemPtr) &gVBLRec.vblTaskRec,gMainSlotNumber);
1510    else
1511      VRemove((QElemPtr) &gVBLRec.vblTaskRec);
1512  }
1513
1514  // ############################################################################## theVBLTask
1515
1516  void theVBLTask(void)
1517  {
1518    VBLRecPtr vblRecPtr;
1519    SInt32    currentA5;
1520
1521    vblRecPtr = (VBLRecPtr) GetVBLRec();
1522    currentA5 = SetA5(vblRecPtr->thisApplicationsA5);
1523
1524    vblRecPtr->inVBlankPeriod = true;
1525    vblRecPtr->vblTaskRec.vblCount = 1;
1526
1527    (void) SetA5(currentA5);
1528  }
1529
1530  // ############################################################ checkSlotVInstallAvailable
1531
1532  Boolean  checkSlotVInstallAvailable(void)
1533  {
1534    return checkTrapAvailable(_SlotVInstall);
1535  }
1536
1537  // ################################################################### checkTrapAvailable
1538
1539  Boolean  checkTrapAvailable(SInt16 theTrap)
1540  {
1541    TrapType  trapType;
1542    SInt16    trapMask = 0x0800;
1543    SInt16    numToolboxTraps;
1544
1545    if((theTrap & trapMask) > 0)
1546      trapType = ToolTrap;
1547    else
1548      trapType = OSTrap;
1549
1550    if(trapType == ToolTrap)
1551      theTrap = theTrap & 0x07FF;
1552
1553    if(NGetTrapAddress(_InitGraf,ToolTrap) == NGetTrapAddress(0xAA6E,ToolTrap))
1554      numToolboxTraps = 0x0200;
1555    else
1556      numToolboxTraps = 0x0400;
1557
1558    if(theTrap >= numToolboxTraps)
1559      theTrap = _Unimplemented;
1560
1561    return(NGetTrapAddress(theTrap,trapType) != NGetTrapAddress(_Unimplemented,ToolTrap));
1562  }
1563
1564  // ##############################################################################################
```

Demonstration Program Comments

When this program is run, the user should:

- Click the Start radio button in the slider control panel and operate the slider control by dragging the indicator.

- On colour displays, observe the appearance of the controls when the pixel depth of the device is set (using the Monitors Control Panel) to pixel depths of 1 (black and white), 2 (four colours), and 4 and greater (16 colours and greater).

- Observe the appearance of the controls when the program is sent to the background.

The user should also choose VBL Task Animated Cursor from the Demonstration menu to view the animated cursor. Note that, in this demonstration program, the slider control can be operated while the animated cursor is active, something that would be illogical in a real application. This is allowed in this demonstration only to invoke the concurrent operation of two VBL tasks (incrementing the cursor's frames and synchronising the drawing of the moving indicator with the screen refresh cycle).

Note that the simple indicator image used by the slider control animates quite smoothly without the assistance of the VBL task. That part of the CDEF source code may thus be regarded as being for VBL task illustrative purposes only.

Also note that animating a cursor using a VBL task, as opposed to the method described at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons, is not recommended. If the application "locks up" after the animation is initiated, the Chapter 12 method will probably result in the cessation of the animation, whereas the VBL task method will almost certainly leave the cursor "spinning". This latter will mislead the user into believing that the application's time-consuming task is still in progress.

CDEFandVBL.c

#define

Lines 58-63 establish constants relating to menu IDs and menu item numbers. Lines 64-69 establish constants for various resource IDs. Line 70 defines MAXLONG as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

#typedef

Lines 74-79 define a data type which will be used by the animated cursor functions. Lines 81-85 define a data type which will be used by the animated cursor VBL task functions.

Global Variables

gColorQuickDraw will be set to true if Color QuickDraw is present. gColourDisplay will be set to true if the pixel depth of the main device is greater than 1. gDone controls program termination. gSleepTime controls the value passed in the sleep parameter of the WaitNextEvent call. gInBackground relates to foreground/background switching. gWindowPtr will be assigned the pointer to single window opened by the program.

Lines 95-96 declare variables of the two types defined at Lines 74-85. gVBLCount controls the value assigned to the vblCount field of the VBL task record. gAnimatedCursorActive is a flag which indicates whether or not the animated cursor is currently active. The colour assigned to gWindowColour will be used to erase certain areas of the window.

