

# 23

## MISCELLANY

*Includes Demonstration Program Miscellany*

### ***Notification From Applications in the Background***

---

#### ***The Need for the Notification Manager***

---

Applications running in the background cannot use the standard methods of communicating with the user, such as alert or dialog boxes, because such windows might easily be obscured by the windows of other applications. Furthermore, even if these windows are visible, the background application cannot be certain that the user has actually received the communication. Accordingly, some more reliable method must be used to manage communication between a background application and the user. The Notification Manager provides such a method.

#### ***Elements of a Notification***

---

The Notification Manager creates **notifications**. A notification comprises one or more of five possible elements, which occur in the following sequence:

- An **mark** appears against the name of the target application in the Application menu.

This mark is intended to prompt the user to switch the marked application to the foreground. The **◆** mark only appears while the application posting the notification remains in the background. It is replaced by the familiar **\_** mark when that application is brought to the foreground.

- The Application menu title begins alternating between the target application's icon and the foreground application's icon, or the Apple menu title begins alternating between the target application's icon and the Apple icon.

The location of the icon alternation in the menu bar is determined by the posting application's mark (if any). If the application posting the notification is marked by either a **◆** mark or a **√** mark in the Application menu, the Application menu title alternates; otherwise the Apple menu title alternates. Note that several applications might post notifications, so there might be a series of alternating icons.

- The Sound Manager plays a sound.

The application posting the notification can request that the system alert sound be used or it can specify its own sound by passing the Notification Manager a handle to a 'snd' resource.

- A modal alert box appears, and the user dismisses it.

The application posting the notification specifies the text for the modal alert box.

- A response function, if specified, executes.

A response function can be used to remove the notification request from the notification queue (see below) or to perform other processing. For example, it can be used to set a global variable to record that the notification was received.

## ***Suggested Notification Strategy***

---

Apple's suggested notification strategy is to allow the user to set the desired level of notification at one of three levels, as follows:

- **Level 1.** Display the ♦ mark next to the name of the application in the Application menu.<sup>1</sup>
- **Level 2.** Display the ♦ mark next to the name of the application in the Application menu and alternate the icons. (This is the suggested default setting.)
- **Level 3.** Display the ♦ mark next to the name of the application in the Application menu, alternate the icons and invoke an alert box to notify the user that something needs to be done.

A sound might also be played at levels 2 and 3, but the user should have the option of turning the sound off. In addition, the user should be provided with the option of turning notification off altogether, except in cases where damage might occur or data would be lost.

That said, Apple accepts that this suggested strategy might not be appropriate for your application. (Indeed, notifications provided by the system software itself do not follow these guidelines.)

## ***Notifications in Action***

---

### ***Overview***

---

The Notification Manager is automatically initialised at system startup.

To issue a notification to the user, you need to create a **notification request** and install it into the **notification queue**, which is a standard Macintosh queue. The Notification Manager interprets the request and presents the notification to the user at the earliest possible time.

Eventually, you will need to remove the notification request from the notification queue. You can do this in the response function or when your application returns to the foreground.

---

<sup>1</sup> Note that displaying the ♦ mark is only possible if the requesting software is listed in the Application Menu (and thus represents a process which is loaded into memory). The requesting software may not be an application. In addition to applications, other software that is largely invisible to the user can use the Notification Manager. Such software includes device drivers, vertical blanking (VBL) tasks, Time Manager tasks, and code which executes during the system startup sequence, such as code contained in extensions.

## Creating a Notification Request

### The Notification Structure

When installing a request into the notification queue, your application must supply a pointer to a **notification structure**, a static and nonrelocatable structure of type `NMRec` which indicates the type of notification you require. Each entry in the notification queue is, in fact, a notification structure. The notification structure is as follows:

```

type
NMRec = RECORD
  qLink:      QElemPtr;      { Address of next element in queue. (Used internally.) }
  qType:      INTEGER;      { Type of data. (8 = nmType). }
  nmFlags:    INTEGER;      { (Reserved.) }
  nmPrivate:  LONGINT;      { (Reserved.) }
  nmReserved: INTEGER;      { (Reserved.) }
  nmMark:     INTEGER;      { Application to identify with _ mark. }
  nmIcon:     Handle;       { Handle to small icon. }
  nmSound:    Handle;       { Handle to sound structure. }
  nmStr:      StringPtr;    { Pointer to string to appear in alert. }
  nmResp:     NMUPP;        { Pointer to response function. }
  nmRefCon:   LONGINT;      { Available for application use. }
END;

NMRecPtr = ^NMRec;

```

#### Field Descriptions:

To set up a notification request, you need to fill in at least the first six of the following fields:

- |                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>qType</code>   | Indicates the type of operating system queue. Set to <code>nmType</code> (8).                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>nmMark</code>  | Indicates whether to place a ♦ mark next to the name of the application in the Application menu. If <code>nmMark</code> is 0, no mark appears. If <code>nmMark</code> is 1, the mark appears next to the name of the calling application. If <code>nmMark</code> is neither 0 nor 1, it is interpreted as the reference number of a desk accessory. An application should set <code>nmMark</code> to 1 and a driver or detached background task (such as a VBL task or Time Manager task) should set <code>nmMark</code> to 0. |
| <code>nmIcon</code>  | A handle to an icon family containing a small colour icon, that is to alternate periodically in the menu bar. If <code>nmIcon</code> is set to <code>nil</code> , no icon appears in the menu bar. This handle must be valid at the time the notification occurs. It does not need to be locked, but it must be non-purgeable.                                                                                                                                                                                                 |
| <code>nmSound</code> | A handle to a sound resource to be played with <code>SndPlay</code> . If <code>nmSound</code> is set to <code>nil</code> , no sound is produced. If <code>nmSound</code> is set to -1, the system alert sound is played. This handle does not need to be locked, but it must be non-purgeable.                                                                                                                                                                                                                                 |
| <code>nmStr</code>   | Points to a string which appears in the alert box. If <code>nmStr</code> is set to <code>nil</code> , no alert box appears. Note that the Notification Manager does not make a copy of this string, so your application should not dispose of this storage until it removes the notification request.                                                                                                                                                                                                                          |
| <code>nmResp</code>  | A universal procedure pointer to a response function. If <code>nmResp</code> is set to <code>nil</code> , no response function executes when the notification is posted. If <code>nmResp</code> is set to -1, then a pre-defined function removes the notification request immediately after it has completed.                                                                                                                                                                                                                 |

If you do not need to do any processing in response to the notification, you should set `nmResp` to `nil`. If you supply the universal procedure pointer to your own response function, the Notification Manager passes it one

parameter: a universal procedure pointer to your notification structure. For example, this is how you would declare a response function having the name `theResponse`:

```
procedure theResponse(nmStructurePtr : NMUPP);
```

You can use response functions to remove notification requests from the notification queue, free any memory<sup>2</sup>, or set a global variable in your application to record that the notification was posted<sup>3</sup>. If you are setting a global variable to enable you to determine that the user actually received the notification, you need to request an alert notification. This is because the response function executes only after the user has clicked the OK button in the alert box.

If you choose audible or alert notifications, you should probably set `nmResp` to -1 so that the notification structure is removed from the queue as soon as the sound has finished or the user has dismissed the alert box. However, if either `nmMark` or `nmIcon` is non-zero, do not set `nmResp` to -1, because the Notification Manager will remove the `mark` or the `icon` before the user sees it.

`nmRefCon` A long integer available for your application's own use.

### **Installing a Notification Request**

`NMInstall` is used to add a notification request to the notification queue. The following is an example call:

```
osErr := NMInstall(@notificationStructure);
```

Before calling `NMInstall`, you should make sure that your application is running in the background. If your application is in the foreground, you simply use standard alert methods, rather than the Notification Manager, to gain the user's attention.

### **Removing a Notification Request**

`NMRemove` is used to remove a notification request from the notification queue. The following is an example call:

```
osErr := NMRemove(@notificationStructure);
```

You can remove requests at any time, either before or after the notification actually occurs.

## **Progress Bars and Scanning for Command-Period Key-Down Events and Mouse-Down Events**

### **Progress Bars**

Operations within an application which tie up the machine for relatively brief periods of time should be accompanied by a cursor shape change to the watch cursor, or perhaps to an animated cursor. On the other hand, lengthy operations should be accompanied by the display of a progress indicator.

The progress indicator control was described at Chapter 14 — More On Controls. A progress indicator created using this control may be determinate or indeterminate.

<sup>2</sup> Note that an `nmResp` value of -1 does not free the memory block containing the queue element; it merely removes that element from the notification queue.

<sup>3</sup> When the Notification Manager calls your response function, it does not set up A5 or low-memory globals for you. If you need to access your application's global variables, you should save its A5 in the `nmRefCon` field.

Determinate progress indicators show how much of the operation has been completed. Indeterminate progress indicators show that an operation is occurring but does not indicate its duration. Ordinarily, progress indicators should be displayed within a dialog box.

As stated at Chapter 2 — Low and Operating System Events, your application should allow the user to cancel a lengthy operation using the Command-period key combination. You might also include a Stop push button in the dialog box in which the progress indicator is located.

## ***Scanning for Command-Period Key-Down Events and Mouse-Down Events in a Stop Button***

---

One way to satisfy this requirement is to periodically call an application-defined routine which scans the event queue for Command-period key-down events and mouse-down events. This routine should return `true` if:

- A Command-period keyboard event is found.
- A mouse-down event is found and the mouse-down was within the Stop button's rectangle.

The application-defined routine should first get a pointer to the first queue element. It should then scan the queue for key-down and mouse-down events.

If a key-down event is found, the next step is to determine whether the Command key was down at the time of the key press. If it was, a check should be made as to whether the key pressed was the period key. If these checks reveal that a Command-period keyboard event has occurred, the function should return immediately, returning `true` to the calling function.

If a mouse-down event is found, the next step is to determine whether the mouse-down was within the Stop push button's rectangle and, if so, briefly highlight the push button before returning `true` to the calling function.

If `true` is returned to the calling function, that function should terminate the lengthy operation and close the progress indicator dialog box.

## ***Soliciting a Colour Choice From the User — The Color Picker***

---

The Color Picker Utilities provide your application with:

- A standard dialog box, called the **Color Picker**, for soliciting a colour choice from the user.
- Functions for converting colour specifications from one **colour model** to another.

### ***Preamble - Colour Models***

---

In the world of colour, three main colour models are used to specify a particular colour. These are the RGB (red, green, blue) model, the CYMK (cyan, magenta, yellow, black) model, and the HLS or HSV (hue, lightness, saturation, or hue, saturation, value) models.

