19 Version 1.1

CUSTOM CONTROL DEFINITION FUNCTIONS AND VBL TASKS

Includes Demonstration Program CDEFandVBLPascal

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.

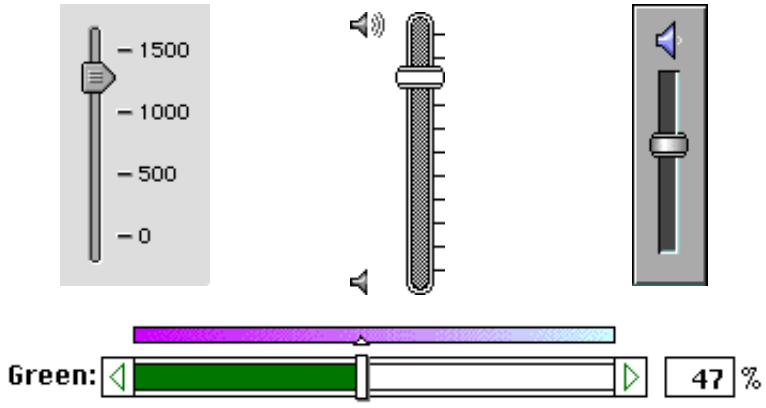


FIG 1 - TYPICAL SLIDER CONTROLS

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:

```
function ControlDef(varCode: SInt16; theControl: ControlRef; message:
                        ControlDefProcMessage; param: SInt32): SInt32;
```

`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 `UpdtControl` 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:
    - 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.

[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 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 in the high-order word and horizontal offset 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:

```
type
  IndicatorDragConstraint = record
    limitRect:      Rect;
    slopRect:       Rect;
    axis:           DragConstraint;
  END;

Also note the following:
  IndicatorDragConstraintPtr = ^IndicatorDragConstraint;
  IndicatorDragConstraintHandle = ^IndicatorDragConstraintPtr;
```

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 slider

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

control has arrow boxes and a repetitive action needs to be performed while the mouse button remains down in one of those boxes.

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.c 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. As a further example, an action procedure for the indicator of a volume slider control could change the volume in response to the user's actions with the slider.

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:

```
type
  VBLTask = record
    qLink:    QElemPtr;
    qType:    integer;
    vblAddr:  VBLUPP;
    vblCount: integer;
    vblPhase: integer;
  END;

VBLTaskPtr = ^VBLTask;
```

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`, you 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:

```
function GetVBLRec: Sint32;

          INLINE $2E88;
```

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`:

```
type
  VBLRec = record
    vblTaskRec : VBLTask;     { The VBL task record.}
    vblA5 : longint;          { Saved value of A5.}
    end;

  VBLRecPtr = ^VBLRec;
```

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

```
  VBLTask = record
    qLink:    QElemPtr;
    qType:    integer;
    vblAddr:  VBLUPP;
    vblCount: integer;
    vblPhase: integer;
  end;

VBLTaskPtr = ^VBLTask;
```

Routines

Slot-Based Installation and Removal Routines

```
function  SlotVInstall(vblBlockPtr: QElemPtr; theSlot: integer): OSErr;
function  SlotVRemove(vblBlockPtr: QElemPtr; theSlot: integer): OSErr;
```

System-Based Installation and Removal Routines

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

Utility Routines

```
function  AttachVBL(theSlot: integer): OSErr;
function  DoVBLTask(theSlot: integer): OSErr;
function  GetVBLQHdr: QHdrPtr;
```