The three global variables at Lines 100-102 will be assigned handles to the custom slider and radio button controls.

main

The main function firstly initialises the system software managers (Line 142). At Lines 146-155, the global variable gColorQuickDrawPresent is set to true if Color QuickDraw is present and, if Color QuickDraw is present, gColourDisplay is set to true if the pixel depth of the main device is greater than 1. The menus are set up at Lines 159-169), a window is opened (Lines 173-184), and an application-defined function is called to get the three controls (Line 188. The main event loop is then entered with gSleepTime set to MAXLONG (Lines 192-199).

doEvents, doMouseDown, doUpdate, doActivate, doActivateWindow, and doOSEvent

doEvents, doMouseDown, doUpdate, doActivate, doActivateWindow, and doOSEvent perform minimal event handling consistent with the requirements of the demonstration.

Note the calls to doDrawControlPanel at Lines 300 and 339. Note also, at Lines 327-328, that the slider control is only highlighted when the window is becoming active if the Start radio button's control value is 1.

doInContent

doInContent further handles mouse-down events which occur within the content region of the window.

Line 367 converts the mouse-down location to local coordinates. The call to FindControl at Line 369 determines whether the mouse-down was within a control and, if so, which particular control and, where relevant, which part of that control.

If the mouse-down was within the slider control (Line 371), and if the part code was the indicator (Line 373), TrackControl is called (Line 374) to take control while the mouse button remains down. As will be seen, the custom CDEF for this custom control uses custom dragging for the indicator; accordingly, TrackControl will invariably return 0 when the mouse button is released. When the mouse button is released, Lines 376-382 erase a small rectangle to the right of the slider control and draw the slider control's value at that location.

If the mouse-down was within either of the radio buttons (Line 384), TrackControl is called at Line 386 to take control until the mouse button is released. If TrackControl returns a non-zero value, and if the control was the Start radio button (Line 388), Line 390 makes the slider control active, and Lines 391-392 toggle the Start and Stop radio button control values. If the mouse-down was within the Stop radio button (Line 394), Line 396 sets the slider control's value to 0, Line 397 makes the slider control inactive, and Lines 398-399 toggle the Start and Stop radio button control values.

Note that, when Line 374 first detects a movement of the mouse, a dragCntl message will be sent to the CDEF. The CDEF will respond by telling the Control Manager that custom dragging is being used, meaning that the CDEF will be following the mouse and updating the indicator position and control value until the button is released.

doMenuChoice

doMenuChoice handles menu choices.

Note that DisposeWindow (Line 438) automatically calls KillControls. A call to KillControls will cause CDEFs to be sent a dispCntl message, providing them with the opportunity to perform any necessary disposal actions.

doGetSliderControlSuite

doGetSliderControlSuite gets the three controls which comprise the slider control panel.

Line 456 gets the custom slider control. Line 457 sets the initial state of the control to inactive. (Note: If autoTrack messages were required by the CDEF, it would also be necessary to set the contrlAction field of this control's control record to (ControlActionUPP) -1 as one of the two actions necessary to cause autoTrack messages to be sent to the CDEF, for example: SetControlAction(gSliderControlHdl,(ControlActionUPP) -1);.)

Lines 459-460 get the radio button controls and call an application-defined function which draws a titled box around the three controls in the suite.

doDrawControlsPanel

doDrawControlsPanel draws a titled box around the three controls in the suite, thereby visually defining the slider control panel. Colours/patterns for the line and text drawing are determined according to whether Color QuickDraw is present, the pixel depth of the main device if Color QuickDraw is present, and whether the program is in the foreground or the background at the time of the draw. The re-check of the main device's pixel depth at Lines 478-484 is simply to accommodate the possibility that the user may change the pixel depth while the program is running.

doStartAnimCursor

doStartAnimCursor is called when the user chooses VBL Task Animated Cursor from the Demonstration menu.

Line 545 sets the variable which will be used to assign a value to the vblCount field of the VBL task record. The value of 30 ensures that the VBL task will execute every 30 VBL interrupts, that is, about every half a second.

Line 546 sets the variable which is used as the sleep parameter in the WaitNextEvent call to a value less than that assigned to gVBLCount.

Line 548 calls an application-defined routine which loads the 'acur' resource, together with the 'CURS' resources specified in the 'acur' resource.