#### ***RGB Model***

---

The RGB model is used where light-produced colours are involved, as in the case of a television set, computer monitor, or stage lighting. In this model, the three primary colours involved (red, green, and blue) are said to be *additive* because, the more of each colour you add, the closer the resulting colour is to white.

## CYMK Model

---

The CYMK model is closely associated with printing, that is, putting colour on a white page. In this model, the three primary colours (cyan, yellow, and magenta<sup>4</sup>) are said to be *subtractive* because, the more of each colour you add, the closer the resulting colour is to black. (The inclusion of black in the model accounts for the fact that the colours of printer's inks may vary slightly from true cyan, yellow, and magenta, meaning that a true black may not be achievable with just a CYM model.)

## HLS and HSV Models

---

The HLS and HSV models separate colour (that is, hue) from saturation and brightness. Saturation is a measure of the amount of white in a colour (the less white, the more saturated the colour). Lightness is the measure of the amount of black in a colour. (The less black, the lighter the colour). The amount of black is specified by the lightness (L) value in the HLS model and by the value (V) value in the HSV model.

The HSL/HLV model may be represented diagrammatically by the HSL/HLV colour cone shown at Fig 1. In this colour cone, hue is represented by an angle between 0° and 360°.

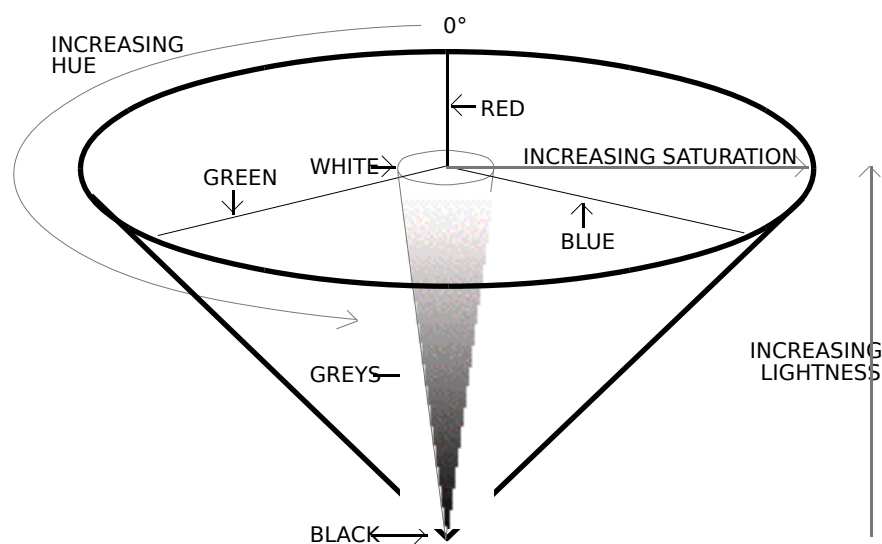


FIG 1 - HSL/HSV COLOUR CONE

## The Color Picker

---

The Color Picker allows the user to specify a colour using either the RGB, CMYK, HLS, or HSV, models.

### Using the Color Picker RGB Mode

---

Fig 2 shows the Color Picker in RGB mode. The desired red, green and blue values may be set using the three slider controls or may be entered directly into the edit text fields on the right of the sliders.

<sup>4</sup> Cyan, magenta, and yellow are the complements of red, green, and blue.

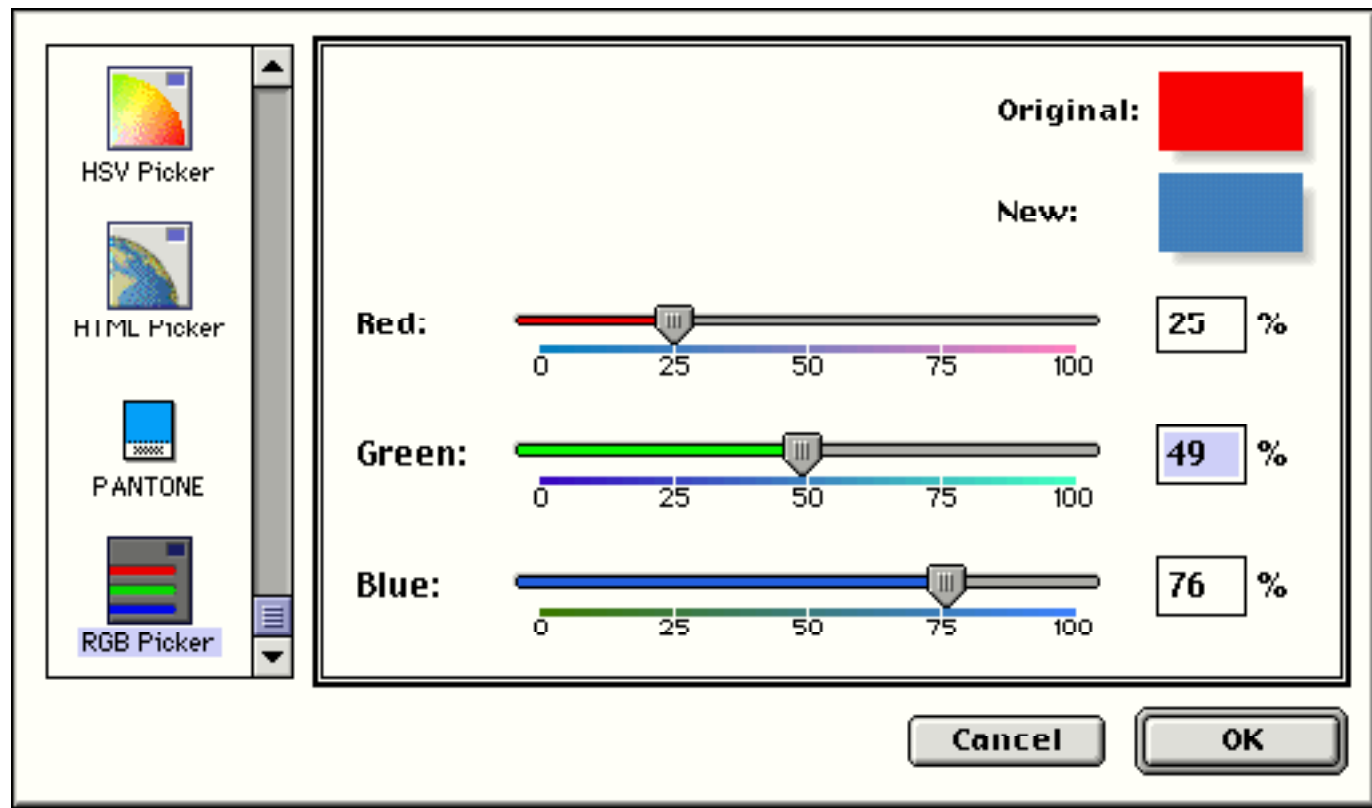


FIG 2 - COLOR PICKER DIALOG IN RGB MODE

### Using the Color Picker in HLS Mode

Fig 3 shows the Color Picker in HLS mode. Hue is specified by an angle, which may be entered at Hue Angle:. Saturation is specified by percentage, which may be entered at Saturation:. Lightness is also specified by a percentage, which may be entered at Lightness: Alternatively, hue and saturation may be selected simultaneously by clicking at the desired point within the coloured disc, and lightness may be set with the slider control.

To relate Fig 3 to Fig 1, the coloured disc at Fig 3 may be considered as the HSL/HSV cone as viewed from above. The lightness slider control can then be conceived of as moving the disc up or down the axis of the cone from the apex (black) to the base (white).

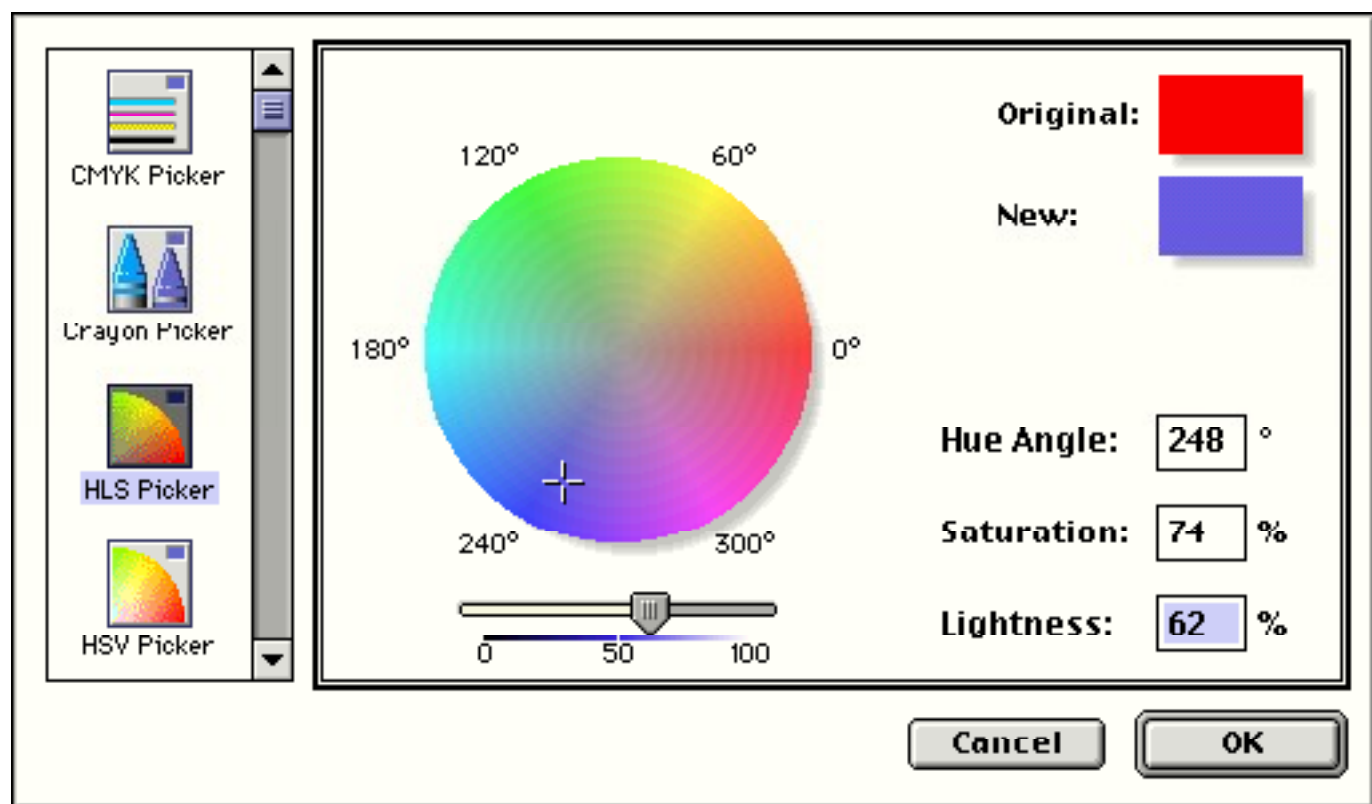


FIG 3 - COLOR PICKER IN HLS MODE

## Invoking the Color Picker

---

The Color Picker is invoked using the `GetColor` function:

```
function GetColor(where : Point; prompt : ConstStr255Param;
                 {const} varinColor : RGBColor;
                 var outColor : RGBColor ) : boolean;
```

|                 |                                                                                                                                                                                               |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>where</i>    | Dialog's upper-left corner. (0,0) causes the dialog box to be positioned centrally on the main screen.                                                                                        |
| <i>prompt</i>   | A prompt string, which is displayed in the upper left corner of the main pane in the dialog box.                                                                                              |
| <i>inColor</i>  | The starting colour, which the user may want for comparison, and which is displayed against Original: in the top right corner of the dialog box.                                              |
| <i>outColor</i> | Initially set to equal <i>inColor</i> . Assigned a new value when the user picks a colour. The colour stored in this parameter is displayed at the top right of the dialog box against New:.) |