Demonstration Program

```
{ ############################################################################################
// CDEFandVBLPascal.p
// ############################################################################################
//
// 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.
//
// 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 and doesActivateOnFGcase
//     flags set.
//
// ############################################################################################ }

program CDEFandVBLPascal(input, output);

{ ……… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, GestaltEqu, Retrace, LowMem,
  Palettes, SegLoad;

{ ……… define the following constants }

const

mApple = 128;
 iAbout = 1;
mFile = 129;
 iQuit = 11;
mDemonstration = 131;
 iVBLAnimCursor = 1;
rMenubar = 128;
rWindow = 128;
rFingersCursor = 128;
rStartRadioButton = 128;
rStopRadioButton = 129;
rSliderControl = 130;
kMaxLong = $7FFFFFFF;

{ ……… user-defined types }

type

AnimCurs = record
  numberOfFrames : integer;
  whichFrame : integer;
  frame : array [0..0] of CursHandle;
  end;
AnimCursPtr = ^AnimCurs;
AnimCursHandle = ^AnimCursPtr;

VBLRec = record
  vblTaskRec : VBLTask;
  thisApplicationsA5 : longint;
  end;
VBLRecPtr = ^VBLRec;

{ ……… global variables }

var

gColorQuickDrawPresent : boolean;
gColorDisplay : boolean;
gDone : boolean;
gSleepTime : longint;
gInBackground : boolean;
gWindowPtr : WindowPtr;
gAnimCursHdl : AnimCursHandle;
gVBLRec : VBLRec;
gVBLCount : integer;
gAnimatedCursorActive : boolean;
gWindowColour : RGBColor;
gSliderControlHdl : ControlHandle;
gStartControlHdl : ControlHandle;
gStopControlHdl : ControlHandle;

theErr : OSErr;
response : longint;
mainDeviceHdl : GDHandle;
```

```
108   bitsPerPixel : integer;
109   menubarHdl : Handle;
110   menuHdl : MenuHandle;
111   eventRec : EventRecord;
112
113   { …… in-line glue for GetVBLRec }
114
115   function GetVBLRec : longint;
116     {$IFC NOT GENERATINGCFM}
117     inline $2E88;
118     {$ENDC}
119
120   { ########################################################################## DoInitManagers }
121
122   procedure DoInitManagers;
123
124     begin
125     MaxApplZone;
126     MoreMasters;
127
128     InitGraf(@qd.thePort);
129     InitFonts;
130     InitWindows;
131     InitMenus;
132     TEInit;
133     InitDialogs(nil);
134
135     InitCursor;
136     FlushEvents(everyEvent, 0);
137     end;
138       {of procedure DoInitManagers}
139
140   { ######################################################################## DoStopAnimCursor }
141
142   procedure DoStopAnimCursor;
143
144     var
145     a : integer;
146     theRect : Rect;
147     ignored : OSErr;
148
149     begin
150     ignored := VRemove(QElemPtr(@gVBLRec.vblTaskRec));
151
152     for a := 0 to (gAnimCursHdl^^.numberOfFrames - 1) do
153       ReleaseResource(Handle(gAnimCursHdl^^.frame[a]));
154
155     ReleaseResource(Handle(gAnimCursHdl));
156
157     gAnimatedCursorActive := false;
158     gSleepTime := kMaxLong;
159
160     SetCursor(qd.arrow);
161
162     SetRect(theRect, 30, 100, 150, 130);
163     RGBBackColor(gWindowColour);
164     FillRect(theRect, qd.white);
165     end;
166       {of procedure DoStopAnimCursor}
167
168   { ######################################################################## AnimCursVBLTask }
169
170   procedure AnimCursVBLTask;
171
172     var
173     theVBLRecPtr : VBLRecPtr;
174     currentA5 : longint;
175
176     begin
177     theVBLRecPtr := VBLRecPtr(GetVBLRec);
178     currentA5 := SetA5(theVBLRecPtr^.thisApplicationsA5);
179
180     SetCursor(gAnimCursHdl^^.frame[gAnimCursHdl^^.whichFrame]^^);
181     gAnimCursHdl^^.whichFrame := gAnimCursHdl^^.whichFrame + 1;
182
183     if (gAnimCursHdl^^.whichFrame = gAnimCursHdl^^.numberOfFrames) then
```

```
    gAnimCursHdl^^.whichFrame := 0;

  theVBLRecPtr^.vblTaskRec.vblCount := gVBLCount;

  currentA5 := SetA5(currentA5);
  end;
    {of procedure AnimCursVBLTask}

{ ##################################################################### DoInstallSystemVBLTask }

procedure DoInstallSystemVBLTask;

  var
  ignored : OSErr;

  begin
  gVBLRec.vblTaskRec.qType := vType;
  gVBLRec.vblTaskRec.vblAddr := VBLUPP(@AnimCursVBLTask);
  gVBLRec.vblTaskRec.vblCount := gVBLCount;
  gVBLRec.vblTaskRec.vblPhase := 0;

  gVBLRec.thisApplicationsA5 := SetCurrentA5;

  ignored := VInstall(QElemPtr(@gVBLRec.vblTaskRec));
  end;
    {of procedure DoInstallSystemVBLTask}

{ ############################################################################ DoGetAnimCursor }

function DoGetAnimCursor(resourceID : integer) : boolean;

  var
  cursorID, a : integer;
  noError : boolean;

  begin
  a := 0;
  noError := false;

  gAnimCursHdl := AnimCursHandle(GetResource('acur', resourceID));
  if (gAnimCursHdl <> nil) then
    begin
    noError := true;
    while ((a < gAnimCursHdl^^.numberOfFrames) and noError) do
      begin
      cursorID := integer(HiWord(longint(gAnimCursHdl^^.frame[a])));

      gAnimCursHdl^^.frame[a] := GetCursor(cursorID);
      if (gAnimCursHdl^^.frame[a] <> nil) then
        a := a + 1
      else
        noError := false;
      end;
    end;

  if (noError) then
    gAnimCursHdl^^.whichFrame := 0;

  DoGetAnimCursor := noError;
  end;
    {of procedure DoGetAnimCursor}

{ ###################################################################### DoStartAnimCursor }

procedure DoStartAnimCursor;

  begin
  gVBLCount := 30;
  gSleepTime := 0;

  if (DoGetAnimCursor(rFingersCursor) = false) then
    ExitToShell;

  DoInstallSystemVBLTask;

  gAnimatedCursorActive := true;
```

```
260
261     MoveTo(40, 110);
262     DrawString('Press any key to');
263     MoveTo(30, 125);
264     DrawString('stop animated cursor');
265     end;
266       {of procedure DoInitManagers}
267
268   { ###################################################################### DoDrawControlsPanel }
269
270   procedure DoDrawControlsPanel;
271
272     var
273     mainDeviceHdl : GDHandle;
274     bitsPerPixel : integer;
275     fontNum, a : integer;
276     gray8 : RGBColor;
277
278     begin
279     gray8.red := $7777;
280     gray8.blue := $7777;
281     gray8.green := $7777;
282
283     GetFNum('Chicago', fontNum);
284     TextFont(fontNum);
285     TextSize(12);
286
287     mainDeviceHdl := LMGetMainDevice;
288     bitsPerPixel := mainDeviceHdl^^.gdPMap^^.pixelSize;
289     if (bitsPerPixel > 1) then
290       gColorDisplay := true
291     else
292       gColorDisplay := false;
293
294     for a := 0 to 1 do
295       begin
296       if (a = 0) then
297         ForeColor(whiteColor)
298       else begin
299         if not (gInBackground) then
300           begin
301           if (gColorQuickDrawPresent and gColorDisplay) then
302             ForeColor(blackColor)
303           else begin
304             ForeColor(blackColor);
305             PenPat(qd.black);
306             TextMode(srcOr);
307             end;
308           end
309         else begin
310           if (gColorQuickDrawPresent and gColorDisplay) then
311             RGBForeColor(gray8)
312           else begin
313             ForeColor(blackColor);
314             PenPat(qd.gray);
315             TextMode(grayishTextOr);
316             end;
317           end;
318         end;
319
320       if not ((a = 0) and not gColorDisplay) then
321         begin
322         MoveTo(156-a, 22-a);
323         LineTo(152-a, 22-a);
324         LineTo(152-a, 230-a);
325         LineTo(246-a, 230-a);
326         LineTo(246-a, 22-a);
327         LineTo(242-a, 22-a);
328
329         MoveTo(163-a, 26-a);
330         DrawString('Engine RPM');
331         end;
332       end;
333
334     ForeColor(blackColor);
335
```

```
  GetFNum('Geneva', fontNum);
  TextFont(fontNum);
  TextSize(10);

  PenPat(qd.black);
  TextMode(srcOr);
  end;
    {of procedure DoDrawControlsPanel}

{ ############################################################## DoGetSliderControlSuite }

procedure DoGetSliderControlSuite;

  begin
  gSliderControlHdl := GetNewControl(rSliderControl, gWindowPtr);
  HiliteControl(gSliderControlHdl, 255);

  gStartControlHdl := GetNewControl(rStartRadioButton, gWindowPtr);
  gStopControlHdl := GetNewControl(rStopRadioButton, gWindowPtr);

  DoDrawControlsPanel;
  end;
    {of procedure DoGetSliderControlSuite}

{ ############################################################################ DoMenuChoice }

procedure DoMenuChoice(menuChoice : longint);

  var
  menuID, menuItem : integer;
  itemName : string;
  daDriverRefNum : integer;

  begin
  menuID := HiWord(menuChoice);
  menuItem := LoWord(menuChoice);

  if (menuID = 0) then
    Exit(DoMenuChoice);

  case (menuID) of

    mApple: begin
      if (menuItem = iAbout) then
        SysBeep(10)
      else begin
        GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
        daDriverRefNum := OpenDeskAcc(itemName);
        end;
      end;

    mFile: begin
      if (menuItem = iQuit) then
        begin
        ExitToShell;
        DisposeWindow(gWindowPtr);
        gDone := true;
        end;
      end;

    mDemonstration: begin
      if (menuItem = iVBLAnimCursor) then
        DoStartAnimCursor;
      end;
    end;
      {of case statement}

  HiliteMenu(0);
  end;
    {of procedure DoMenuChoice}

{ ############################################################################ DoInContent }

procedure DoInContent(var theEvent : EventRecord; theWindowPtr : WindowPtr);

  var
```

```
412     controlHdl : ControlHandle;
413     partCode : integer;
414     theRect : Rect;
415     theString : string;
416     ignored : ControlPartCode;
417
418     begin
419     GlobalToLocal(theEvent.where);
420
421     partCode := FindControl(theEvent.where, theWindowPtr, controlHdl);
422
423     if (controlHdl = gSliderControlHdl) then
424       begin
425       if (partCode = kControlIndicatorPart) then
426         ignored := TrackControl(controlHdl, theEvent.where, nil);
427
428       RGBBackColor(gWindowColour);
429       SetRect(theRect, 253, 107, 390, 119);
430       FillRect(theRect, qd.white);
431       MoveTo(255, 117);
432       DrawString('Slider Control Value: ');
433       NumToString(longint(GetControlValue(controlHdl)), theString);
434       DrawString(theString);
435       end
436     else if ((controlHdl = gStartControlHdl) or (controlHdl = gStopControlHdl)) then
437       begin
438       if (TrackControl(controlHdl, theEvent.where, nil) <> 0) then
439         begin
440         if (controlHdl = gStartControlHdl) then
441           begin
442           HiliteControl(gSliderControlHdl, 0);
443           SetControlValue(gStartControlHdl, 1);