Line 551 calls an application-defined function which installs a system-based VBL task.

Line 553 sets a flag which indicates whether the animated cursor is currently active and Lines 555-558 draw some advisory text in the window.

doGetAnimCursor

doGetAnimCursor is identical to the function of the same name in the demonstration program at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons. When it exits, the fields of a variable of type animCurs contain the number of frames (that is, the number of 68-byte Cursor structures) and the handles to those Cursor structures.

doInstallSystemVBLTask

doInstallSystemVBLTask installs the system-based VBL task.

Lines 593-596 fill in the relevant fields of the VBL task record. Note that the vblAddr field points to the application-defined function animCursVBLTask, and that the vblCount field is set to 30.

Line 598 gets the pointer to the application's A5 world and saves it to the appropriate field of a global variable.

Line 600 installs the specified VBL task into the system-based queue.

animCursVBLTask

animCursVBLTask is the VBL task, which will be executed every 30 VBL interrupts. Its purpose is the same as that of the similar function doSpinAnimCursor at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

Line 610 gets a pointer to the variable which contains the field holding the pointer to the application's A5 world. Line 611 saves the current A5 and, at the same time, sets the application's A5 world as that to be used for the duration of the VBL task's execution. The task can now access the application's global variables.

Lines 613-615 increment the animated cursor frame number.

When the task executes, the value in the vblCount field of the VBL task record will be set to 0. Line 617 resets this field to a value higher that 0 (in this case, 30), otherwise the task will never execute again.

Line 619 restores the save A5 world pointer.

doStopAnimCursor

doStopAnimCursor is called when the user presses any key.

Line 629 removes the VBL task from the queue. Lines 631-634 free up the memory occupied by the cursors. Line 636 resets the animated-cursor-active flag. Line 637 resets WaitNextEvent's sleep parameter. Line 639 sets the standard arrow cursor and Lines 641-643 erase the advisory text from the window.

CDEF1.c

CDEF1.c is the source code for the custom radio button control definition function (CDEF). The CDEF responds only to the initCntl, drawCntl, and testCntl messages.

#define

Lines 680-685 establish constants representing the resource IDs for six 'cicn' (colour icon) resources. Line 686 establishes a constant representing the part code returned in the doTestMessage function.

Global Variables

gQDGlobalsPtr will be assigned the address of the host application's QuickDraw globals. gColorQuickDrawPresent will be set to false if Color QuickDraw is not present. gColourDisplay will be set to true if the pixel depth of the main device is greater than 1. The global variables at Lines 693-698 will be assigned handles to the CIcon records for the six colour icons.

main

The main function receives the incoming message and switches accordingly, returning the appropriate value to the Control Manager.

Lines 716 and 740 have to do with accessing the CDEF's global variables. (See Setting up globals in 68K code resources in the Code Resource Projects/Writing Code Resources section of Chapter 2 (Creating Mac OS Projects) in the CodeWarrior manual Targeting Mac OS.)

Note that, in the case of a drawCntl message (Lines 724), no action is taken if the contrlVis field of the control's control record specifies that the control is currently invisible.

doInitMessage

doInitMessage handles initCntl messages to completion.

The function drawMono uses the QuickDraw globals gray, black, and white. This raises the question of how a code resource (such as this CDEF) can access the host application's QuickDraw globals. SetCurrentA5 is used to return the address of the current A5 world, and the globals are accessed as an offset from that (Lines 751-752).

Lines 753-755 check whether Color QuickDraw is present and set the global variable gColorQuickDrawPresent accordingly.

doDrawMessage

doDrawMessage performs the initial handling of drawCntl messages.

Line 770 gets a pointer to the graphics port of the window in which the control resides. Lines 772-776 save the current drawing environment.

Line 778 sets the graphics port and Line 780 obtains the current value of the control. Lines 782-784 establish a rectangle the required size of the radio button image and positioned at the top left of the control's rectangle. Lines 786-788 set the font to Chicago 12 point.

If Color QuickDraw is present (Line 790), Lines 792-803 load the colour icons if they are not currently in memory.

Lines 806-811 set the global variable gColourDisplay to true if the pixel depth of the main device is greater than 1. If Color QuickDraw is present and the pixel depth is greater than 1, the CDEF-defined function drawColour is then called, otherwise the CDEF-defined function drawMono is called (Lines 813-816).