**Returns:** A Boolean value indicating whether the user clicked on the OK button or Cancel button.

If the user clicks the OK button in the Color Picker dialog, your application should adopt the `outColor` value as the colour chosen by the user. If the user clicks the Cancel button, your application should assume that the user has decided to make no colour change, that is, the colour should remain as that represented by the `inColor` parameter.

## Coping With Multiple Monitors

---

### Overview

---

Many Macintosh models can accommodate more than one monitor. In a multi-monitor system, the Monitors control panel allows the user to specify which of the attached monitors is to be the **main screen** (that is, the screen containing the menu bar) and to set the position of the other screen, or screens, relative to the main screen.

The maximum number of colours capable of being displayed by a given Macintosh at the one time is determined by the video capability of that particular Macintosh. The maximum number of colours capable of being displayed on a given screen at the one time depends on settings made by the user using the Monitors and Sound control panel. In a multi-monitor environment, therefore, it is possible for each screen to be set to a different pixel depth.

In more technical terms, a Monitors control panel colours/grays setting sets the pixel depth of a particular **video device**. A brief review of the subject of video devices is therefore appropriate at this point.

### Video Devices Revisited

---

As stated at Chapter 11 — QuickDraw Preliminaries:

- A **graphics device** is anything into which QuickDraw can draw, a **video device** (such as a plug-in video card or a built-in video interface) is a graphics device that controls screens, Color QuickDraw stores information about video devices in `GDevice` structures, the system creates and initialises a `GDevice` structure for each video device found during start-up<sup>5</sup>, all structures are linked together in a list called the

<sup>5</sup> The Monitors and Sound control panel stores the pixel depth and other configuration information in a resource of type 'scrn' (resource ID 0). This resource contains an array of data structures which are analogous to `GDevice` records. Each



**device list**, and the global variable `DeviceList` holds a handle to the first structure in the list.

- At any given time, one, and only one, graphics device is the **current device**<sup>6</sup>, that is, the one in which the drawing is taking place. A handle to the current device's `GDevice` structure is placed in the global variable `TheGDevice`.

By default, the `GDevice` structure corresponding to the first video device found at start up is marked as the (initial) current device, and all other graphics devices in the list are initially marked as inactive. When the user moves a window to, or creates a window on, another screen, and your application draws into that window, `Color QuickDraw` automatically makes the video device for that screen the current device and stores that information in `TheGDevice`. As `Color QuickDraw` draws across a user's video devices, it keeps switching to the `GDevice` structure for the video device on which it is actively drawing.

Also recall from Chapter 11 — `QuickDraw` Preliminaries that two of the fields in a `GDevice` structure are:

- `gdMap`, which contains a handle to a pixel map which, in turn, contains a field (`pixelSize`) containing the device's pixel depth (that is, the number of bits per pixel).
- `gdRect`, which contains the device's global boundaries.

### Requirements of the Application

Accommodating a multi-monitor environment requires that you address the following issues:

- **Image Optimisation.** To draw a particular graphic, your application may have to call different drawing functions for that graphic depending on the characteristics of the video device intersecting your window's drawing region, the aim being to optimise the appearance of the image regardless of whether it is being displayed on, say, a grayscale device or a colour device. Recall from Chapter 11 — `QuickDraw` Preliminaries that when `QuickDraw` displays a colour on a grayscale screen, it computes the luminance, or intensity of light, of the desired colour and uses that value to determine the appropriate gray value to draw. It is thus possible that, for example, two overlapping objects drawn in two quite different colours on a colour screen may appear in the same shade of gray on a grayscale screen. In order for the user to differentiate between these two objects on a grayscale screen, you would need to provide an alternative drawing function which draws the two objects in different shades of gray on grayscale screens.
- **Window Zooming.** The second issue is window zooming. For example, if the user drags a window currently zoomed to the user state so that it spans two screens, and then clicks the zoom box to zoom the window to the standard state, your application will need to determine which screen contains the largest area of the window, calculate the standard state for that screen (which will depend, amongst other things, on whether that screen contains the menu bar), and finally zoom the window out to the standard state for that particular screen.
- **Window Dragging and Sizing.** In window dragging operations in a single-monitor environment, `&qd.screenBits.bounds` is typically passed in the `limitRect` parameter of `DragWindow`. (`bounds` is a rectangle which encloses the main screen.) Similarly, in window sizing operations in a single-monitor environment, the values in the `bottom` and `right` fields of `bounds` are typically assigned to the `bottom` and `right` fields of the

---

element of this array contains information about a different video device. When `InitGraf` is called to initialize `QuickDraw`, it checks the System file for the 'scrn' resource. If the resource is found, and if it matches the hardware, `InitGraf` organises the video devices according to the resource's contents. If the resource is not found, `QuickDraw` uses only the video device of the startup screen.

<sup>6</sup> The current device is sometimes referred to as the **active device**.

rectangle passed in the `sizeRect` parameter of `GrowWindow`. For a multi-monitor environment, you should use the rectangle in the `rgnBBox` field of the `Region` structure filled in by a call to `LMGetGrayRgn`. This rectangle bounds the current desktop region, which spans multiple monitors.

## Image Optimisation

---

The `QuickDraw` function `DeviceLoop` is central to the matter of optimising the appearance of your images. `DeviceLoop` searches for graphics devices which intersect your window's drawing region, informing your application of each graphics device it finds and providing your application with information about the current device's attributes. Armed with this information, your application can then invoke whichever of its drawing functions is optimised for those particular attributes.

`DeviceLoop`'s second parameter is a pointer to an application-defined function. That function must be defined like this:

```
procedure myDrawingFunction(depth, deviceFlags : Sint16; targetDevice : GDHandle;
                           userData : SInt32)
```

`DeviceLoop` calls this function for each dissimilar video device it finds. If it encounters similar devices (that is, devices having the same pixel depth, colour table seeds, etc.) it will make only one call to `myDrawingFunction`, pointing to the first such device encountered. `DeviceLoop`'s behaviour can, however, be modified by supplying the `flags` parameter with one of the following values:

| Value                       | Meaning                                                                                                                    |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>singleDevices</code>  | Do not group similar devices when calling drawing function.                                                                |
| <code>dontMatchSeeds</code> | Do not consider <code>ctSeed</code> fields of <code>ColorTable</code> structures for graphics devices when comparing them. |
| <code>allDevices</code>     | Ignore value of <code>drawingRgn</code> parameter and instead call drawing function for every screen.                      |

## Window Zooming

---

Handling window zooming in a multi-monitors environment requires that your application provide a special application-defined function. The user may have moved a window to a different screen, or to a position where it spans two separate screens, since it was last zoomed. When the user elects to zoom that window to the **standard state**<sup>7</sup>, your application-defined function must first determine the screen on which the zoomed window is to appear and the appropriate standard state for that screen.

The screen on which the zoomed window should appear should be the screen on which the window is currently displayed or, if the window spans screens, the screen containing the largest area of the window. The appropriate standard state will depend on:

- The device's global boundaries, as contained in the `gdRect` field of the `gDevice` structure.
- The requirements of the application. (As stated at Chapter 4 — Windows, the standard state on the main screen is typically the gray area of the screen minus three pixels all round.)
- Whether the screen on which the zoomed window is to appear contains the menu bar.

After determining the screen on which the zoomed window is to appear and calculating the standard state, your application-defined function should call `ZoomWindow` to redraw the window frame in its new location and, finally, redraw the window's content region.

<sup>7</sup> See Chapter 4 — Windows for a description of standard state, user state, and the state data record.

## 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. 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 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. 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 functions which may do so.
- Applicable to 680x0 code only, a VBL task cannot call a function from any other code segment (see Code Segmentation, below) 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 structure) 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<sup>8</sup>.

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 structure from the VBL queue or by setting the `vblCount` field of the task structure (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 structure or by re-setting the `vblCount` field of the VBL task structure (see below) to the appropriate value.

## **Installing and Removing a VBL Task**

---

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

---

<sup>8</sup> You load a system-based task's VBL 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 Structure

---

The VBL task structure 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 function that the Vertical Retrace Manager is to execute.

`vblCount`    The number of interrupts before the function 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 function 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 VBL task structure 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 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 structure) 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 structure and install it into the appropriate queue.

## **VBL Task Structures Access — 680x0 Code**

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

When the Vertical Retrace Manager executes the VBL task in a 680x0 environment, 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 VBL task structure, 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.

## **VBL Task Structure Access — PowerPC Code**

In the PowerPC environment, the address of the VBL task structure is passed to the to the VBL task as an explicit parameter.

## **Accessing Application Global Variables - 680x0 Code**

Recall from Chapter 1 that the boundary between the current application's global variables and its application parameters are stored in the 680x0 microprocessor's A5 register. Since all 680x0 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 functions, 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 680x0 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 VBL task structure. Accordingly, if your application stores its A5 directly following the VBL task structure, it can locate this value by first locating the VBL task structure. To store the A5 value directly following the VBL task structure, define a new data type whose first field contains the VBL task structure and whose second field will hold the value in the A5 register retrieved by a call to `SetCurrentA5`:

```
type
    VBLRecord = record
        vblTaskStruc : VBLTask;           { The VBL task structure. }
        vblA5 : SInt32;                 { Saved value of A5. }
    end;
    VBLRecordPtr = ^VBLRecord ;
```

You can think of this new data type as an expanded VBL task structure.