444           SetControlValue(gStopControlHdl, 0);
445           end
446         else if (controlHdl = gStopControlHdl) then
447           begin
448           SetControlValue(gSliderControlHdl, 0);
449           HiliteControl(gSliderControlHdl, 255);
450           SetControlValue(gStartControlHdl, 0);
451           SetControlValue(gStopControlHdl, 1);
452
453           RGBBackColor(gWindowColour);
454           SetRect(theRect, 253, 107, 390, 119);
455           FillRect(theRect, qd.white);
456           end;
457         end;
458       end;
459     end;
460       {of procedure DoInContent}
461
462   { ############################################################################ DoActivateWindow }
463
464   procedure DoActivateWindow(becomingActive : boolean);
465
466     var
467     controlVal : integer;
468
469     begin
470     if (becomingActive) then
471       begin
472       controlVal := GetControlValue(gStartControlHdl);
473       if (controlVal = 1) then
474         HiliteControl(gSliderControlHdl, 0);
475       HiliteControl(gStartControlHdl, 0);
476       HiliteControl(gStopControlHdl, 0);
477       end
478     else begin
479       HiliteControl(gSliderControlHdl, 255);
480       HiliteControl(gStartControlHdl, 255);
481       HiliteControl(gStopControlHdl, 255);
482       end;
483
484     DoDrawControlsPanel;
485     end;
486       {of procedure DoActivateWindow}
487
```

```
{ ############################################################################## DoOSEvent }

procedure DoOSEvent(var theEvent : EventRecord);

  begin
  case BAnd(BSR(theEvent.message, 24), $000000FF) of

    suspendResumeMessage: begin
      gInBackground := BAnd(theEvent.message, resumeFlag) = 0;
      DoActivateWindow(not gInBackground);
      end;

    mouseMovedMessage: begin
      end;
    end;
      {of case statement}
  end;
    {of procedure DoOSEvent}

{ ############################################################################## DoUpdate }

procedure DoUpdate(var theEvent : EventRecord);

  var
  theWindowPtr : WindowPtr;

  begin
  theWindowPtr := WindowPtr(theEvent.message);

  BeginUpdate(theWindowPtr);

  SetPort(theWindowPtr);
  UpdateControls(theWindowPtr, theWindowPtr^.visRgn);
  DoDrawControlsPanel;

  EndUpdate(theWindowPtr);
  end;
    {of procedure DoUpdate}

{ ############################################################################## DoActivate }

procedure DoActivate(var theEvent : EventRecord);

  var
  theWindowPtr : WindowPtr;
  becomingActive : boolean;

  begin
  theWindowPtr := WindowPtr(theEvent.message);

  becomingActive := (BAnd(theEvent.modifiers, activeFlag) = activeFlag);
  DoActivateWindow(becomingActive);
  end;
    {of procedure DoActivate}

{ ############################################################################## DoMouseDown }

procedure DoMouseDown(var theEvent : EventRecord);

  var
  partCode : integer;
  theWindowPtr : WindowPtr;
  menuHdl : MenuHandle;

  begin
  partCode := FindWindow(theEvent.where, theWindowPtr);
  menuHdl := GetMenuHandle(mDemonstration);

  case (partCode) of

    inMenuBar: begin
      if (gAnimatedCursorActive) then
        DisableItem(menuHdl, iVBLAnimCursor)
      else
        EnableItem(menuHdl, iVBLAnimCursor);
      DoMenuChoice(MenuSelect(theEvent.where));
```

```
564         end;
565
566       inSysWindow: begin
567         SystemClick(theEvent, theWindowPtr);
568         end;
569
570       inContent: begin
571         if (theWindowPtr <> FrontWindow) then
572           SelectWindow(theWindowPtr)
573         else
574           DoInContent(theEvent, theWindowPtr);
575         end;
576
577       inDrag: begin
578         DragWindow(theWindowPtr, theEvent.where, qd.screenBits.bounds);
579         end;
580       end;
581         {of case statement}
582     end;
583       {of procedure DoMouseDown}
584
585   { ############################################################################## DoEvents }
586
587   procedure DoEvents(var theEvent : EventRecord);
588
589     begin
590     case (theEvent.what) of
591
592       mouseDown: begin
593         DoMouseDown(theEvent);
594         end;
595
596       keyDown, autoKey: begin
597         if (gAnimatedCursorActive) then
598           DoStopAnimCursor;
599         end;
600
601       updateEvt: begin
602         DoUpdate(theEvent);
603         end;
604
605       activateEvt: begin
606         DoActivate(theEvent);
607         end;
608
609       osEvt: begin
610         DoOSEvent(theEvent);
611         HiliteMenu(0);
612         end;
613       end;
614         {of case statement}
615     end;
616       {of procedure DoEvents}
617
618   { ########################################################### start of main program }
619
620   begin
621
622   gColorQuickDrawPresent := false;
623   gColorDisplay := false;
624   gAnimatedCursorActive := false;
625   gWindowColour.red := $DDDD;
626   gWindowColour.green := $DDDD;
627   gWindowColour.blue := $DDDD;
628
629     { …… initialise managers }
630
631     DoInitManagers;
632
633     { …… check for Color QuickDraw }
634
635     theErr := Gestalt(gestaltQuickdrawVersion, response);
636     if (response >= gestalt8BitQD) then
637       begin
638       gColorQuickDrawPresent := true;
639
```

```
      mainDeviceHdl := LMGetMainDevice;
      bitsPerPixel := mainDeviceHdl^^.gdPMap^^.pixelSize;
      if (bitsPerPixel > 1) then
        gColorDisplay := true;
      end;

    { ……… set up menu bar and menus }

    menubarHdl := GetNewMBar(rMenubar);
    if (menubarHdl = nil) then
      ExitToShell;
    SetMenuBar(menubarHdl);
    DrawMenuBar;

    menuHdl := GetMenuHandle(mApple);
    if (menuHdl = nil) then
      ExitToShell
    else
      AppendResMenu(menuHdl, 'DRVR');

    { ……… open window }

    if (gColorQuickDrawPresent) then
      begin
      gWindowPtr := GetNewCWindow(rWindow, nil, WindowPtr(-1));
      if (gWindowPtr = nil) then
        ExitToShell;
      end
    else begin
      gWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
      if (gWindowPtr = nil) then
        ExitToShell;
      end;

    SetPort(gWindowPtr);

    { ……… get slider control suite }

    DoGetSliderControlSuite;

    { ……… enter eventLoop }

    gDone := false;
    gSleepTime := kMaxLong;

    while not (gDone) do
      begin
      if (WaitNextEvent(everyEvent, eventRec, gSleepTime, nil)) then
        DoEvents(eventRec);
      end;

  end.
    {of main program block}

  { ############################################################################### }

  { ###############################################################################
  // CDEF1Pascal.p              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 Grayscale Appearance for
  // System 7.5 published by Apple Computer Inc.
  //
  // ############################################################################### }
```

```
716
717 unit CDEF1Pascal;
718
719
720
721 { ……… unit interface section }
722
723 interface
724
725 { ……… include the following Universal Interfaces }
726
727 uses
728
729   Controls, Fonts, Menus, Quickdraw, QuickdrawText, Processes, Icons, Types,
730   Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, QDOffscreen, SegLoad, Retrace,
731   Traps, PascalA4, GestaltEqu, LowMem;
732
733 { ……… define the following constants }
734
735 const
736
737 rActiveDeselect = 128;
738 rActiveSelect = 129;
739 rInactiveDeselect = 130;
740 rInactiveSelect = 131;
741 rFeedbackDeselect = 132;
742 rFeedbackSelect = 133;
743 partCode = 1;
744
745 { ……… procedure and function interfaces }
746
747   {$MAIN}
748   function main(varCode : integer; controlHdl : ControlHandle; message : integer;
749       param : longint) : longint;
750
751
752
753 { ……… unit implementation section }
754
755 implementation
756
757 { ……… global variables }
758
759 var
760
761 gTheQDGlobalsPtr : QDGlobalsPtr;
762 gColorQuickDrawPresent : boolean;
763 gColorDisplay : boolean;
764 gActiveDeselectHdl : CIconHandle;
765 gActiveSelectHdl : CIconHandle;
766 gInactiveDeselectHdl : CIconHandle;
767 gInactiveSelectHdl : CIconHandle;
768 gFeedbackDeselectHdl : CIconHandle;
769 gFeedbackSelectHdl : CIconHandle;
770
771 { ……… procedure and function interfaces }
772
773   procedure DoInitMessage; forward;
774   procedure DoDrawMessage(controlHdl : ControlHandle); forward;
775   procedure DrawColour(controlHdl : ControlHandle; controlValue : integer;
776         controlRect : Rect); forward;
777   procedure DrawMono(controlHdl : ControlHandle; controlValue : integer;
778         controlRect : Rect); forward;
779   function  DoTestMessage(controlHdl : ControlHandle;
780         param : longint) : longint; forward;
781
782 { ########################################################################## main }
783
784 function main(varCode : integer; controlHdl : ControlHandle; message : integer;
785                               param : longint) : longint;
786
787   var
788   returnValue : longint;
789   oldA4, ignored : longint;
790
791   begin
```

```
792     oldA4 := SetCurrentA4;
793
794     case (message) of
795
796       initCntl: begin
797         DoInitMessage;
798         end;
799
800       drawCntl: begin
801         if (controlHdl^^.contrlVis <> 0) then
802           DoDrawMessage(controlHdl);
803         returnValue := 0;
804         end;
805
806       testCntl: begin
807         returnValue := DoTestMessage(controlHdl, param);
808         end;
809
810       otherwise begin
811         returnValue := 0;
812         end;
813       end;
814         {of case statement}
815
816     main := returnValue;
817     ignored := SetA4(oldA4);
818     end;
819       {of main procedure}
820
821   { ############################################################################ DoInitMessage }
822
823   procedure DoInitMessage;
824
825     var
826     theErr : OSErr;
827     response : longint;
828
829     begin
830
831     gTheQDGlobalsPtr :=
832       QDGlobalsPtr(longint(Ptr(SetCurrentA5)) - (sizeof(QDGlobals) - sizeof(GrafPtr)));
833
834     theErr := Gestalt(gestaltQuickdrawVersion, response);
835
836     if (response >= gestalt8BitQD) then
837       gColorQuickDrawPresent := true
838     else
839       gColorQuickDrawPresent := false;
840
841     gColorDisplay := false;
842     gActiveDeselectHdl := nil;
843     gActiveSelectHdl := nil;
844     gInactiveDeselectHdl := nil;
845     gInactiveSelectHdl := nil;
846     gFeedbackDeselectHdl := nil;
847     gFeedbackSelectHdl := nil;
848     end;
849       {of procedure DoInitMessage}
850
851   { ############################################################################ DoDrawMessage }
852
853   procedure DoDrawMessage(controlHdl : ControlHandle);