Lines 818-822 restore the previously save drawing environment.

drawColour

drawColour draws the radio button and its title in a Color QuickDraw environment.

Line 833 saves the current foreground colour

If the control is inactive (Line 835), the following occurs. If the pixel depth of the main device is greater than 1, the foreground colour is set to medium grey colour, otherwise the text mode is set to grayishTextOr (Lines 837-840) Depending on the current control value, the appropriate inactive state colour icon is then drawn (Lines 842-845).

If the control is active (Line 847), the following occurs. The foreground colour is set to black and the text drawing mode is set to the default (Lines 849-850). Depending on the current control value, the appropriate active state colour icon is then drawn (Lines 852-855).

If the contrlHilite element of the control record contains partCode (see the function doTestMessage) (Line 857), the appropriate "mouse is currently down within the control rectangle" icon is drawn (Lines 859-862).

Lines 865-866 then draw the control's title at the appropriate location, following which the foreground colour saved at Line 833 is restored (Line 868).

drawMono

drawMono draws the radio button and its title in a non-Color QuickDraw environment.

Lines 875-876 set the foreground and background colours. Line 878 sets the pen size to 1,1, the pen mode to patCopy, and the pen pattern to black.

If the control is inactive or active and the contrlHilite field of the control record does not contain partCode (Line 880), the following occurs. If the control is inactive, the pen pattern and the text drawing pattern are both set to gray (Lines 882-886), otherwise they are set to black and the default respectively (Lines 887-891). A framed circle is then drawn (Line 893), and the interior of this circle is drawn in white (Lines 895-897). Then, if the control's value is 1, a smaller filled circle is drawn inside the first in either the gray pattern or black depending on whether the control is currently active or inactive (Lines 899-907).

If the the contrlHilite field of the control record contains partCode (Line 909), another circle is drawn 1 pixel inside the main circle.

Lines 916-917 then draw the control's title.

Note that this function draws a radio button with an appearance identical to that created by the standard radio button CDEF except that the control itself, as well as the title, is drawn dimmed when the control is inactive. Also, the interior of the button will always appear in white and not in the background colour of the window.

doTestMessage

doTestMessage handles testCntl messages to completion.

Lines 929-930 extract the mouse-down (local) coordinates from the param parameter. Lines 932-935 test whether the mouse-down was within the control's rectangle, returning partCode if it was or 0 otherwise.

CDEF2.c

CDEF2.c is the source code for the custom slider control definition function (CDEF). The CDEF uses custom dragging; accordingly, it responds only to initCntl, drawCntl, testCntl, dragCntl, and dispCntl messages.

#define

kInactive represents the value used to make a control inactive. kIndicatorHeight is the nominal height of the slider control's indicator in pixels. Line 978 is the resource ID for a 'PICT' resource containing images of the slider's track in the active state, the slider's track in the inactive state, the indicator in the active state, the indicator in the inactive state, and the indicator in the pressed state.

#typedef

The VBLRec data type is used by the functions relating to a VBL task. (That task delays the drawing of a dragged indicator until the vertical blank period occurs.)

The SliderDataRec data type will contain data relevant to the control. A handle to a SliderDataRec record will be assigned to contrlData field of the control's control record.

Global Variables

gColorQuickDrawPresent will be set to true if Color QuickDraw is present.

If the trap SlotVInstall is available (it will not be available on non-modular Macintoshes such as the Classic), gMainSlotNumber will be assigned the slot number of the main video device and a slot-based VBL task will to be synchronised with that video device's retrace.

gDragMessageFlag will be set to true if a mousedown occurs within the indicator rectangle and while a cntlDrag message is being handled.

gVBLInstallFail is a flag which will indicate whether or not the installation of a VBL task is successful.

gVBLRec is used by the VBL task functions.

main

The main function receives the incoming message and switches accordingly, returning the appropriate value to the Control Manager.

Lines 1043 and 1079 have to do with accessing the CDEF's global variables. (See Setting up globals in 68K code resources in the Code Resource Projects/Writing Code Resources section of Chapter 2 (Creating Mac OS Projects) in the CodeWarrior manual Targeting Mac OS.)

Lines 1045 and 1077 save and restore the pen state.

Note that, in the case of a drawCntl message (Line 1054), no action is taken if the contrlVis field of the control's control record specifies that the control is currently invisible.