## ***Accessing Application Global Variables - PowerPC Code***

---

Setting and restoring the A5 register has no relevance in PowerPC code. In the PowerPC environment, the table of contents register always points to the table of contents for the currently executing code, through which the application's global variables can be addressed. As a result, your application's global variables are transparently available to any code compiled into your application.

## ***Ensuring Compatibility with the Operating Environment***

---

If your application is to run successfully in the software and hardware environments that may be present in a wide range of Macintosh models, it must be able to acquire information about a number of machine-dependent features and, where appropriate, act on that information.

## ***Getting Operating Environment Information - The Gestalt Function***

---

The Gestalt function may be used to acquire a wide range of information about the operating environment.

```
function Gestalt(selector : OSType; var response : SInt32) : OSErr;
```

*selector* Selector code.

*response* 4-byte return result which provides the requested information. When all four bytes are not needed, the result is expressed in the low-order byte.

**Returns:** Error code. (0 = no error.)

The types of information capable of being retrieved by Gestalt are as follows:

- The type of machine.
- The version of the System file currently running.
- The type of CPU.
- The type of keyboard attached to the machine.
- The type of floating-point unit (FPU) installed, if any.
- The type of memory management unit (MMU).
- The size of the available RAM.
- The amount of available virtual memory.
- The versions and features of various drivers and managers.

## ***Gestalt Selectors***

---

To use Gestalt, you pass it a **selector**, which specifies exactly what information your application is seeking. Of those selectors which are pre-defined by the Gestalt Manager, there are two sub-types:

- **Environmental Selectors.** Environmental selectors are those which return information about the existence, or otherwise, of a feature. This information can be used by your application to guide its actions. Some examples of the many available

environmental selectors, and the information returned in the `response` parameter, are as follows:

| <b>Selector</b>                      | <b>Information Returned</b> |
|--------------------------------------|-----------------------------|
| <code>gestaltFPUType</code>          | FPU type.                   |
| <code>gestaltKeyboardType</code>     | Keyboard type.              |
| <code>gestaltLogicalRAMSize</code>   | Logical RAM size.           |
| <code>gestaltPhysicalRAMSize</code>  | Physical RAM size.          |
| <code>gestaltQuickdrawVersion</code> | QuickDraw version.          |
| <code>gestaltTextEditVersion</code>  | TextEdit version.           |

- **Informational Selectors.** Informational selectors are those which provide information which should be used for the user's enlightenment only. This information should never be used as proof positive of some feature's existence, nor should it be used to guide your application's actions. Some example of informational selectors, and the information they return, are as follows:

| <b>Selector</b>                   | <b>Information Returned</b> |
|-----------------------------------|-----------------------------|
| <code>gestaltMachineType</code>   | Machine type.               |
| <code>gestaltROMVersion</code>    | ROM version.                |
| <code>gestaltSystemVersion</code> | System file version.        |

## ***Gestalt Responses***

In almost all cases, the last few characters in the selector's name form a suffix which indicates the type of value that will be returned in the `response` parameter. The following shows the meaningful suffixes:

| <b>Suffix</b>        | <b>Returned Value</b>                                                                                                                                                                                                                                       |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Attr</code>    | A range of 32 bits, the meaning of which must be determined by comparison with a list of constants.                                                                                                                                                         |
| <code>Count</code>   | A number indicating how many of the indicated type of items exist.                                                                                                                                                                                          |
| <code>Size</code>    | A size, usually in bytes.                                                                                                                                                                                                                                   |
| <code>Table</code>   | Base address of a table.                                                                                                                                                                                                                                    |
| <code>Type</code>    | An index describing a particular type of feature.                                                                                                                                                                                                           |
| <code>Version</code> | A version number. Implied decimal points may separate digits of the returned value. For example, a value of <code>0x0750</code> returned in response to the <code>gestaltSystemVersion</code> selector means that system software version 7.5.0 is present. |

## ***Using Gestalt — Examples***

The header file `Gestalt.h` defines and describes Gestalt Manager selectors, together with the many constants which may be used to test the `response` parameter.

### ***Example 1***

For example, when `Gestalt` is used to check whether Version 1.3 or later of Color QuickDraw is present, the value returned in the `response` parameter may be compared with `gestalt32BitQD13` as follows:

```
var
  theErr : OSErr;
  response : SInt32;
  colorQuickDrawVers13Present : boolean;

begin
  colorQuickDrawVers13Present := true;

  theErr := Gestalt(gestaltQuickdrawVersion, response);
  if (theErr = noErr) then
    begin
      if (response < gestalt32BitQD13) then
        begin
```



```

    colorQuickDrawVers1.Present := false;
  end;
end;

```

## Example 2

Many constants in Gestalt.h represent bit numbers. In this example, the value returned in the response parameter is tested to determine whether bit number 5 (gestaltHasSoundInputDevice) is set:

```

var
  theErr : OSErr;
  response : SInt32;
  hasSoundInputDevice : boolean;

begin
  hasSoundInputDevice = false;

  theErr := Gestalt(gestaltSoundAttr, response);
  if (theErr = noErr) then
    begin
      gHasSoundInputDevice := BitTst(@response, 31 - gestaltHasSoundInputDevice);
    end;

```

Note that the function BitTst is used to determine whether the specified bit is set. Bit numbering with BitTst is the opposite of the usual MC680x0 numbering scheme used by Gestalt. Thus the bit to be tested must be subtracted from 31. This is illustrated in the following:

```

Bit numbering as used in BitTst
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Bit as numbered in MC69000 CPU operations, and used by Gestalt
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

gestaltHasSoundInputDevice = 5
31 - 5 = 26

```

## Code Segmentation and Heap Space Optimisation - 680x0 Code

680x0 Macintosh programs may be divided into several **segments**. The Macintosh system software limits segments to 32K; accordingly, if you are writing a large program, you must segment your code.

Observing the 32K limit is, however, not the only reason for segmenting your 680x0 code. Segments equate, in the built application, to units of executable code which are stored in resources of type 'CODE' and which are loaded into your application's heap as relocatable blocks. Because these resources are loaded into memory only when required, and because your application can cause them to be marked as purgeable when no longer needed, segmentation allows you to optimise your 680x0 application's heap space. Put another way, segmentation allows you to provide the user with the maximum possible heap space to accommodate the windows and user data, etc., created while the 680x0 application is running.

The main segment (that is, the segment containing the main function) is loaded and locked by the system when the application is launched. Thereafter, when the application makes a call to a function in one of the remaining segments, the Segment Loader, with no help from the application, automatically loads that segment, moves it high in the application's heap, locks it, and passes control to the called function.

Ultimately, of course, all code segments will be brought into memory and locked, creating the same memory-hogging situation as would obtain if the application had not been segmented. To prevent that situation, your application should, at the appropriate time, unlock these blocks and make them purgeable. Note that this applies to all but the main code segment, which must never be unlocked or made purgeable. The following

describes an appropriate methodology for unlocking and marking as purgeable the other code segments of your application:

- Create a new **stub**, or “do nothing” function, for each of the code segments you want to unload. For example, this is a stub for a code segment called `UpdateSegment`:

```
procedure UpdateSegment;
begin
end;
```

- Include each stub in its associated code segment.
- Write a function called, say, `DoUnloadSegments` which calls the Segment Loader function `UnloadSeg` for each of the stubs. The following is an example:

```
procedure DoUnloadSegments;
begin
  UnloadSeg(UpdateSegment);
  UnloadSeg(ActivateSegment);
  { Other UnloadSeg calls here as required. }
end;
```

Note that each `UnloadSeg` call looks up the code segment that contains the stub function in its input parameter, unlocks that segment, and makes it purgeable. Note also that you could pass any of the segment's functions as the parameter to the `UnloadSeg` call; however, it is preferable to use stubs dedicated to this purpose because the other functions in the segment could well be moved to another segment during future updating of the code.

- Place the `DoUnloadSegments` function in the main code segment and call it at the bottom of the main event loop (which should also be located in the main code segment) so that all code segments specified in the function will be unlocked and marked as purgeable after a received event has been handled to completion. The following is an example:

```
begin
...
while not gDone do
begin
  if WaitNextEvent(everyEvent, theEvent, kMaxLong, nil) then
  begin
    DoEvents(theEvent);
  end;

  DoUnloadSegments;
end;
end.
```

One or more of the unlocked and purgeable code segments may then be purged by the Memory Manager if this becomes necessary in order to satisfy a memory allocation request. When a call is subsequently made to a function contained in one of the purged segments, the Segment Loader once again loads that segment into your application's heap as a relocatable block.

### ***PowerPC Considerations***

---

There is no need to include conditional compilation directives in source code containing segmentation directives before that code is compiled for the PowerPC. Compilers which produce PowerPC code ignore segmentation directives, and any calls to the Segment Manager's `UnloadSeg` function are simply ignored.

## ***Main Notification Manager Data Types and Functions***

---

### ***Data Types***

---

#### ***Notification Structure***

```
NMRec = RECORD
  qLink:      QElemPtr;      { Address of next element in queue. (Used internally.) }
  qType:      INTEGER;       { Type of data. (8 = nmType). }
  nmFlags:    INTEGER;       { (Reserved.) }
  nmPrivate:  LONGINT;       { (Reserved.) }
  nmReserved: INTEGER;       { (Reserved.) }
  nmMark:     INTEGER;       { Application to identify with _ mark. }
  nmIcon:     Handle;        { Handle to small icon. }
  nmSound:    Handle;        { Handle to sound structure. }
  nmStr:      StringPtr;     { Pointer to string to appear in alert. }
  nmResp:     NMUPP;         { Pointer to response function. }
  nmRefCon:   LONGINT;      { Available for application use. }
END;
```

```
NMRecPtr = ^NMRec;
```

### ***Functions***

---

#### ***Add Notification Request to the Notification Queue***

```
function NMInstall(nmReqPtr : NMRecPtr) : OSErr;
```

#### ***Remove Notification Request from the Notification Queue***

```
function NMRemove(nmReqPtr : NMRecPtr) : OSErr;
```

## ***Relevant Process Manager Data Types and Functions***

---

### ***Data Types***

---

#### ***Process Serial Number***

```
ProcessSerialNumber = RECORD
  highLongOfPSN:  UInt32;
  lowLongOfPSN:   UInt32;
END;
```

### ***Functions***

---

#### ***Get Process Serial Number of a Particular Process***

```
FUNCTION GetCurrentProcess(VAR PSN: ProcessSerialNumber): OSErr;
```

#### ***Get Process Serial Number of Foreground Process***

```
FUNCTION GetFrontProcess(VAR PSN: ProcessSerialNumber): OSErr;
```