854
855     var
856     theWindowPtr : WindowPtr;
857     oldPort : GrafPtr;
858     oldFont, oldSize, oldTextMode, controlValue, fontNum : integer;
859     oldPenState : PenState;
860     controlRect : Rect;
861     mainDeviceHdl : GDHandle;
862     bitsPerPixel : integer;
863
864     begin
865     theWindowPtr := WindowPtr(controlHdl^^.contrlOwner);
866
867     GetPort(oldPort);
```

```
868     oldFont := theWindowPtr^.txFont;
869     oldSize := theWindowPtr^.txSize;
870     oldTextMode := theWindowPtr^.txMode;
871     GetPenState(oldPenState);
872
873     SetPort(theWindowPtr);
874
875     controlValue := GetControlValue(controlHdl);
876
877     controlRect := controlHdl^^.contrlRect;
878     controlRect.right := controlRect.left + 12;
879     controlRect.bottom := controlRect.top + 12;
880
881     GetFNum('Chicago', fontNum);
882     TextFont(fontNum);
883     TextSize(12);
884
885     if (gColorQuickDrawPresent) then
886       begin
887       if (gActiveDeselectHdl = nil) then
888         gActiveDeselectHdl := GetCIcon(rActiveDeselect);
889       if (gActiveSelectHdl = nil) then
890         gActiveSelectHdl := GetCIcon(rActiveSelect);
891       if (gInactiveDeselectHdl = nil) then
892         gInactiveDeselectHdl := GetCIcon(rInactiveDeselect);
893       if (gInactiveSelectHdl = nil) then
894         gInactiveSelectHdl := GetCIcon(rInactiveSelect);
895       if (gFeedbackDeselectHdl = nil) then
896         gFeedbackDeselectHdl := GetCIcon(rFeedbackDeselect);
897       if (gFeedbackSelectHdl = nil) then
898         gFeedbackSelectHdl := GetCIcon(rFeedbackSelect);
899       end;
900
901     mainDeviceHdl := LMGetMainDevice;
902     bitsPerPixel := mainDeviceHdl^^.gdPMap^^.pixelSize;
903     if (bitsPerPixel > 1) then
904       gColorDisplay := true
905     else
906       gColorDisplay := false;
907
908     if (gColorQuickDrawPresent and gColorDisplay) then
909       DrawColour(controlHdl, controlValue, controlRect)
910     else
911       DrawMono(controlHdl, controlValue, controlRect);
912
913     SetPenState(oldPenState);
914     TextMode(oldTextMode);
915     TextFont(oldFont);
916     TextSize(oldSize);
917     SetPort(oldPort);
918     end;
919       {of procedure DoDrawMessage}
920
921   { ############################################################################ DrawColour }
922
923   procedure DrawColour(controlHdl : ControlHandle; controlValue : integer;
924                            controlRect : Rect);
925
926     var
927     oldForeColour : RGBColor;
928     blackColour : RGBColor;
929     gray8 : RGBColor;
930
931     begin
932     blackColour.red := $0000;
933     blackColour.green := $0000;
934     blackColour.blue := $0000;
935     gray8.red := $7777;
936     gray8.green := $7777;
937     gray8.blue := $7777;
938
939     GetForeColor(oldForeColour);
940
941     if (controlHdl^^.contrlHilite = 255) then
942       begin
943       if (gColorDisplay) then
```

```
944         RGBForeColor(gray8)
945       else
946         TextMode(grayishTextOr);
947
948       if (controlValue = 1) then
949         PlotCIcon(controlRect, gInactiveSelectHdl)
950       else
951         PlotCIcon(controlRect, gInactiveDeselectHdl);
952       end
953     else if (controlHdl^^.contrlHilite = 0) then
954       begin
955       RGBForeColor(blackColour);
956       TextMode(srcOr);
957
958       if (controlValue = 1) then
959         PlotCIcon(controlRect, gActiveSelectHdl)
960       else
961         PlotCIcon(controlRect, gActiveDeselectHdl);
962       end
963     else if (controlHdl^^.contrlHilite = partCode) then
964       begin
965       if (controlValue = 1) then
966         PlotCIcon(controlRect, gFeedbackSelectHdl)
967       else
968         PlotCIcon(controlRect, gFeedbackDeselectHdl);
969       end;
970
971     MoveTo(controlRect.left + 17, controlRect.top + 10);
972     DrawString(controlHdl^^.contrlTitle);
973
974     RGBForeColor(oldForeColour);
975     end;
976       {of procedure DrawColour}
977
978   { ############################################################################ DrawMono }
979
980   procedure DrawMono(controlHdl : ControlHandle; controlValue : integer; controlRect : Rect);
981
982     begin
983     ForeColor(blackColor);
984     BackColor(whiteColor);
985
986     PenNormal;
987
988     if ((controlHdl^^.contrlHilite = 255) or (controlHdl^^.contrlHilite = 0)) then
989       begin
990       if (controlHdl^^.contrlHilite = 255) then
991         begin
992         PenPat(gTheQDGlobalsPtr^.gray);
993         TextMode(grayishTextOr);
994         end
995       else begin
996         PenPat(gTheQDGlobalsPtr^.black);
997         TextMode(srcOr);
998         end;
999
1000      FrameOval(controlRect);
1001
1002      InsetRect(controlRect, 1, 1);
1003      FillOval(controlRect, gTheQDGlobalsPtr^.white);
1004      InsetRect(controlRect, -1, -1);
1005
1006      if (controlValue = 1) then
1007        begin
1008        InsetRect(controlRect, 3, 3);
1009        if (controlHdl^^.contrlHilite = 255) then
1010          FillOval(controlRect, gTheQDGlobalsPtr^.gray)
1011        else
1012          FillOval(controlRect, gTheQDGlobalsPtr^.black);
1013        InsetRect(controlRect, -3, -3);
1014        end;
1015      end
1016    else if (controlHdl^^.contrlHilite = partCode) then
1017      begin
1018      InsetRect(controlRect, 1, 1);
1019      FrameOval(controlRect);
```

```
      InsetRect(controlRect, -1, -1);
      end;

    MoveTo(controlRect.left + 17, controlRect.top + 10);
    DrawString(controlHdl^^.contrlTitle);
    end;
      {of procedure DrawMono}

{ ############################################################################ DoTestMessage }

function DoTestMessage(controlHdl : ControlHandle; param : longint) : longint;

    var
    controlRect : Rect;
    mouseXY : Point;

    begin
    controlRect := controlHdl^^.contrlRect;

    mouseXY.v := HiWord(param);
    mouseXY.h := LoWord(param);

    if (PtInRect(mouseXY, controlRect)) then
      DoTestMessage := partCode
    else
      DoTestMessage := 0;

    end;
      {of function DoTestMessage}

end.
    {of unit CDEF1Pascal}

{ ############################################################################################ }

{ ############################################################################################
// CDEF2Pascal.p                    Custom control definition function for slider control
// ############################################################################################
//
// This CDEF displays:
//
// •   A 3D coloured slider control on colour displays set to pixel depths greater than 1.
//
// •   A black-and-white slider control on colour displays set to pixel depths less than
//     2.
//
// •   A black-and-white slider control if Color QuickDraw is not present.
//
// The CDEF utilises two 'PICT' resources (purgeable).  One resource contains the colour
// version of the slider control components.  The other comprises the black and white
// version.  The appearance of the coloured slider conforms to the specification for
// slider controls contained in the document Apple Greyscale Appearance for System 7.5
// published by Apple Computer Inc.
//
// ############################################################################################ }

unit CDEF2Pascal;



{ …… unit interface section }

interface

{ …… include the following Universal Interfaces }

uses

    Controls, Fonts, Menus, Quickdraw, Processes, Types,
    Memory, Events, ToolUtils, OSUtils, Devices, QDOffscreen, SegLoad, Retrace,
    Traps, PascalA4, GestaltEqu, LowMem;

{ …… procedure and function interfaces }

    {$MAIN}
    function main(varCode : integer; theControl : ControlHandle; message : integer;
```

```
1096            param : longint) : longint;
1097
1098
1099
1100    { ……… unit implementation section }
1101
1102    implementation
1103
1104    { ……… define the following constants }
1105
1106    const
1107
1108    kInactive = 255;
1109    kIndicatorHeight = 16;
1110    rTrackPict = 128;
1111
1112    { ……… user-defined types }
1113
1114    type
1115
1116    VBLRec = record
1117      vblTaskRec : VBLTask;
1118      inVBlankPeriod : boolean;
1119      thisApplicationsA5 : longint;
1120      end;
1121    VBLRecPtr = ^VBLRec;
1122
1123    SliderDataRec = record
1124      offScreenPort : GWorldPtr;
1125      offScreenPortRect : Rect;
1126      trackActiveRect : Rect;
1127      trackInactiveRect : Rect;
1128      indicatorActiveRect : Rect;
1129      indicatorPressedRect : Rect;
1130      indicatorInactiveRect : Rect;
1131      compositeRect : Rect;
1132      currentPort : GWorldPtr;
1133      currentDevice : GDHandle;
1134      ColorQuickDrawPresent : boolean;
1135      mainSlotNumber : integer;
1136      slotVInstallPresent : boolean;
1137      dragMessageFlag : boolean;
1138      VBLInstallFail : boolean;
1139      VBLRec : VBLRec;
1140      end;
1141    SliderDataPtr = ^SliderDataRec;
1142    SliderDataHdl = ^SliderDataPtr;
1143
1144    { ……… in-line glue for GetVBLRec }
1145
1146    function GetVBLRec : VBLRecPtr;
1147
1148      {$IFC NOT GENERATINGCFM }
1149      inline $2E88;
1150      {$ENDC}
1151
1152    { ……… procedure and function interfaces }
1153
1154      procedure DoInitMessage(theControl : ControlHandle); forward;
1155      procedure DoDrawMessage(theControl : ControlHandle); forward;
1156      function  DoTestMessage(theControl : ControlHandle;
1157            param : longint) : longint; forward;
1158      function  DoDragMessage(theControl : ControlHandle) : longint; forward;
1159      procedure DoDisposeMessage(theControl : ControlHandle); forward;
1160      procedure CreateOffScreenGWorld(theControl : ControlHandle); forward;
1161      procedure PixelDepthCheck(theControl : ControlHandle); forward;
1162      procedure DrawControlActive(theControl : ControlHandle); forward;
1163      procedure DrawControlInactive(theControl : ControlHandle); forward;
1164      function  CalcIndicatorRect(theControl : ControlHandle) : Rect; forward;
1165      function  InstallVBLTask(theControl : ControlHandle) : OSErr; forward;
1166      procedure RemoveVBLTask(theControl : ControlHandle); forward;
1167      procedure TheVBLTask; forward;
1168      function  CheckSlotVInstallAvailable : boolean; forward;
1169      function  CheckTrapAvailable(theTrap : integer) : boolean; forward;
1170
1171    { ############################################################################## main }
```

```
1172
1173  function main(varCode : integer; theControl : ControlHandle; message : integer;
1174                                param : longint) : longint;
1175
1176    var