doInitMessage

doInitMessage performs the initial handling of initCntl messages.

Lines 1094-1096 check whether Color QuickDraw is present and set the global variable gColorQuickDrawPresent accordingly.

Line 1098 locks the control's handle. Line 1100 allocates a relocatable block for a slider control data structure, the handle to which is assigned to the control record's contrlData field. Line 1102 calls a CDEF-defined function which creates an offscreen graphics world in which the images of the slider track and indicator will be stored. That completed, Line 1104 unlocks the control's handle.

Line 1106 checks for the availability of the trap SlotVInstall. If that trap is available, Line 1109-1112 get the slot number of the main graphics device. (This process involves getting a handle to the startup gDevice record, extracting from that record the device driver's reference number, getting a handle to the DCtlEntry structure, and then getting the slot number.)

doDrawMessage

doDrawMessage performs the initial handling of drawCntl messages.

It is always possible that the user will change the pixel depth of the display device, using the Monitors control panel, while the program is running. If Color QuickDraw is present (Line 1120), a CDEF-defined function is called to check whether the pixel depth of the display device equates to that of the offscreen graphics world (see below). If the two are not the same, and as will be seen, the CDEF-defined function destroys the offscreen graphics world and then recreates it with the appropriate pixel depth.

If the slider control is currently inactive (Line 1123), a CDEF-defined function is called to draw the control in its inactive state (Line 1124), otherwise a CDEF-defined function is called to draw the control in the active state (Lines 1125-1126).

doTestMessage

doTestMessage handles testCntl messages to completion.

Line 1136 calls a CDEF-defined function which returns a rectangle whose size is the same as the indicator and whose vertical location is determined by the current value in the control record's contrlValue field

Lines 1138-1139 extract the mouse-down (local) coordinates from the param parameter. If the mouse-down was within the indicator rectangle (Line 1141), the global variable gDragMessageFlag is set to true, the control is drawn, the global variable gDragMessageFlag is set to false again, and kControlIndicatorPart is returned. As will be seen, the effect of this is to cause the control to be drawn with the indicator appearing in the pressed state, thus providing the appropriate feedback to the user.

If the mousedown was not within the indicator rectangle (Line 1148), 0 is returned.

doDragMessage

doDragMessage handles dragCntl messages to completion, calling other CDEF-defined functions to draw the control and to install/remove a VBL task. It returns a non-zero value to advise the Control Manager that custom dragging is being performed.

Line 1164 sets gDragMessageFlag to true to record that a dragCntl message is currently being processed. Line 1166 locks the control's handle.

Lines 1168-1173 calculate the height of the complete slider control, less half the indicator's height, less the unused sections at the top and bottom. This establishes, in the trackHeight variable, the range of pixels over which the indicator is permitted to move (Line 1174). Line 1176 determines the range of control values, that is, the difference between the control's maximum and minimum values. Line 1177 then calculates a value which will allow the control's new value to be readily derived from the new indicator position when the indicator is moved.

Lines 1179-1180 set the slop rectangle to equal the window's port rectangle. This represents the outer limit beyond which vertical mouse movement will not result in indicator movement and control value change.

Line 1182 calls a CDEF-defined function which installs a VBL task. Lines 1183-1186 set a global variable to record whether or not the installation was successful.

Line 1188 calls the CDEF-defined function which returns a rectangle whose size is the same as the indicator and whose vertical location is determined by the current value in the control record's contrlValue field. Line 1190 gets the coordinates of the mouse position at the time that the mouse button was pressed.

Line 1192 initiates the custom dragging loop, which will continue until the mouse button is released. Within the loop:

- Line 1194 gets the current mouse coordinates and Line 1195 establishes whether the mouse cursor has moved vertically since the last time Line 1194 executed.
- If the cursor has moved vertically and if the cursor is still within the slop rectangle (Line 1197):
    - Lines 1199-1200 adjust the top and bottom of the indicator rectangle to reflect the change in mouse cursor position, and Line 1202 gets the new indicator centre.
    - Line 1204 calculates the required new control value to reflect the new indicator centre location, and assigns it to the contrlValue field of the control record.
    - Lines 1206-1209 ensure that the control's value can never be set outside the control's maximum and minimum values. In effect, this also limits the top and bottom track locations to which the indicator can be dragged.