#### ***Compare Two Process Serial Numbers***

```
FUNCTION SameProcess({CONST}VAR PSN1: ProcessSerialNumber;
  {CONST}VAR PSN2: ProcessSerialNumber; VAR result: BOOLEAN): OSErr;
```

## **Relevant Event Manager Data Types and Functions**

---

### **Data Types**

---

#### **QHdr (Defines the Queue Header)**

```
QHdrPtr = ^QHdr;
QHdr = RECORD
  qFlags:    INTEGER;
  qHead:    QElemPtr;
  qTail:    QElemPtr;
END;
```

#### **QElem**

```
QElemPtr = ^QElem;
QElem = RECORD
  qLink:    QElemPtr;
  qType:    INTEGER;
  qData:    ARRAY [0..0] OF INTEGER;
END;
```

#### **EvQEI (Defines an Entry in the Operating System Event Queue)**

```
EvQEIPtr = ^EvQEI;
EvQEI = RECORD
  qLink:    QElemPtr;
  qType:    SInt16;
  evtQWhat: EventKind;    { This part is identical to the EventRecord as defined above. }
  evtQMessage: UInt32;
  evtQWhen:  UInt32;
  evtQWhere: Point;
  evtQModifiers: EventModifiers;
END;
```

### **Functions**

---

#### **Get Address of Event Queue Header**

```
function LMGetEventQueue : QHdrPtr;
```

## **Relevant Color Picker Utilities Function**

---

```
FUNCTION GetColor(where: Point; prompt: Str255; {CONST}VAR inColor: RGBColor;
  VAR outColor: RGBColor): BOOLEAN;
```

## **Relevant QuickDraw Constants and Functions**

---

### **Constants**

---

#### **Flag Bits for gdFlags Field of GDevice Structure**

```
mainScreen    = 11  { Graphics device is main screen. }
screenDevice  = 13  { Graphics device is a screen device. }
screenActive   = 15  { Graphics device is current device. }
```

### **Functions**

---

#### **Getting Available Graphics Devices**

```
FUNCTION LMGetDeviceList: GDHandle;
FUNCTION LMGetMainDevice: GDHandle;
FUNCTION GetNextDevice: GDHandle;
```



## Version 2.1

```
// MiscellanyInterface.p
//
// This program demonstrates:
//
// • The use of the Notification Manager to allow an application running in the
//   background to communicate with the foreground application.
//
// • The use of the determinate progress indicator control to show progress during a
//   time-consuming operation, together with scanning the event queue for Command-period
//   key-Down events and mouse-Down events for the purpose of terminating the lengthy
//   operation before it concludes of its own accord.
//
// • Image drawing optimisation and window zooming in a multi-monitors environment.
//
// • The use of the Color Picker to solicit a choice of colour from the user.
//
// • System-based and slot-based VBL tasks.
//
// • The use of stubs in 68K code segments, together with a function which uses those
//   stubs to unlock code segments and make them purgeable.
//
// 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) for a window in which graphics
//   and information relevant to the demonstrations is displayed.
//
// • A 'DLOG' resource (purgeable), and associated 'DITL', 'dlgx', and 'dftb' resources
//   (purgeable), for a dialog box in which the progress indicator is displayed.
//
// • 'CNTL' resources (purgeable) for the progress indicator dialog.
//
// • 'icn#', 'ics4', and 'ics8' resources (non-purgeable) which contain the application
//   icon shown in the Application menu during the Notification Manager demonstration.
//
// • A 'snd ' resource (non-purgeable) used in the Notification Manager demonstration.
//
// • A 'STR ' resource (non-purgeable) containing the text displayed in the alert box
//   invoked by the Notification Manager.
//
// • A 'STR#' resource (purgeable) containing the label and narrative strings for the
//   notification-related alert displayed by Miscellany.
//
// • An 'acur' resource and associated 'CURS' resources (purgeable) used in the system-
//   based VBL task demonstration.
//
// • A 'PICT' resource (non-purgeable) used in the slot-based VBL task demonstration.
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents, DoesActivateOnFGSwitch,
//   canBackground, and is32BitCompatible flags set.
//
// Miscellany source code is contained in six files. In the 68K project, each source
// code file is assigned to a different segment. (Note that this small program Does not
// really require such segmentation. The code is segmented only to facilitate the
// demonstration of the Segment Loader aspects, which apply only to the 68K version.)
//
// }
unit MiscellanyInterface;

interface

//
.....
..... includes

uses

  { Universal Interfaces }
  Appearance, ColorPicker, CursorCtl, Devices, Fonts, LowMem, Menus, Processes,
  Retrace, SegLoad, Sound, TextUtils, ToolUtils;

//
.....
..... constants

const
```

```

mApple = 128;
iAbout = 1;
mFile = 129;
iQuit = 11;
mDemonstration = 131;
iNotification = 1;
iProgress = 2;
iColourPicker = 3;
iMultiMonitors = 4;
iSystemVertBlank = 5;
iSlotVertBlank = 6;
rMenuBar = 128;
rWindow = 128;
rAlert = 128;
rDialog = 128;
iStopButton = 1;
iProgressIndicator = 2;
rIconFamily = 128;
rBarkSound = 8192;
rString = 128;
rAlertStrings = 128;
indexLabel = 1;
indexNarrative = 2;
rPicture = 128;

```

```
//
```

```
..... type definitions
```

```
type
```

```

VBLStructure = record
  vblTaskStruc : VBLTask;
  thisApplicationsA5 : SInt32;
  inVBlankPeriod : boolean;
end;
VBLStructurePtr = ^VBLStructure;

```

```
var
```

```

gWindowPtr : WindowPtr;
gProcessSerNum : ProcessSerialNumber;
gMultiMonDragBounds, gMultiMonGrowBounds : Rect;
gSystemVBLTaskDemo : boolean;
gMultiMonitorsDrawDemo : boolean;
gColourPickerDemo : boolean;
gWhiteColour : RGBColor;
gBlueColour : RGBColor;

```

```
end.
```

```
{ of unit MiscellanyInterface }
```

```
{ }

```

```

{ }
// Miscellany.p
// }

```

```
program Miscellany;
```

```
uses
```

```

MiscellanyInterface, ColourPicker, NotificationDemo, MultiMonitor,
ProgressIndicator, VerticalBlank;

```

```
//
```

```
..... global variables
```

```
var
```

```

DoDeviceLoopDrawUPP : DeviceLoopDrawingUPP;
gDone : boolean;

```

```
// ..... main
program block variables
```

```

mainMenuBarHdl : Handle;
mainMenuHdl : MenuHandle;

```





```

    EraseRect(gWindowPtr^.portRect);
    DoSetUpNotification;
end;

iProgress: begin
    RGBBackColor(gBlueColour);
    EraseRect(gWindowPtr^.portRect);
    DoProgressIndicator;
end;

iColourPicker: begin
    gColourPickerDemo := true;
    DoColourPicker;
end;

iMultiMonitors: begin
    gMultiMonitorsDrawDemo := true;
    InvalRect(gWindowPtr^.portRect);
end;

iSystemVertBlank: begin
    gSystemVBLTaskDemo := true;
    RGBBackColor(gBlueColour);
    EraseRect(gWindowPtr^.portRect);
    DoSystemVerticalBlank;
end;

iSlotVertBlank: begin
    RGBBackColor(gBlueColour);
    EraseRect(gWindowPtr^.portRect);
    DoSlotVBLTask;
end;

otherwise begin
end;

end;
    { of case statement }

end;
end;

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

//  DoEvents

procedure DoEvents({const} var theEvent : EventRecord);
var
    partCode : SInt16;
    theWindowPtr : WindowPtr;
    charCode : SInt8;
    newSize, userData : SInt32;

begin
    case theEvent.what of

        mouseDown: begin
            partCode := FindWindow(theEvent.where, theWindowPtr);

            case partCode of

                inMenuBar: begin
                    DoMenuChoice(MenuSelect(theEvent.where));
                end;

                inContent: begin
                    if (theWindowPtr <> FrontWindow) then
                        begin
                            SelectWindow(theWindowPtr);
                        end;
                    end;

                inDrag: begin
                    DragWindow(theWindowPtr, theEvent.where, gMultiMonDragBounds);
                end;

                inGrow: begin
                    newSize := GrowWindow(theWindowPtr, theEvent.where, gMultiMonGrowBounds);
            end;
        end;
    end;
end;

```



```

begin

    gVBLCount := 5;

    gMultiMonitorsDrawDemo := false;
    gColourPickerDemo := false;

    gInColour.red := $CCCC;
    gInColour.green := $0000;
    gInColour.blue := $0000;

    gWhiteColour.red := $FFFF;
    gWhiteColour.green := $FFFF;
    gWhiteColour.blue := $FFFF;

    gBlueColour.red := $4444;
    gBlueColour.green := $4444;
    gBlueColour.blue := $9999;

    //
..... initialise managers

    DoInitManagers;

    // .....
create routine descriptor

    DoDeviceLoopDrawUPP := NewDeviceLoopDrawingProc(@DoDeviceLoopDraw);

    // ..... set
up menu bar and menus

    mainMenubarHdl := GetNewMBar(rMenubar);
    if (mainMenubarHdl = nil) then
        begin
            ExitToShell;
        end;
    SetMenuBar(mainMenubarHdl);
    DrawMenuBar;

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

    //
..... open window

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

    SetPort(gWindowPtr);
    TextSize(10);

    // ..... get process serial number of this
process

    mainErr := GetCurrentProcess(gProcessSerNum);

    // ..... get window drag and sizing limits for multiple monitors

    mainGrayRgnHdl := LMGetGrayRgn;
    gMultiMonDragBounds := mainGrayRgnHdl^^.rgnBBox;
    SetRect(gMultiMonGrowBounds, 445, 302,
        ((gMultiMonDragBounds.right) - (gMultiMonDragBounds.left)),
        ((gMultiMonDragBounds.bottom) - (gMultiMonDragBounds.top)));

    //
..... enter EventLoop