1177    oldPenState : PenState;
1178    returnValue : longint;
1179    oldA4, ignored : longint;
1180
1181    begin
1182    oldA4 := SetCurrentA4;
1183
1184    GetPenState(oldPenState);
1185
1186    case (message) of
1187
1188      initCntl: begin
1189        DoInitMessage(theControl);
1190        returnValue := 0;
1191        end;
1192
1193      drawCntl: begin
1194        if (theControl^^.contrlVis <> 0) then
1195          DoDrawMessage(theControl);
1196        returnValue := 0;
1197        end;
1198
1199      testCntl: begin
1200        returnValue := DoTestMessage(theControl, param);
1201        end;
1202
1203      dragCntl: begin
1204        returnValue := DoDragMessage(theControl);
1205        end;
1206
1207      dispCntl: begin
1208        DoDisposeMessage(theControl);
1209        returnValue := 0;
1210        end;
1211
1212      otherwise begin
1213        returnValue := 0;
1214        end;
1215      end;
1216        {of case statement}
1217
1218    SetPenState(oldPenState);
1219
1220    main := returnValue;
1221    ignored := SetA4(oldA4);
1222    end;
1223      {of main procedure}
1224
1225  { ############################################################################ DoInitMessage }
1226
1227  procedure DoInitMessage(theControl : ControlHandle);
1228
1229    var
1230    theErr : OSErr;
1231    response : longint;
1232    mainDeviceHdl : GDHandle;
1233    mainDeviceRefNum : integer;
1234    deviceCtlEntryHdl : DCtlHandle;
1235    theSliderDataHdl : SliderDataHdl;
1236
1237    begin
1238    theControl^^.contrlData := NewHandleClear(sizeof(SliderDataRec));
1239
1240    if (theControl^^.contrlData <> nil) then
1241      begin
1242      theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1243      theSliderDataHdl^^.ColorQuickDrawPresent := true;
1244      theSliderDataHdl^^.dragMessageFlag := false;
1245      theSliderDataHdl^^.VBLInstallFail := true;
1246
1247      theErr := Gestalt(gestaltQuickdrawVersion, response);
```

```
1248      if (response < gestalt8BitQD) then
1249        theSliderDataHdl^^.ColorQuickDrawPresent := false;
1250
1251      HLock(Handle(theControl));
1252
1253      CreateOffScreenGWorld(theControl);
1254
1255      HUnlock(Handle(theControl));
1256
1257      theSliderDataHdl^^.slotVInstallPresent := CheckSlotVInstallAvailable;
1258      if (theSliderDataHdl^^.slotVInstallPresent) then
1259        begin
1260        mainDeviceHdl := LMGetMainDevice;
1261        mainDeviceRefNum := mainDeviceHdl^^.gdRefNum;
1262        deviceCtlEntryHdl := GetDCtlEntry(mainDeviceRefNum);
1263        theSliderDataHdl^^.mainSlotNumber :=
1264              integer(AuxDCEHandle(deviceCtlEntryHdl)^^.dCtlSlot);
1265        end;
1266      end;
1267    end;
1268      {of procedure DoInitMessage}
1269
1270  { ############################################################################ DoDrawMessage }
1271
1272  procedure DoDrawMessage(theControl : ControlHandle);
1273
1274    begin
1275    if (SliderDataHdl(theControl^^.contrlData)^^.ColorQuickDrawPresent) then
1276      PixelDepthCheck(theControl);
1277
1278    if (theControl^^.contrlHilite = kInactive) then
1279      DrawControlInactive(theControl)
1280    else
1281      DrawControlActive(theControl);
1282    end;
1283      {of procedure DoInitMessage}
1284
1285  { ############################################################################ DoTestMessage }
1286
1287  function DoTestMessage(theControl : ControlHandle; param : longint) : longint;
1288
1289    var
1290    indicatorRect : Rect;
1291    mouseXY : Point;
1292    theSliderDataHdl : SliderDataHdl;
1293
1294    begin
1295    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1296    indicatorRect := CalcIndicatorRect(theControl);
1297
1298    mouseXY.v := HiWord(param);
1299    mouseXY.h := LoWord(param);
1300
1301    if (PtInRect(mouseXY, indicatorRect)) then
1302      begin
1303      theSliderDataHdl^^.dragMessageFlag := true;
1304      DrawControlActive(theControl);
1305      theSliderDataHdl^^.dragMessageFlag := false;
1306      DoTestMessage := kControlIndicatorPart;
1307      end
1308    else DoTestMessage := 0;
1309
1310    end;
1311      {of procedure DoInitMessage}
1312
1313  { ############################################################################ DoDragMessage }
1314
1315  function DoDragMessage(theControl : ControlHandle) : longint;
1316
1317    var
1318    indicatorRect, slopRect, trackRect : Rect;
1319    indicatorHeight, indicatorHalfHeight, indicatorCentre, trackHeight : integer;
1320    startMouseXY, currentMouseXY : Point;
1321    controlValueRange, differenceMouseY : integer;
1322    ratio : longreal;
1323    myWindowPtr : WindowPtr;
```

```
1324    theErr : OSErr;
1325    theSliderDataHdl : SliderDataHdl;
1326
1327    begin
1328    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1329    theSliderDataHdl^^.dragMessageFlag := true;
1330
1331    HLock(Handle(theControl));
1332
1333    indicatorHeight := kIndicatorHeight;
1334    indicatorHalfHeight := indicatorHeight div 2;
1335
1336    trackRect := theControl^^.contrlRect;
1337    InsetRect(trackRect, 0, indicatorHalfHeight + 4);
1338    trackRect.bottom := trackRect.bottom + 1;
1339    trackHeight := trackRect.bottom - trackRect.top;
1340
1341    controlValueRange := theControl^^.contrlMax - theControl^^.contrlMin;
1342    ratio := longreal(controlValueRange / trackHeight);
1343
1344    myWindowPtr := theControl^^.contrlOwner;
1345    slopRect := myWindowPtr^.portRect;
1346
1347    theErr := InstallVBLTask(theControl);
1348    if (theErr = noErr) then
1349      theSliderDataHdl^^.VBLInstallFail := false
1350    else
1351      theSliderDataHdl^^.VBLInstallFail := true;
1352
1353    indicatorRect := CalcIndicatorRect(theControl);
1354
1355    GetMouse(startMouseXY);
1356
1357    while (StillDown) do
1358      begin
1359      GetMouse(currentMouseXY);
1360      differenceMouseY := startMouseXY.v - currentMouseXY.v;
1361
1362      if ((differenceMouseY <> 0) and (PtInRect(currentMouseXY, slopRect))) then
1363        begin
1364        indicatorRect.top := indicatorRect.top - differenceMouseY;
1365        indicatorRect.bottom := indicatorRect.bottom - differenceMouseY;
1366
1367        indicatorCentre := indicatorRect.top + indicatorHalfHeight;
1368
1369        theControl^^.contrlValue := longint(trunc((trackRect.bottom
1370                            - indicatorCentre) * ratio));
1371
1372        if (theControl^^.contrlValue > theControl^^.contrlMax) then
1373          theControl^^.contrlValue := theControl^^.contrlMax;
1374        if (theControl^^.contrlValue < theControl^^.contrlMin) then
1375          theControl^^.contrlValue := theControl^^.contrlMin;
1376
1377        DrawControlActive(theControl);
1378
1379        startMouseXY := currentMouseXY;
1380        end;
1381      end;
1382
1383    if not (theSliderDataHdl^^.VBLInstallFail) then
1384      RemoveVBLTask(theControl);
1385
1386    theSliderDataHdl^^.dragMessageFlag := false;
1387    DrawControlActive(theControl);
1388
1389    HUnlock(Handle(theControl));
1390
1391    DoDragMessage := 1;
1392    end;
1393      {of procedure DoInitMessage}
1394
1395  { ###################################################################### DoDisposeMessage }
1396
1397  procedure DoDisposeMessage(theControl : ControlHandle);
1398
1399    var
```

```
1400    theRect : Rect;
1401    theSliderDataHdl : SliderDataHdl;
1402
1403    begin
1404    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1405    theRect := theControl^^.contrlRect;
1406    theRect.right := theRect.right + (theRect.right - theRect.left);
1407    EraseRect(theRect);
1408
1409    if (theSliderDataHdl^^.offScreenPort <> nil) then
1410      DisposeGWorld(theSliderDataHdl^^.offScreenPort);
1411
1412    if (theControl^^.contrlData <> nil) then
1413      DisposeHandle(theControl^^.contrlData);
1414    end;
1415      {of procedure DoDisposeMessage}
1416
1417  { ################################################################ CreateOffScreenGWorld }
1418
1419  procedure CreateOffScreenGWorld(theControl : ControlHandle);
1420
1421    var
1422    theSliderDataHdl : SliderDataHdl;
1423    resourceOffset : integer;
1424    pixMapHdl : PixMapHandle;
1425    pictureHdl : PicHandle;
1426    currentPortDepth : integer;
1427    ignored : QDErr;
1428    ignoredBool : boolean;
1429
1430    begin
1431    resourceOffset := 0;
1432    currentPortDepth := 1;
1433
1434    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1435
1436    theSliderDataHdl^^.compositeRect := theControl^^.contrlRect;
1437    OffsetRect(theSliderDataHdl^^.compositeRect, - theSliderDataHdl^^.compositeRect.left,
1438               - theSliderDataHdl^^.compositeRect.top);
1439    SetRect(theSliderDataHdl^^.trackActiveRect, 50, 0, 100, 139);
1440    SetRect(theSliderDataHdl^^.trackInactiveRect, 100, 0, 150, 139);
1441    SetRect(theSliderDataHdl^^.indicatorActiveRect, 0, 139, 16, 154);
1442    SetRect(theSliderDataHdl^^.indicatorPressedRect, 16, 139, 32, 154);
1443    SetRect(theSliderDataHdl^^.indicatorInactiveRect, 32, 139, 48, 154);
1444    SetRect(theSliderDataHdl^^.offScreenPortRect, 0, 0, 150, 154);
1445
1446    GetGWorld(theSliderDataHdl^^.currentPort, theSliderDataHdl^^.currentDevice);
1447
1448    HLock(Handle(theSliderDataHdl));
1449
1450    ignored := NewGWorld(theSliderDataHdl^^.offScreenPort, 0,
1451                theSliderDataHdl^^.offScreenPortRect, nil, nil, 0);
1452
1453    pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.offScreenPort);
1454    ignoredBool := LockPixels(pixMapHdl);
1455
1456    SetGWorld(theSliderDataHdl^^.offScreenPort, nil);
1457
1458    EraseRect(theSliderDataHdl^^.offScreenPortRect);
1459
1460    if (theSliderDataHdl^^.ColorQuickDrawPresent) then
1461      begin
1462      pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.currentPort);
1463      currentPortDepth := pixMapHdl^^.pixelSize;
1464      end;
1465
1466    if (not (theSliderDataHdl^^.ColorQuickDrawPresent) or (currentPortDepth < 2)) then
1467      resourceOffset := 1;
1468
1469    pictureHdl := GetPicture(rTrackPict + resourceOffset);
1470    if (pictureHdl <> nil) then
1471      begin
1472      HNoPurge(Handle(pictureHdl));
1473      DrawPicture(pictureHdl, theSliderDataHdl^^.offScreenPortRect);
1474      HPurge(Handle(pictureHdl));
1475      end;
```

```
1476
1477    SetGWorld(theSliderDataHdl^^.currentPort, theSliderDataHdl^^.currentDevice);
1478    UnlockPixels(pixMapHdl);
1479    HUnlock(Handle(theSliderDataHdl));
1480
1481    end;
1482      {of procedure CreateOffScreenGWorld}
1483
1484  { ############################################################################ PixelDepthCheck }