    - Line 1211 then calls a CDEF-defined function to redraw the slider control with the indicator in its new location. That done, the variable containing the starting mouse location is reset to the current mouse location (Line 1213), and the loop continues.

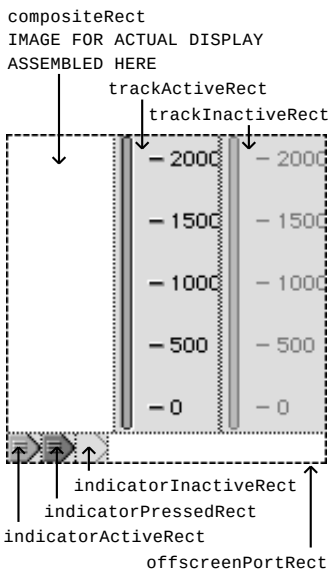
When the mouse button is released, the loop exits and Lines 1217-1218 remove the VBL task if the previous installation was successful. Line 1220 sets gDragMessageFlag to record that the custom dragging loop is no longer being executed. The final call to the drawing function at Line 1221 ensures that the indicator will return to the non-pressed appearance as soon as the mouse button is released. Line 1223 unlocks the control's handle

Line 1225 returns a non-zero value to advise the Control Manager that custom dragging is being employed.

doDisposeMessage

doDisposeMessage handles dispCntl messages. Lines 1234-1236 erase the slider control. Lines 1238-1239 dispose of the offscreen graphics world and Lines 1241-1242 release the memory occupied by the control's custom data.

createOffScreenGWorld



createOffScreenGWorld is called, when the initCntl message is received, to set up an offscreen graphics world in which the individual graphics components of the slider control are stored and in which the image for actual display will be assembled The diagram illustrates the result of this function.

Line 1255 gets a handle to the control's custom data. Lines 1257-1265 establish a number of rectangles, which are stored in the fields of the control's custom data record.

Line 1267 saves the current graphics world. Line 1269 creates the offscreen graphics world, Line 1272 locks the control's custom data record handle, and Lines 1274-1275 lock the offscreen buffer in memory preparatory to a drawing operation.

Line 1277 sets the offscreen graphics world as the current graphics world and Line 1279 cleans the slate.

Lines 1281-1288 have to do with determining which 'PICT' resource (colour or black and white) gets loaded at Line 1290. Firstly, if Color QuickDraw is present (Line 1281), the pixel depth of the current port is obtained (Lines 1283-1284). Then, if Color QuickDraw is not present, or if the pixel depth is 1, a variable is set to ensure that the 'PICT' resource with the black and white image is loaded (Lines 1287-1288).

Lines 1290-1295 read in the appropriate 'PICT' resource and draw the picture in the offscreen graphics port as shown in the diagram.

Line 1297 sets the graphics world saved at Line 1267 as the current graphics world. Line 1298 unlocks the offscreen buffer and Line 1299 unlocks the handle to the control's data record.

pixelDepthCheck

pixelDepthCheck checks whether there is any difference between the current pixel depth of the display device and the offscreen graphics world and, if so, destroys and recreates the offscreen graphics world so that the two pixel depths are made identical.

Line 1310 gets a handle to the control's custom data. Lines 1312-1315 get the two pixel depths. If they are not the same (Line 1317), Lines 1319-1320 destroy and recreate the offscreen graphics world. (Because the pixelDepth parameter in the NewGWorld call at Line 1269 is 0, the offscreen graphics world will be created with the the greatest pixel depth from among all screens whose boundary rectangles intersect the rectangle specified in the boundsRect parameter.)

drawControlActive

drawControlActive assembles the image for display in the offscreen graphics world and then copies that image from the offscreen graphics port to the window's graphics port. The latter action is delayed, in certain circumstances, until the vertical blanking period.

Lines 1334-1335 save the current foreground and background colours. Line 1337 locks the control record and Lines 1339-1340 lock the control's custom data. Lines 1342-1343 set the window which owns the control as the current graphics port.

Lines 1345-1346 locks the offscreen buffer and Lines 1348-1349 set the foreground colour to black and the background colour to white preparatory to upcoming calls to CopyBits (Recall that this measure prevents the possibility of unwanted colours being applied to the image.)