```

## Version 2.1

```
gDone := false;

while not gDone do
  begin
    if WaitNextEvent(everyEvent, mainEvent, 0, nil) then
      begin
        DoEvents(mainEvent);
      end
    else begin
      Doldle;
    end;

    UnloadSegments;
  end;

  ExitToShell; // This is here to workaround a compiler bug.
end.
{ of main program block }

{ }

{ }
// NotificationDemo.p
// }

unit NotificationDemo;

interface

uses

  { Other project files }
  MiscellanyInterface;

procedure notificationSegment;
procedure DoSetUpNotification;
procedure DoPrepareNotificationStructure;
procedure Doldle;
procedure DoOSEvent({const} var theEvent : EventRecord);
procedure DoDisplayMessageToUser;

implementation

//
.....
..... global variables

var

gNotificationStructure : NMRec;
gStartingTickCount : longint;
gNotificationDemoInvoked : boolean;
gNotificationInQueue : boolean;
gInBackground : boolean;

// notificationSegment

procedure notificationSegment;
begin
end;
  { of procedure notificationSegment }

// DoSetUpNotification

procedure DoSetUpNotification;
begin
  DoPrepareNotificationStructure;
  gNotificationDemoInvoked := true;

  gStartingTickCount := TickCount;

  RGBForeColor(gWhiteColour);
  MoveTo(10, 279);
  DrawString('Please click on the desktop now to make the Finder ');
  DrawString('the frontmost application. ');
  MoveTo(10, 292);
  DrawString('(This application will post a notification 10 seconds from now. ');
end;
```

```
    { of procedure DoSetUpNotification }

// DoPrepareNotificationStructure DoPrepareNotificationStructure
procedure DoPrepareNotificationStructure;
var
  iconSuiteHdl : Handle;
  soundHdl : Handle;
  stringHdl : StringHandle;
  ignoredErr : OSErr;

begin
  ignoredErr := GetIconSuite(iconSuiteHdl, rIconFamily, kSelectorAllSmallData);
  soundHdl := GetResource('snd ', rBarkSound);
  stringHdl := GetString(rString);

  gNotificationStructure.qType := nmType;
  gNotificationStructure.nmMark := 1;
  gNotificationStructure.nmIcon := iconSuiteHdl;
  gNotificationStructure.nmSound := soundHdl;
  gNotificationStructure.nmStr := stringHdl^;
  gNotificationStructure.nmResp := nil;
  gNotificationStructure.nmRefCon := 0;
end;
  { of procedure DoPrepareNotificationStructure }

// Doldle Doldle
procedure Doldle;
var
  frontProcessSerNum : ProcessSerialNumber;
  isSameProcess : boolean;
  ignoredErr : OSErr;

begin
  if gNotificationDemolnocked then
    begin
      if (TickCount > gStartingTickCount + 600) then
        begin
          ignoredErr := GetFrontProcess(frontProcessSerNum);
          ignoredErr := SameProcess(frontProcessSerNum, gProcessSerNum, isSameProcess);

          if not isSameProcess then
            begin
              ignoredErr := NMInstall(@gNotificationStructure);
              gNotificationDemolnocked := false;
              gNotificationInQueue := true;
            end
          else begin
            DoDisplayMessageToUser;
            gNotificationDemolnocked := false;
          end;

          EraseRect(gWindowPtr^.portRect);
        end;
      end;
    end;
  { of procedure Doldle }

// DoOSEvent DoOSEvent
procedure DoOSEvent({const} var theEvent : EventRecord);
begin
  if (BAnd(BSR(theEvent.message, 24), $000000FF) = suspendResumeMessage) then
    begin
      glnBackground := BAnd(theEvent.message, resumeFlag) = 0;
      if ((not glnBackground) and gNotificationInQueue) then
        begin
          DoDisplayMessageToUser;
        end;
    end;
  end;
  { of procedure DoOSEvent }

// DoDisplayMessageToUser DoDisplayMessageToUser
procedure DoDisplayMessageToUser;
var
  paramRec : AlertStdAlertParamRec;
  labelText : Str255;
```

## Version 2.1

```
narrativeText : Str255;
itemHit : SInt16;
ignoredErr : OSErr;

begin
if gNotificationInQueue then
  begin
  ignoredErr := NMRemove(@gNotificationStructure);
  gNotificationInQueue := false;
  end;

EraseRect(gWindowPtr^.portRect);

paramRec.movable := true;
paramRec.helpButton := false;
paramRec.filterProc := nil;
paramRec.defaultText := StringPtr(kAlertDefaultOKText);
paramRec.cancelText := nil;
paramRec.otherText := nil;
paramRec.defaultButton := kAlertStdAlertOKButton;
paramRec.cancelButton := 0;
paramRec.position := kWindowDefaultPosition;

GetIndString(labelText, rAlertStrings, indexLabel);
GetIndString(narrativeText, rAlertStrings, indexNarrative);

ignoredErr := StandardAlert(kAlertNoteAlert, @labelText, @narrativeText, @paramRec, itemHit);

ignoredErr := DisposeIconSuite(gNotificationStructure.nmlIcon, false);
ReleaseResource(gNotificationStructure.nmSound);
ReleaseResource(Handle(gNotificationStructure.nmStr));
end;
  { of procedure DoDisplayMessageToUser }

end.
  { of unit NotificationDemo }
{ }
{ }
// ProgressIndicator.p
// }

unit ProgressIndicator;

interface

uses

  { Universal Interfaces 3.2 fail to define LMGetEventQueue for 68K.
  Including the following file and MiscellanyLib fixes the problem. }
  UIFix,

  { Other project files }
  MiscellanyInterface;

procedure progressBarSegment;
procedure DoProgressIndicator;
function DoCheckForCancel(theDialogPtr : DialogPtr) : boolean;

implementation

// progressBarSegment

procedure progressBarSegment;
begin
end;
  { of procedure progressBarSegment }

// DoProgressBar

procedure DoProgressIndicator;
var
  theDialogPtr : DialogPtr;
  progressBarHdl : ControlHandle;
  statusMax, statusCurrent : SInt16;
  a, b, c : SInt16;
  soundHdl : Handle;
```

```

theRect : Rect;
redColour : RGBColor;
ignoredErr : OSErr;

begin
redColour.red := $FFFF;
redColour.green := $0000;
redColour.blue := $0000;

theDialogPtr := GetNewDialog(rDialog, nil, WindowPtr(-1));
if (theDialogPtr = nil) then
begin
ExitToShell;
end;

SetPort(theDialogPtr);
UpdateControls(theDialogPtr, GrafPtr(theDialogPtr)^.visRgn);
SetPort(gWindowPtr);

ignoredErr := GetDialogItemAsControl(theDialogPtr, iProgressIndicator, progressBarHdl);

statusMax := 3456;
statusCurrent := 0;
SetControlMaximum(progressBarHdl, statusMax);

for a := 0 to 8 do
begin
for b := 8 to 423 by 18 do
begin
if DoCheckForCancel(theDialogPtr) then
begin
soundHdl := GetResource('snd ', rBarkSound);
ignoredErr := SndPlay(nil, SndListHandle(soundHdl), false);
ReleaseResource(soundHdl);
DisposeDialog(theDialogPtr);


EraseRect(gWindowPtr^.portRect);
MoveTo(10, 292);
RGBForeColor(gWhiteColour);
DrawString('Operation cancelled at user request');

Exit(DoProgressIndicator);
end;

for c := 8 to 286 by 18 do
begin
SetRect(theRect, b+a, c+a, b+17-a, c+17-a);
if (a < 3) then
begin
RGBForeColor(gWhiteColour);
end
else if ((a > 2) and (a < 6)) then
begin
RGBForeColor(redColour);
end
else if (a > 5) then
begin
RGBForeColor(gWhiteColour);
end;
FrameRect(theRect);

SetControlValue(progressBarHdl, statusCurrent);
statusCurrent := statusCurrent + 1;
end;
end;
end;

DisposeDialog(theDialogPtr);
EraseRect(gWindowPtr^.portRect);
MoveTo(10, 292);
RGBForeColor(gWhiteColour);
DrawString('Operation completed');
end;
{ of procedure DoProgressIndicator }

//  DoCheckForCancel

function DoCheckForCancel(theDialogPtr : DialogPtr) : boolean;
var

```

## Version 2.1

```
oldPort : GrafPtr;
foundCommandPeriod : boolean;
eventQHdrPtr : QHdrPtr;
eventQEIPtr : EvQEIPtr;
theType : SInt16;
theHandle : Handle;
theRect : Rect;
finalTicks : UInt32;
keyCode : SInt32;
commandKeyDown : SInt32;

begin
  GetPort(oldPort);
  SetPort(theDialogPtr);

  foundCommandPeriod := false;

  eventQHdrPtr := LMGetEventQueueGlue;
  eventQEIPtr := EvQEIPtr(eventQHdrPtr^.qHead);

  while ((eventQEIPtr <> nil) and not foundCommandPeriod) do
    begin
      if (eventQEIPtr^.evtQWhat = mouseDown) then
        begin
          GlobalToLocal(eventQEIPtr^.evtQWhere);
          GetDialogItem(theDialogPtr, iStopButton, theType, theHandle, theRect);

          if PtInRect(eventQEIPtr^.evtQWhere, theRect) then
            begin
              HiliteControl(ControlHandle(theHandle), kControlButtonPart);
              Delay(8, finalTicks);
              HiliteControl(ControlHandle(theHandle), 0);
              foundCommandPeriod := true;
              FlushEvents(everyEvent, 0);
            end;
          end;
        else if (eventQEIPtr^.evtQWhat = keyDown) then
          begin
            keyCode := BAnd(eventQEIPtr^.evtQMessage, keyCodeMask);
            keyCode := BSR(keyCode, 8);

            commandKeyDown := BAnd(eventQEIPtr^.evtQModifiers, cmdKey);

            if (commandKeyDown <> 0) then
              begin
                if (keyCode = $2F) then
                  begin
                    foundCommandPeriod := true;
                  end;
                end;
              end;
            end;

            if not foundCommandPeriod then
              begin
                eventQEIPtr := EvQEIPtr(eventQEIPtr^.qLink);
              end;
            end;
          end;
        SetPort(oldPort);

        DoCheckForCancel := foundCommandPeriod;
      end;
      { of function DoCheckForCancel }
    end.
    { of unit ProgressIndicator }

    { }
    { }
    // ColourPicker.p
    // { }

  unit ColourPicker;

  interface

  uses
```











```

end.
  { of unit MultiMonitor }

{ }

{ }
// VerticalBlank.p
// }

unit VerticalBlank;

interface

uses

  { Other project files }
  MiscellanyInterface;

procedure DoSystemVerticalBlank;
function DoInstallSystemVBLTask : OSErr;

{$ifc TARGET_CPU_68K}
procedure theSystemVBLTask;
{$elsec}
procedure theSystemVBLTask(theVBLStructurePtr : VBLStructurePtr);
{$endc}
procedure DoStopSystemVBLTask;
procedure DoSlotVBLTask;
function DoInstallSlotVBLTask : OSErr;

{$ifc TARGET_CPU_68K}
procedure theSlotVBLTask;
{$elsec}
procedure theSlotVBLTask(theVBLStructurePtr : VBLStructurePtr);
{$endc}
procedure DoStopSlotVBLTask;

var

gVBLCount : SInt16;

implementation

// ..... in-
line glue for GetVBLRec

{$ifc TARGET_CPU_68K}
function GetVBLRecGlue : SInt32;
  inline $2E88;
{$endc}

//
..... global variables

var

gSystemVBLTaskUPP : VBLUPP;
gSlotVBLTaskUPP : VBLUPP;
gSystemVBLStructure : VBLStructure;
gSlotVBLStructure : VBLStructure;
gMainSlotNumber : SInt16;

// ..... DoSystemVerticalBlank

procedure DoSystemVerticalBlank;
  var
    osErr : OSErr;

  begin
    gSystemVBLTaskUPP := NewVBLProc(@theSystemVBLTask);

    RGBForeColor(gWhiteColour);