1485
1486  procedure PixelDepthCheck(theControl : ControlHandle);
1487
1488    var
1489    theSliderDataHdl : SliderDataHdl;
1490    pixMapHdl : PixMapHandle;
1491    currentPortDepth, gworldPortDepth : integer;
1492
1493    begin
1494    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1495
1496    pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.currentPort);
1497    currentPortDepth := pixMapHdl^^.pixelSize;
1498    pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.offScreenPort);
1499    gworldPortDepth := pixMapHdl^^.pixelSize;
1500
1501    if (currentPortDepth <> gworldPortDepth) then
1502      begin
1503      DisposeGWorld(theSliderDataHdl^^.offScreenPort);
1504      CreateOffScreenGWorld(theControl);
1505      end;
1506
1507    end;
1508      {of procedure CreateOffScreenGWorld}
1509
1510  { ############################################################################ DrawControlActive }
1511
1512  procedure DrawControlActive(theControl : ControlHandle);
1513
1514    var
1515    oldForeColour, oldBackColour : RGBColor;
1516    theSliderDataHdl : SliderDataHdl;
1517    myWindowPtr : WindowPtr;
1518    pixMapHdl : PixMapHandle;
1519    indicatorRect : Rect;
1520    ignoredBool : boolean;
1521
1522    begin
1523    GetForeColor(oldForeColour);
1524    GetBackColor(oldBackColour);
1525
1526    HLock(Handle(theControl));
1527
1528    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1529    HLock(Handle(theSliderDataHdl));
1530
1531    myWindowPtr := WindowPtr(theControl^^.contrlOwner);
1532    SetPort(myWindowPtr);
1533
1534    pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.offScreenPort);
1535    ignoredBool := LockPixels(pixMapHdl);
1536
1537    ForeColor(blackColor);
1538    BackColor(whiteColor);
1539
1540    CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1541             GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1542             theSliderDataHdl^^.trackActiveRect, theSliderDataHdl^^.compositeRect,
1543             srcCopy, nil);
1544
1545    indicatorRect := CalcIndicatorRect(theControl);
1546    OffsetRect(indicatorRect, -theControl^^.contrlRect.left,
1547            -theControl^^.contrlRect.top);
1548
1549    if (theSliderDataHdl^^.dragMessageFlag) then
1550      begin
1551      CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
```

```
1552            GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1553            theSliderDataHdl^^.indicatorPressedRect, indicatorRect, srcCopy, nil);
1554      end
1555    else begin
1556      CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1557            GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1558            theSliderDataHdl^^.indicatorActiveRect, indicatorRect, srcCopy, nil);
1559      end;
1560
1561    if ((theSliderDataHdl^^.dragMessageFlag) and
1562          not (theSliderDataHdl^^.VBLInstallFail)) then
1563      begin
1564      if (theSliderDataHdl^^.VBLRec.inVBlankPeriod) then
1565        begin
1566        theSliderDataHdl^^.VBLRec.inVBlankPeriod := false;
1567
1568        CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1569            GrafPtr(myWindowPtr)^.portBits, theSliderDataHdl^^.compositeRect,
1570            theControl^^.contrlRect, srcCopy, nil);
1571        end;
1572      end
1573    else begin
1574      CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1575             GrafPtr(myWindowPtr)^.portBits, theSliderDataHdl^^.compositeRect,
1576             theControl^^.contrlRect, srcCopy, nil);
1577      end;
1578
1579    UnlockPixels(pixMapHdl);
1580    HUnlock(Handle(theSliderDataHdl));
1581    HUnlock(Handle(theControl));
1582
1583    RGBForeColor(oldForeColour);
1584    RGBBackColor(oldBackColour);
1585    end;
1586      {of procedure CreateOffScreenGWorld}
1587
1588  { ################################################################## DrawControlInactive }
1589
1590  procedure DrawControlInactive(theControl : ControlHandle);
1591
1592    var
1593    oldForeColour, oldBackColour : RGBColor;
1594    theSliderDataHdl : SliderDataHdl;
1595    myWindowPtr : WindowPtr;
1596    pixMapHdl : PixMapHandle;
1597    indicatorRect : Rect;
1598    ignoredBool : boolean;
1599
1600    begin
1601    GetForeColor(oldForeColour);
1602    GetBackColor(oldBackColour);
1603
1604    HLock(Handle(theControl));
1605
1606    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1607    HLock(Handle(theSliderDataHdl));
1608
1609    myWindowPtr := WindowPtr(theControl^^.contrlOwner);
1610    SetPort(myWindowPtr);
1611
1612    pixMapHdl := GetGWorldPixMap(theSliderDataHdl^^.offScreenPort);
1613    ignoredBool := LockPixels(pixMapHdl);
1614
1615    ForeColor(blackColor);
1616    BackColor(whiteColor);
1617
1618    CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1619           GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1620           theSliderDataHdl^^.trackInactiveRect, theSliderDataHdl^^.compositeRect,
1621           srcCopy, nil);
1622
1623    indicatorRect := CalcIndicatorRect(theControl);
1624    OffsetRect(indicatorRect, -theControl^^.contrlRect.left,
1625              -theControl^^.contrlRect.top);
1626
1627    CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
```

```
1628            GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1629            theSliderDataHdl^^.indicatorInactiveRect, indicatorRect, srcCopy, nil);
1630
1631    CopyBits(GrafPtr(theSliderDataHdl^^.offScreenPort)^.portBits,
1632            GrafPtr(myWindowPtr)^.portBits, theSliderDataHdl^^.compositeRect,
1633            theControl^^.contrlRect, srcCopy, nil);
1634
1635    UnlockPixels(pixMapHdl);
1636    HUnlock(Handle(theSliderDataHdl));
1637    HUnlock(Handle(theControl));
1638
1639    RGBForeColor(oldForeColour);
1640    RGBBackColor(oldBackColour);
1641    end;
1642      {of procedure DrawControlInactive}
1643
1644  { ################################################################## CalcIndicatorRect }
1645
1646  function CalcIndicatorRect(theControl : ControlHandle) : Rect;
1647
1648    var
1649    indicatorHeight, indicatorHalfHeight : integer;
1650    trackRect, indicatorRect : Rect;
1651    trackHeight, controlValue, controlMax, controlMin, indicatorCentre : integer;
1652    ratio : longreal;
1653
1654    begin
1655    indicatorHeight := kIndicatorHeight;
1656    indicatorHalfHeight := indicatorHeight div 2;
1657
1658    trackRect := theControl^^.contrlRect;
1659    InsetRect(trackRect, 0, indicatorHalfHeight + 4);
1660    trackRect.bottom := trackRect.bottom + 1;
1661    trackHeight := trackRect.bottom - trackRect.top;
1662
1663    controlValue := theControl^^.contrlValue;
1664    controlMax := theControl^^.contrlMax;
1665    controlMin := theControl^^.contrlMin;
1666
1667    ratio := longreal((controlValue) / (controlMax - controlMin));
1668    indicatorCentre := trackRect.bottom - integer(trunc(ratio * trackHeight));
1669
1670    SetRect(indicatorRect, trackRect.left, indicatorCentre - indicatorHalfHeight,
1671            trackRect.left + 16, indicatorCentre + indicatorHalfHeight - 1);
1672
1673    CalcIndicatorRect := indicatorRect;
1674    end;
1675      {of function CalcIndicatorRect}
1676
1677  { ######################################################################## InstallVBLTask }
1678
1679  function InstallVBLTask(theControl : ControlHandle) : OSErr;
1680
1681    var
1682    theErr : OSErr;
1683    theSliderDataHdl : SliderDataHdl;
1684
1685    begin
1686    theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
1687    theSliderDataHdl^^.VBLRec.inVBlankPeriod := false;
1688
1689    theSliderDataHdl^^.VBLRec.vblTaskRec.qType := vType;
1690    theSliderDataHdl^^.VBLRec.vblTaskRec.vblAddr := VBLUPP(@TheVBLTask);
1691    theSliderDataHdl^^.VBLRec.vblTaskRec.vblCount := 1;
1692    theSliderDataHdl^^.VBLRec.vblTaskRec.vblPhase := 0;
1693
1694    theSliderDataHdl^^.VBLRec.thisApplicationsA5 := SetCurrentA5;
1695
1696    if (theSliderDataHdl^^.slotVInstallPresent) then
1697      theErr := SlotVInstall(QElemPtr(@theSliderDataHdl^^.VBLRec.vblTaskRec),
1698                  theSliderDataHdl^^.mainSlotNumber)
1699    else
1700      theErr := VInstall(QElemPtr(@theSliderDataHdl^^.VBLRec.vblTaskRec));
1701
1702    InstallVBLTask := theErr;
1703    end;
```

```
    {of function InstallVBLTask}

{ ############################################################################ RemoveVBLTask }

procedure RemoveVBLTask(theControl : ControlHandle);

  var
  ignoredErr : OSErr;
  theSliderDataHdl : SliderDataHdl;

  begin
  theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);

  if (theSliderDataHdl^^.slotVInstallPresent) then
    ignoredErr := SlotVRemove(QElemPtr(@theSliderDataHdl^^.VBLRec.vblTaskRec),
            theSliderDataHdl^^.mainSlotNumber)
  else
    ignoredErr := VRemove(QElemPtr(@theSliderDataHdl^^.VBLRec.vblTaskRec));

  end;
    {of function RemoveVBLTask}

{ ############################################################################ TheVBLTask }

procedure TheVBLTask;

  var
  theVBLRecPtr : VBLRecPtr;
  currentA5 : longint;
  ignoredLong : longint;

  begin
  theVBLRecPtr := VBLRecPtr(GetVBLRec);
  currentA5 := SetA5(theVBLRecPtr^.thisApplicationsA5);

  theVBLRecPtr^.inVBlankPeriod := true;
  theVBLRecPtr^.vblTaskRec.vblCount := 1;

  ignoredLong := SetA5(currentA5);
  end;
    {of function TheVBLTask}

{ ########################################################## CheckSlotVInstallAvailable }

function CheckSlotVInstallAvailable : boolean;

  begin
  CheckSlotVInstallAvailable := CheckTrapAvailable(_SlotVInstall);
  end;
    {of function CheckSlotVInstallAvailable}

{ ############################################################################ CheckTrapAvailable }

function CheckTrapAvailable(theTrap : integer) : boolean;

  var
  theTrapType : TrapType;
  trapMask : integer;
  numToolboxTraps : integer;

  begin
  trapMask := $0800;

  if (BAnd(theTrap, trapMask) > 0) then
    theTrapType := ToolTrap
  else
    theTrapType := OSTrap;

  if (theTrapType = ToolTrap) then
    theTrap := BAnd(theTrap, $07FF);

  if (NGetTrapAddress(_InitGraf, ToolTrap) = NGetTrapAddress($AA6E, ToolTrap)) then
    numToolboxTraps := $0200
  else
    numToolboxTraps := $0400;
```

```
1780    if (theTrap >= numToolboxTraps) then
1781      theTrap := _Unimplemented;
1782
1783    CheckTrapAvailable :=
1784      (NGetTrapAddress(theTrap, theTrapType) <> NGetTrapAddress(_Unimplemented, ToolTrap));
1785
1786    end;
1787      {of function CheckTrapAvailable}
1788
1789  end.
1790    {of unit CDEF2Pascal}
1791
1792  { ############################################################################################ }
```