Lines 1351-1354 copy the active track image to the composite area of the offscreen graphics world. Lines 1356-1358 determine exactly where the indicator image needs to be drawn in the composite area. Then, if the mouse is down in the indicator rectangle or the indicator is being dragged, the pressed indicator image is copied to this rectangle, otherwise the active indicator image is copied (Lines 1360-1371). The image in the composite area of the offscreen graphics world is now ready for display.

If, in this instance, drawSliderControl has been called by doDragMessage and a VBL task has been installed (Line 1373), and if the VBL task has just executed (Line 1375), the composite image is copied from the offscreen graphics port to the window's graphics port. Note that the flag which is set by the VBL task every time it executes is reset to false as part of this sequence (Line 1377). The object of all this is to delay the drawing of the updated slider control, when it is being dragged, until the vertical blank period.

Going back to Line 1375, if that line indicates that the VBL task has not executed since the last time the execution flag was set to false, the function returns to doDragMessage without any CopyBits call being executed that time around. This fruitless cycle will continue until Line 1375 indicates that the VBL task has just executed.

Going back to Line 1373, if, for some reason, the VBL task was not installed successfully within doDragMessage, then drawSliderControl just goes straight ahead and copies the image to the window's graphics port regardless of the current stage of the screen refresh cycle. That is, Lines 1384-1389 execute in that circumstance.

Lines 1384-1389 are also executed, and Lines 1375-1382 are bypassed, when drawControlActive is called directly from doDrawMessage on receipt of a drawCntl message - for example, as a consequence of an UpdateControls call in the main program.

Line 1391 unlocks the offscreen buffer, Line 1392 unlocks the control's data record and Line 1393 unlocks the control record. Lines 1395-1396 restore the foreground and background colours saved at Lines 1334-1335.

drawControlInactive

drawControlInactive is called by doDrawMessage if the contrlHilite field of the control record indicates that the control is currently inactive. This function is similar to drawControlActive except that it does not need to accommodate the possibility that the slider indicator is being moved. Also, the track and indicator images assembled in the offscreen graphics world are the inactive versions.

calcIndicatorRect

calcIndicatorRect calculates the indicator's rectangle based on the control's current value.

Lines 1460-1461 gets the half-height of the indicator. Lines 1463-1466 calculate the effective track height, that is, the number of pixels over which the indicator is permitted to move.

Lines 1468-1470 extract the control's current value, and its maximum and minimum values, from the control record.

Lines 1472-1473 set the centre of the indicator's rectangle to reflect the control's current value and Lines 1475-1476 finish the job of setting the new coordinates of the indicator's rectangle. Line 1478 returns that rectangle to the calling function.

installVBLTask

installVBLTask is called by doDragMessage to install the VBL task.

Line 1487 sets to false the flag which is set to true when the VBL task executes. Lines 1489-1492 fill in the appropriate fields of the VBL task record. (Note that the vblCount field is set to 1 so that the VBL task will execute at the first interrupt.) Line 1494 saves the pointer to the A5 world.

If the trap SlotVInstall is available, the VBL task installed in the slot-based queue (Lines 1496-1497), otherwise it is inserted into the system-based queue (Lines 1498-1499). The success, or otherwise, of the attempted installation is returned to the calling function (Line 1501).

removeVBLTask

removeVBLTask simply removes the VBL task from the relevant queue.

theVBLTask

theVBLTask is the VBL task itself.

Line 1521 gets a pointer to the variable which contains the field holding the pointer to the application's A5 world. Line 1522 saves the current A5 and, at the same time, sets the application's A5 world as that to be used for the duration of the VBL task's execution. The task can now access the global variables.

Lines 1524 sets to true the flag which indicates that the VBL task has executed. When the task executes, the value in the vblCount field of the VBL task record will be 0. So that the task will execute again at the next interrupt, Line 1525 sets the vblCount field of the VBL task record back to 1. Line 1527 sets the current A5 world back to that saved at Line 1634.

checkSlotVInstallAvailable and checkTrapAvailable

checkSlotVInstallAvailable is called from Line 1106. checkTrapAvailable is called to check for the availability of the trap SlotVInstall. This code is repeated and explained at Chapter 21 — Miscellany.

Creating the CDEF Resource

To create CDEF resources from code such as that at Lines 648-1564, follow the same general procedure as is described for LDEFs in Demonstration Program Comments at Chapter 18 — Lists and Custom List Definition Functions.