```









- Choose the Colour Picker item and make colour choices using the various available modes. Note that, when the Colour Picker is dismissed by clicking the OK button, the requested RGB colour values for the chosen colour are displayed in hexadecimal, together with a rectangle in that colour, in the Miscellany window.
- Choose the Multiple Monitors Draw item, noting that the drawing of the simple demonstration image is optimised as follows:
  - On a monitor set to display 256 or more colours, the image is drawn in three distinct colours. The luminance of the three colours is identical, meaning that, if these colours are drawn on a grayscale screen, they will all appear in the same shade of gray.
  - On a monitor set to display 256 shades of gray, the image is drawn in three distinct shades of gray.
- Choose the System-Based VBL Task item, noting the cursor animation sequenced by a system-based VBL task.
- Choose the Slot-Based VBL Task item, bearing in mind that the successive re-drawing of the picture is delayed until the monitor is in the vertical blank period.

In addition, if the user's system has more than one monitor, the user should zoom the window in and out when the window is on the main monitor, when it has been dragged to the second monitor, and when it has been dragged to a position where it is partially displayed on both monitors, noting the standard state, and the monitor zoomed to, in each case.

## ***MiscellanyInterface.p***

---

### ***type definitions***

---

The VBLStructure data type, which can be regarded as an expanded VBL task structure, will be used by both the system-based and slot-based VBL tasks. The first field is a VBL task structure. The second field will be used to save the A5 register (680x0 code only). The third field will be used by the slot-based VBL task only.

### ***Global Variables***

---

gProcessSerNum will be assigned the process serial number of the Miscellany application. gMultiMonDragBounds will be passed in the bounds parameter of the DragWindow function. gMultiMonGrowBounds will be passed in the bBox parameter of the GrowWindow function.

## ***Miscellany.p***

---

### ***Global Variables***

---

DoDeviceLoopDrawUPP will be assigned a universal procedure pointer to the application-defined image-optimising drawing routine called by DeviceLoop.

### ***main program block***

---

The call to NewDeviceLoopDrawingProc creates a routine descriptor for the image-optimising drawing routine DoDeviceLoopDraw.

GetCurrentProcess gets the process serial number of this process, and enters the main event loop.

The next block defines two rectangles. gMultiMonDragBounds is set to equate to the current desktop region, which potentially crosses multiple devices and consists of everything below the menu bar. This establishes the limits within which the user will be able to drag the window. gMultiMonGrowBounds establishes the minimum and maximum heights and widths to apply to window resizing.

Within the event loop, note that:

- The call to DIdle is relevant to the notification demonstration only.
- The application-defined routine UnloadSegments is called at the bottom of the event loop after the event received by WaitNextEvent has been handled to completion.

### ***DoEvents***

---

Note that, within the mouseDown case, gMultiMonDragBounds is passed in the bounds parameter of DragWindow, gMultiMonGrowBounds is passed in the bBox parameter of GrowWindow, and the application-defined window-zooming routine DoZoomWindowMultiMonitors is called at the inZoomIn/inZoomOut case.

Within the updateEvt case, if the Multiple Monitors Draw item in the Demonstration menu has been chosen (gMultiMonitorsDrawDemo is true), a call is made to DeviceLoop and the universal procedure pointer to the application-defined drawing routine DoDeviceLoopDraw is passed as the second parameter.

## ***DoMenuChoice***

---

At the fifth and sixth lines, if the system-based VBL task is in operation, the application-defined routine DoStopSystemVBLTask is called to remove the VBL task.

When the Multiple Monitors Draw item in the Demonstration menu is chosen, the window's port rectangle is invalidated so as to force an update event and consequential call to DeviceLoop.

## ***UnloadSegments***

---

UnloadSegments unlocks, and marks as purgeable, the specified 68K code segments, that is, the segment in which a stub ("do nothing" function) resides. Note that a stub has not been included in the segment containing the VBL task demonstration code, and that that segment is preloaded and locked. This is because a VBL task's code must be locked into physical memory if virtual memory is in operation.

## ***NotificationDemo.p***

---

### ***notificationSegment***

---

notificationSegment is the stub, or "do nothing" function, called by UnloadSegments at the bottom of the main event loop.

### ***DoSetUpNotification***

---

DoSetUpNotification is called when the user chooses Notification from the Demonstration menu.

The first line calls an application-defined routine which fills in the relevant fields of a notification structure. The next line assigns true to a global variable which records that the Notification item has been chosen by the user.

The next line saves the system tick count at the time that the user chose the Notification item. This value is used later to determine when 10 seconds have elapsed following the execution of this line.

### ***DoPrepareNotificationStructure***

---

DoPrepareNotificationStructure fills in the relevant fields of the notification structure.

First, however, GetIconSuite creates an icon family based on the specified resource ID and the third parameter, which limits the family to 'ics#', 'ics4' and 'ics8' icons. The GetIconSuite call returns the handle to the suite in its first parameter. The call to GetResource loads the specified 'snd ' resource. GetString loads the specified 'STR ' resource.

The first line of the main block specifies the type of operating system queue. The next line specifies that the ♦ mark is to appear next to the application's name in the Application menu. The next three lines assign the icon suite, sound and string handles previously obtained. The next line specifies that no response function is required to be executed when the notification is posted.

## ***Doldle***

---

Doldle is called from the main event loop when a null event is received. Recall that the canBackground flag is set in the application's 'SIZE' resource, meaning that the application will receive null events when it is in the background.

If the user has not just chosen the Notification item in the Demonstration menu (gNotificationDemoInvoked is false), Doldle simply returns immediately.

If, however, that item has just been chosen, and if 10 seconds (600 ticks) have elapsed since that choice was made, the following occurs:

- The calls to GetFrontProcess and SameProcess determine whether the current foreground process is Miscellany. If it is not, the notification request is installed in the notification queue by NMInstall and a global variable is set to indicate that a request has been placed in the queue by Miscellany. Also, gNotificationDemoInvoked is set to false so as to ensure that the main if block only executes once after the Notification item is chosen.
- If, however, the current foreground process is Miscellany, the application-defined routine DoDisplayMessageToUser is called to present the required message to the user, via an alert box, in the normal way. Once again gNotificationDemoInvoked is reset to false so as to ensure that the main if block only executes once after the Notification item is chosen.

## ***DoOSEvent***

---

DoOSEvent handles operating system events.

If the event is a resume event (that is, Miscellany is coming to the foreground) and if the notification request is still in the notification queue (gNotificationInQueue is true), the application-defined routine DoDisplayMessageToUser is called to remove the notification request from the queue and have Miscellany convey the required message to the user via an alert box.

## ***DoDisplayMessageToUser***

---

DoDisplayMessageToUser is called by DoOSEvent and Doldle in the circumstances previously described.

If a Miscellany notification request is in the queue, NMRemove removes it from the queue and gNotificationInQueue is set to false to reflect this condition. (Recall that, if the nmResp field of the notification structure is not assigned -1, the application itself must remove the queue element from the queue.)

Regardless of whether there was a notification in the queue or not, Miscellany then presents its alert. When the alert is dismissed, the notification's icon suite, sound and string resources are released/disposed of.

## ***ProgressIndicator.p***

---

### ***DoProgressIndicator***

---

DoProgressIndicator is called when the user chooses Determinate Progress Indicator from the Demonstration menu.

GetNewDialog creates a modal dialog box. The call to UpdateControls draws the dialog box's controls. The call to GetDialogItemAsControl retrieves the handle to the dialog's progress indicator control. SetControlMaximum sets the control's maximum value to equate to the number of steps in a simulated time-consuming task.

The main for loop performs the simulated time-consuming task, represented to the user by the drawing of a large number of coloured rectangles in the window. The task involves 3456 calls to FrameRect.

Within the middle for loop, the application-defined routine DoCheckForCancel is called to check whether the user has pressed the Command-period key combination or clicked the dialog's Stop button. If so, a 'snd ' resource is loaded, played, and released, the dialog is disposed of, an advisory message is drawn in the window, and the function returns.

Within the inner for loop, the rectangles are drawn. Each time round this inner loop, a progress indicator control's value is incremented.

When the outer loop exits (that is, when the Command-period key combination is not pressed or the Stop button is not clicked before the simulated time-consuming task completes), the dialog is disposed of.

### ***DoCheckForCancel***

---

DoCheckForCancel scans the event queue for Command-period keyboard events and mouse-down events.

The first line sets a variable so as to begin by assuming that such an event is not in the queue.

LMGetEventQueue gets a pointer to the event queue header. The next line gets a pointer to first queue element. The while loop scans the whole of the event queue, exiting only when a Command-period key event is found in the queue, a mouse-down is found in the queue and that mouse-down was within the Stop push button's rectangle, or the entire queue has been scanned.

Inside the loop, if a mouse-down event is found, GetDialogItem is called to get the Stop push button's rectangle and PtInRect is called to determine whether the mouse-down was within that rectangle. If it was, the push button is highlighted for 8 ticks and the variable foundCommandPeriod is set to true, causing the loop to exit and DoCheckForCancel to return true. The call to FlushEvents ensures that, when DoProgressIndicator disposes of the dialog, the window will be activated.

Also inside the loop, if a key-down event is found, the first two lines in the else if block get the key code and the third line checks whether the Command key was down. If the Command key was down, and if the period key was the key pressed (keyCode \$2f), the variable foundCommandPeriod is set to true, causing the loop to exit and DoCheckForCancel to return true.

If neither a mouse-down in the Stop push button nor a Command-period keyboard event was found, the last two lines in the while loop call up the next queue for examination.

The last line returns the result of the search.

## ***ColourPicker.p***

---

### ***DoColourPicker***

---

DoColourPicker is called when the user chooses Colour Picker from the Demonstration menu.

The first block erases the window's content area and paints a rectangle in the colour which will be passed in GetColor's inColor parameter.