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 which 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.

CDEFandVBLPascal.p

The constant declaration block

Lines 54-59 establish constants relating to menu IDs and menu item numbers. Lines 60-65 establish constants for various resource IDs. Line 66 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The type declaration block

Lines 72-78 define a data type which will be used by the animated cursor functions. 80-84 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. gColorDisplay 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 96-97 declare variables of the two types defined in the type declaration block. 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 101-103 will be assigned handles to the custom slider and radio button controls.

The procedure DoStopAnimCursor

DoStopAnimCursor is called when the user presses any key.

Line 150 removes the VBL task from the queue. Lines 152-155 free up the memory occupied by the cursors. Line 157 resets the animated-cursor-active flag. Line 158 resets WaitNextEvent's sleep parameter. Line 160 sets the standard arrow cursor and Lines 162-164 erase the advisory text from the window.

The procedure 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 177 gets a pointer to the variable which contains the field holding the pointer to the application's A5 world. Line 178 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 180-184 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 186 resets this field to a value higher that 0 (in this case, 30), otherwise the task will never execute again.

Line 188 restores the saved A5 world pointer.

The procedure DoInstallSystemVBLTask

DoInstallSystemVBLTask installs the system-based VBL task.

Lines 200-203 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 205 gets the pointer to the application's A5 world and saves it to the appropriate field of a global variable.

Line 207 installs the specified VBL task into the system-based queue.

The function 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.

The procedure DoStartAnimCursor

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

Line 251 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 252 sets the variable which is used as the sleep parameter in the WaitNextEvent call to a value less than that assigned to gVBLCount.

Line 254 calls an application-defined routine which loads the 'acur' resource, together with the 'CURS' resources specified in the 'acur' resource.

Line 257 calls an application-defined function which installs a system-based VBL task.

Line 259 sets a flag which indicates whether the animated cursor is currently active and Lines 261-264 draw some advisory text in the window.

The procedure 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 287-292 is simply to accommodate the possibility that the user may change the pixel depth while the program is running.

The procedure DoGetSliderControlSuite

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

Line 350 gets the custom slider control. Line 349 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 353-354 get the radio button controls and call an application-defined routine which draws a titled box around the three controls in the suite.

The procedure DoMenuChoice

DoMenuChoice handles menu choices.

Note that DisposeWindow (Line 391) 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.

The procedure DoInContent

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

Line 419 converts the mouse-down location to local coordinates. The call to FindControl at Line 421 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 423), and if the part code was the indicator (Line 425), TrackControl is called (Line 426) 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 428-434 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 436), TrackControl is called at Line 438 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 440), Line 442 makes the slider control active, and Lines 443-444 toggle the Start and Stop radio button control values. If the mouse-down was within the Stop radio button (Line 446), Line 448 sets the slider control's value to 0, Line 449 makes the slider control inactive, and Lines 450-451 toggle the Start and Stop radio button control values.

Note that, when Line 426 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.

The procedures DoActivateWindow, DoOSEvent, DoUpdate, DoActivate, DoMouseDown, and DoEvents

DoEvents, DoMouseDown, DoUpdate, DoActivate, DoActivateWindow, and DoOSEvent perform minimal event handling consistent with the requirements of the demonstration.

Note the calls to DoDrawControlsPanel at Lines 484 and 521. Note also, at Lines 473-474, that the slider control is only highlighted when the window is becoming active if the Start radio button's control value is 1.

The main program block

The main function firstly initialises the system software managers (Line 631). At Lines 635-644, the global variable gColorQuickDrawPresent is set to true if Color QuickDraw is present and, if Color QuickDraw is present, gColorDisplay is set to true if the pixel depth of the main device is greater than 1. The menus are set up at Lines 648-658), a window is opened (Lines 662-674), and an application-defined procedure is called to get the three controls (Line 678). The main event loop is then entered with gSleepTime set to kMaxLong (Lines 683-689).

CDEF1Pascal.p

CDEF1Pascal.p is the source code for the custom radio button control definition function (CDEF). The CDEF responds only to the initCntl, drawCntl, and testCntl messages.

The constant declaration block

Lines 737-742 establish constants representing the resource IDs for six 'cicn' (colour icon) resources. Line 743 establishes a constant representing the part code returned in the DoTestMessage function.

The type declaration block

Global variables are defined at Lines 761-769. gTheQDGlobalsPtr will be assigned the address of the host application's QuickDraw globals. gColorQuickDrawPresent will be set to false if Color QuickDraw is not present. gColorDisplay will be set to true if the pixel depth of the main device is greater than 1. The global variables at Lines 764-769 will be assigned handles to the CIcon records for the six colour icons.

The function main

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

Lines 792 and 817 have to do with enabling the CDEF to access its own global variables.

Note that, in the case of a drawCntl message (Lines 800-804), no action is taken if the contrlVis field of the control's control record specifies that the control is currently invisible.

The procedure DoInitMessage

DoInitMessage handles initCntl messages to completion.

The procedure 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 831-832).

Lines 836-839 check whether Color QuickDraw is present and set the global variable ColorQuickDrawPresent accordingly.

The procedure DoDrawMessage

DoDrawMessage performs the initial handling of drawCntl messages.

Line 865 gets a pointer to the graphics port of the window in which the control resides. Lines 867-871 save the current drawing environment.

Line 873 sets the graphics port and Line 875 obtains the current value of the control. Lines 877-879 establish a rectangle the required size of the radio button image and positioned at the top left of the control's rectangle. Lines 881-883 set the font to Chicago 12 point.

If Color QuickDraw is present (Line 885), Lines 987-898 load the colour icons if they are not currently in memory.

Lines 901-906 set the global variable gColorDisplay 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 908-911)

Lines 913-917 restore the previously save drawing environment.

The procedure DrawColour

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

Line 939 saves the current foreground colour

If the control is inactive (Line 941), 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 943-946) Depending on the current control value, the appropriate inactive state colour icon is then drawn (Lines 948-951).

If the control is active (Line 953), the following occurs. The foreground colour is set to black and the text drawing mode is set to the default (Lines 955-956). Depending on the current control value, the appropriate active state colour icon is then drawn (Lines 958-962).

If the contrlHilite element of the control record contains partCode (see the procedure DoTestMessage) (Line 963), the appropriate "mouse is currently down within the control rectangle" icon is drawn (Lines 965-968).

Lines 971-972 then draw the control's title at the appropriate location, following which the foreground colour saved at Line 939 is restored (Line 974).

The procedure DrawMono

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

Lines 983-984 set the foreground and background colours. Line 986 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 988), the following occurs. If the control is inactive, the pen pattern and the text drawing pattern are both set to gray (Lines 990-994), otherwise they are set to black and the default respectively (Lines 996-997). A framed circle is then drawn (Line 1000), and the interior of this circle is drawn in white (Lines 1002-1004). 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 1006-1014).