The next line assigns 0 to the fields of the Point variable to be passed in GetColor's where parameter. ((0,0) will cause the Colour Picker dialog box to be centred on the main screen.)

The call to GetColor displays the Colour Picker's dialog box. GetColor retains control until the user clicks either the OK button or the Cancel button, at which time the port rectangle is invalidated, causing the function DoDrawColourPickerChoice to be called.

## **DoDrawColourPickerChoice**

If the user clicked the OK button, a filled rectangle is painted in the window in the colour returned in GetColor's outColor parameter, and the values representing the red, green, and blue components of this colour are displayed at the top of the window in hexadecimal. Note that the application-defined routine DoDecimalToHexadecimal is called to convert the decimal (UInt32) values in the fields of the RGBColor variable outColor to hexadecimal.

If the user clicks the Cancel button, a filled rectangle is painted in the window in the colour passed in GetColor's inColor parameter.

## **DoDecimalToHexadecimal**

DoDecimalToHexadecimal converts a UInt16 value to a hexadecimal string.

## **MultiMonitor.p**

### **DoDeviceLoopDraw**

DoDeviceLoopDraw is the image-optimising drawing routine the universal procedure pointer to which is passed in the second parameter in the DeviceLoop call in the DoEvents routine. (Recall that the DeviceLoop call is made whenever the Multiple Monitors Draw item in the Demonstration menu has been selected and an update event is received.) DeviceLoop scans all active video devices, calling DoDeviceLoopDraw whenever it encounters a device which intersects the drawing region, and passing certain information to DoDeviceLoopDraw.

The second line casts the SInt32 value received in the userData parameter to a WindowPtr. The window's content area is then erased.

If an examination of the device's attributes, as received in the deviceFlags formal parameter, reveals that the device is a colour device, three rectangles are painted in the window in three different colours. (The luminance value of these colours is the same, meaning that the rectangles would all be the same shade of gray if they were drawn on a monochrome (grayscale) device.)

If the examination of the device's attributes reveals that the device is a monochrome device, the rectangles are painted in three distinct shades of gray.

### **DoZoomWindowMultiMonitors**

DoZoomWindowMultiMonitors is called when the user clicks in the window's zoom box.

The first two lines save and set the current graphics port. The third line erases the window's port rectangle prior to the zoom so as to avoid flicker.

The main if block executes only if the direction of the zoom is "out" to the standard state. The purpose of this block of code is to determine the standard state rectangle and, in a multi-monitors environment, the monitor on which the zoomed window is to be displayed.

The first block inside the if block converts the window's port rectangle to global coordinates and gets the height of the window's title bar. The next line establishes a rectangle equal to the window's port rectangle, plus the window's title bar, in global coordinates.

LMGetDeviceList gets a handle to the first gDevice structure in the device list and the variable greatestArea is assigned 0.

The while loop then walks the device list. For each active video device, the associated gDevice structure's gdRect field is compared to the window's rectangle by a call to SectRect. If the two rectangles intersect, the coordinates of the intersection are assigned to the intersectRect variable, otherwise an empty rectangle ((0,0)(0,0)) is assigned to intersectRect. The area of the intersection rectangle is calculated and stored in the variable intersectArea. If the new value in intersectArea is greater than that calculated during any previous pass through the loop, the variable zoomDeviceHdl is assigned the GDHandle of the device currently being examined.

GetNextDevice gets the handle to the next device in the device list. The while loop exits when this call returns nil. When the loop exits, the contents of the variable zoomDeviceHdl represents the video device on which the window should be zoomed to the standard state, that is, the device on which the largest area of the window currently appears.

If the call to LMGetMainDevice reveals that this device is the main device, the height of the menu bar is added to the value in the variable which holds the window's title bar height.

The next two lines determine the width of the window frame and the following four lines establish the standard state rectangle. This latter is three pixels inside the rectangle contained in the gdRect field of the device's gDevice structure, but with the top adjusted to account for the height of the title bar (and the menu bar if the device is the main device) and the sides and bottom adjusted for the width of the window frame. The last two lines in the main if block then assign this rectangle to the stdState field of the window's state data structure.

Below the main if block, ZoomWindow is called to zoom the window in the appropriate direction, following which an application-defined routine is called to redraw the window contents as appropriate. Finally, the saved graphics port is restored.

## ***DoRedoWindowContent***

---

DoRedoWindowContent is called by DoZoomWindowMultiMonitors to redraw the content region of a newly-zoomed window. In this demonstration the window's content area is simply invalidated, forcing an update event.

## ***VerticalBlank.p***

---

### ***DoSystemVerticalBlank***

---

DoSystemVerticalBlank is called when the user chooses System-Based VBL Task from the Demonstration menu.

In this demonstration, the VBL task is used to increment the frames of an animated cursor so as to provide visual evidence to the user of the execution of the task. It should be noted, however, that animating a cursor using a VBL task, as opposed to the methods described at Chapter 13 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons, is not recommended. The Chapter 13 methods will ensure the cessation of the animation if the application "locks up" after the animation is initiated, 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.

The first line creates a routine descriptor for the system-based VBL task.

The 'acur' resource ID for the animated cursor is 0. Accordingly, InitCursorCtl is called with NULL passed in the newCursors parameter to load that resource.

The application-defined routine DoInstallSystemVBLTask is then called to install the system-based VBL task. If this call is not successful, the routine descriptor is disposed of and the function returns.

### ***DoInstallSystemVBLTask***

---

DoInstallSystemVBLTask installs the system-based VBL task.

The vblAddr field of the VBL task structure is assigned the universal procedure pointer to the application-defined routine theSystemVBLTask, and the vblCount field is assigned the value 5. This means that the VBL task will execute every five interrupts.

Applicable to 680x0 compilation only, the sixth line gets the current value in the system global variable CurrentA5 (that is, the application's A5) and assigns it to the relevant field of the expanded VBL task structure.

VInstall installs the specified VBL task into the system-based queue.

### ***theSystemVBLTask***

---

theSystemVBLTask is the VBL task, which will be executed every 5 VBL interrupts. Much of the code pertains to 680x0 compilation only, reflecting the fact that, in the 680x0 environment:

- The current A5 must be saved, and the application's A5 must be made the current A5, if the VBL task needs to access an application global variable (as it does in this case).
- The in-line function GetVBLRec needs to be called to get the address of the VBL task structure.

Applicable to 680x0 compilation only, GetVBLRec is called to get the address of the VBL task structure (which is the same as the address of the expanded VBL structure). Also, SetA5 is called to save the current A5 and, at the same time, set the application's A5 world (previously saved to the thisApplicationsA5 field of the expanded VBL task structure) as that to be used for the duration of the VBL task's execution. The task can now access the application's global variables.

The actual task (the call to SpinCursor) increments the animated cursor frame. When the task executes, the value in the vblCount field of the VBL task record will be set to 0. Accordingly, the next line resets this field to a value higher than 0 (in this case, 5), otherwise the task will never execute again. The global variable gVBLCount is used for this purpose.

Applicable to 680x0 compilation only, the last line restores the saved A5 world.

### ***DoStopSystemVBLTask***

---

DoStopSystemVBLTask is called when the user chooses any menu item.

VRemove removes the VBL task from the queue, DisposeRoutineDescriptor disposes of the specified routine descriptor, and SetCursor sets the standard arrow cursor shape.

### ***DoSlotVBLTask***

---

DoSlotVBLTask is called when the user chooses Slot-Based VBL Task from the Demonstration menu. In this demonstration, the slot-based VBL task is used to delay the drawing of a picture at a new location until the monitor enters the vertical blank period.

The first line creates a routine descriptor for the slot-based VBL task.

The next block gets the slot number of the main graphics device. This process involves getting a handle to the startup gDevice structure, extracting from that structure the device driver's reference number, getting a handle to the DCtlEntry structure, and then getting the slot number.

DoInstallSlotVBLTask is then called to install the slot-based VBL task. If this call is not successful, the function returns.

The call to GetPicture loads the specified 'PICT' resource, following which the picFrame field in the resource is copied to a local variable. Within the outer while loop, this rectangle is continually offset between successive calls to DrawPicture, causing the picture to appear to move around the window. The inner (empty) while loop continues to cycle until the inBlankPeriod field of the expanded VBL task structure is set to true by the VBL task, at which time the Picture is drawn and the inBlankPeriod field is set back to false.

When the user clicks the mouse, the outer while loop exits and doStopSlotVBLTask is called to remove the slot-based VBL task.

## **DoInstallSlotVBLTask**

DoInstallSlotVBLTask installs the slot-based VBL task.

The first four lines 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.

The fifth line sets a field in the expanded VBL task structure to false. This field will be set to true by the slot-based VBL task. Note that there is no need to save the application's A5 in this case because the slot-based VBL task does not need to access an application global variable.

SlotVInstall attempts to install the task in the slot-based queue. The success, or otherwise, of the attempted installation is returned to the calling function.

## **theSlotVBLTask**

theSlotVBLTask is the slot-based VBL task itself. It is similar to theSystemVBLTask except that no measures are required to facilitate access to the application's global variables.

When the task executes, it sets to true the flag in the expanded VBL task structure which indicates that the vertical blanking period has just been entered. 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, the last line sets the vblCount field of the VBL task record back to 1.

## **DoStopSlotVBLTask**

DoStopSlotVBLTask is called when the user clicks the mouse button. It removes the slot-based VBL task from the relevant queue and disposes of the routine descriptor.

Recall that, at the Demonstration Program Controls2 Comments section at Chapter 7 — Introduction to Controls, mention was made of horizontal "tearing" of the picture in the lower section of the window when it was being scrolled using the scroll arrows, and that this would be addressed in the Demonstration Program Comments section of this chapter.

This "tearing" can be eliminated by delaying the drawing of the picture until the vertical blanking period is entered, using the same approach as is used in the slot-based VBL task demonstration, above. Simply install a slot-based VBL task using the relevant source code in the Miscellany demonstration and change the bottom of the actionFuncLive function in Controls2.p to the following:

```
DoMoveScrollBar(controlHdl,scrollDistance);
end;

SetOrigin(GetControlValue(docStrucHdl^^.scrollbarLiveHdl, 0);

{ Set this flag to true if VBL task is not successfully installed.}
if not gVBLInstallFail then
begin
  { Wait for vertical blanking period. }
  while not gSlotVBLStructure.inVBlankPeriod) do
  begin
    end;
  DrawPicture(gPictHandleLive, gPictRectLive);
  gSlotVBLStructure.inVBlankPeriod := false;
end
else begin
  DrawPicture(gPictHandleLive, gPictRectLive);
end;
```

```
SetOrigin(0,0);  
end;
```