If the the contrlHilite field of the control record contains partCode (Line 1026), another circle is drawn 1 pixel inside the main circle.

Lines 1023-1024 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.

The procedure DoTestMessage

DoTestMessage handles testCntl messages to completion.

Lines 1037-1040 extract the mouse-down (local) coordinates from the param parameter. Lines 1042-1045 test whether the mouse-down was within the control's rectangle, returning partCode if it was or 0 otherwise.

CDEF2Pascal.p

CDEF2Pascal.p 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, dispCntl, and autoTrack messages.

The constant declaration block

kInactive represents the value used to make a control inactive. kIndicatorHeight is the nominal height of the slider control's indicator in pixels. Line 1110 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.

The type declaration block

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. It also contains some additional fields which could have been declared as global variables (as in CDEF1) but which have been made elements of the SliderData data type in this program as a means of showing an alternative to the use of global variables. A handle to a SliderDataRec record will be assigned to contrlData field of the control's control record.

ColorQuickDrawPresent 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), mainSlotNumber 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.

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

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

VBLRec is used by the VBL task functions.

The function main

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

Although no global variables are used in this program, Lines 1182 and 1221 remain necessary. In this program, the Line 1182 call is required so as to prevent Gestalt returning -5551 (undefined selector).

Lines 1184 and 1218 save and restore the pen state.

Note that, in the case of a drawCntl message (Line 1193), no action is taken if the contrlVis field of the control's control record specifies that the control is currently invisible.

The procedure DoInitMessage

DoInitMessage performs the initial handling of initCntl messages.

Line 1238 allocates a relocatable block for a slider control data structure, the handle to which is assigned to the control record's contrlData field.

Lines 1247-1249 check whether Color QuickDraw is present and set ColorQuickDrawPresent accordingly.

Line 1251 locks the control's handle. Line 1253 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 1255 unlocks the control's handle.

Line 1257 checks for the availability of the trap SlotVInstall. If that trap is available, Line 1258-1264 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.)

The procedure 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 1275), 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 1278), a CDEF-defined function is called to draw the control in its inactive state (Line 1279), otherwise a CDEF-defined function is called to draw the control in the active state (Line 1281).

The function DoTestMessage

DoTestMessage handles testCntl messages to completion.

Line 1296 calls a CDEF-defined routine 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 1298-1299 extract the mouse-down (local) coordinates from the param parameter. If the mouse-down was within the indicator rectangle (Line 1301), dragMessageFlag is set to true, the control is drawn, dragMessageFlag is set to false again, and kIndicatorControlPart 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 1308), 0 is returned.

The function 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 1329 sets gDragMessageFlag to true to record that a dragCntl message is currently being processed. Line 1331 locks the control's handle.

Lines 1333-1334 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 1339). Line 1341 determines the range of control values, that is, the difference between the control's maximum and minimum values. Line 1342 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 1344-1345 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 1347 calls a CDEF-defined function which installs a VBL task. Lines 1348-1351 set a SliderDataHdl field to record whether or not the installation was successful.

Line 1353 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 1355 gets the coordinates of the mouse position at the time that the mouse button was pressed.

Line 1357 initiates the custom dragging loop, which will continue until the mouse button is released. Within the loop:

- Line 1359 gets the current mouse coordinates and Line 1360 establishes whether the mouse cursor has moved vertically since the last time Line 1359 executed.
- If the cursor has moved vertically and if the cursor is still within the slop rectangle (Line 1362):
    - Lines 1364-1365 adjust the top and bottom of the indicator rectangle to reflect the change in mouse cursor position, and Line 1367 gets the new indicator centre.
    - Line 1367 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 1372-1375 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 1377 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 1379), and the loop continues.

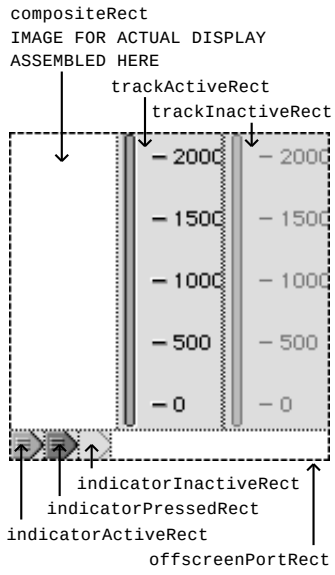
When the mouse button is released, the loop exits and Lines 1383-1384 remove the VBL task if the previous installation was successful. Line 1386 sets dragMessageFlag to record that the custom dragging loop is no longer being executed. The final call to the drawing function at Line 1387 ensures that the indicator will return to the non-pressed appearance as soon as the mouse button is released. Line 1389 unlocks the control's handle

Line 1391 returns a non-zero value to advise the Control Manager that custom dragging is being employed.

The procedure DoDisposeMessage

DoDisposeMessage handles dispCntl messages. Lines 1405-1406 erase the slider control. Lines 1409-1410 dispose of the offscreen graphics world and Lines 1413-1414 release the memory occupied by the control's custom data.

The procedure 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 1434 gets a handle to the control's custom data. Lines 1436-1444 establish a number of rectangles, which are stored in the fields of the control's custom data record.

Line 1446 saves the current graphics world. Line 1448 locks the control's custom data record handle, Line 1450 creates the offscreen graphics world, and Lines 1453-1454 lock the offscreen buffer in memory preparatory to a drawing operation.

Line 1456 sets the offscreen graphics world as the current graphics world and Line 1458 cleans the slate.

Lines 1460-1467 have to do with determining which 'PICT' resource (colour or black and white) gets loaded at Line 1469. Firstly, if Color QuickDraw is present (Line 1460), the pixel depth of the current port is obtained (Lines 1462-1463). 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 1466-1467).

Lines 1469-1475 read in the appropriate 'PICT' resource and draw the picture in the offscreen graphics port as shown in the diagram.

Line 1477 sets the graphics world saved at Line 1446 as the current graphics world. Line 1478 unlocks the offscreen buffer and Line 1479 unlocks the handle to the control's data record.

The procedure 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 1494 gets a handle to the control's custom data. Lines 1496-1499 get the two pixel depths. If they are not the same (Line 1501), Lines 1503-1504 destroy and recreate the offscreen graphics world. (Because the pixelDepth parameter in the NewGWorld call at Line 1450 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.)

The procedure 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 1523-1524 save the current foreground and background colours. Line 1526 locks the control record and Lines 1528-1529 lock the control's custom data. Lines 1531-1532 set the window which owns the control as the current graphics port.

Lines 1534-1535 locks the offscreen buffer and Lines 1537-1538 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 1540-1543 copy the active track image to the composite area of the offscreen graphics world. Lines 1545-1547 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 1549-1559). 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 (Lines 1561-1562), and if the VBL task has just executed (Line 1564), 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 1566). 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 1564, if that line indicates that the VBL task has not executed since the last time the execution flag was set to false, the procedure returns to DoDragMessage without any CopyBits call being executed that time around. This fruitless cycle will continue until Line 1564 indicates that the VBL task has just executed.

Going back to Line 1562, 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 1573-1577 execute in that circumstance.

Lines 1573-1577 are also executed, and Lines 1564-1571 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 1579 unlocks the offscreen buffer, Line 1580 unlocks the control's data record and Line 1581 unlocks the control record. Lines 1583-1584 restore the foreground and background colours saved at Lines 1523-1524.

The procdure DrawControlInactive

DrawControlInactive is called by DoDrawMessage if the contrlHilite field of the control record indicates that the control is currently inactive. This procedure 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.

The function CalcIndicatorRect

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

Lines 1655-1656 gets the half-height of the indicator. Lines 1658-1661 calculate the effective track height, that is, the number of pixels over which the indicator is permitted to move.

Lines 1663-1665 extract the control's current value, and its maximum and minimum values, from the control record.

Lines 1667-1668 set the centre of the indicator's rectangle to reflect the control's current value and Lines 1670-1671 finish the job of setting the new coordinates of the indicator's rectangle. Line 1673 returns that rectangle to the calling function.

The function InstallVBLTask

InstallVBLTask is called by DoDragMessage to install the VBL task.

Line 1687 sets to false the flag which is set to true when the VBL task executes. Lines 1689-1692 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 1694 saves the pointer to the A5 world.

If the trap SlotVInstall is available, the VBL task installed in the slot-based queue (Lines 1696-1698), otherwise it is inserted into the system-based queue (Line 1700). The success, or otherwise, of the attempted installation is returned to the calling function (Line 1702).

The procedure RemoveVBLTask

RemoveVBLTask simply removes the VBL task from the relevant queue.

The procedure TheVBLTask

TheVBLTask is the VBL task itself.

Line 1736 gets a pointer to the variable which contains the field holding the pointer to the application's A5 world. Line 1737 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 1739 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 1740 sets the vblCount field of the VBL task record back to 1. Line 1742 sets the current A5 world back to that saved at Line 1737.

The functions CheckSlotVInstallAvailable and CheckTrapAvailable

CheckSlotVInstallAvailable is called from Line 1257. CheckTrapAvailable is called to check for the availability of the trap SlotVInstall. This code is repeated and explained at Chapter 22 — Miscellany.

Creating the CDEF Resource

To create CDEF resources from code such as that in the listings above for CDEF1Pascal and CDEF2Pascal, follow the same general procedure as is described for LDEFs in Demonstration Program Comments at Chapter 18 — Lists and Custom List Definition Functions.