12 Version 1.2 (Frozen)

OFFSCREEN GRAPHICS WORLDS, PICTURES, CURSORS, AND ICONS

Includes Demonstration Program GWorldPicCursIconPascal

Offscreen Graphics Worlds

Introduction

An **offscreen graphics world** may be regarded as a virtual screen on which your application can draw a complex image without the user seeing the various steps your application takes before completing the image. The image in an offscreen graphics world is drawn into a part of memory not used by the video device. It therefore remains hidden from the user.

One of the key advantages of using an offscreen graphics ports is that it allows you to improve on-screen drawing speed and visual smoothness. For example, suppose your application draws multiple graphics objects in a window and then needs to update part of that window. If your image is very complex, your application can copy it from an offscreen graphics world to the screen faster than it can repeat all of the steps necessary to draw the image on-screen. At the same time, the inelegant visual effects associated with the time-consuming drawing a large number of separate objects are avoided.

Another typical use for an offscreen graphics port is demonstrated at Chapter 19 — Custom Control Definition Functions and VBL Tasks. In the demonstration program at that chapter, the images of two parts of a slider control (the track and the "thumb") are assembled into a composite image in an offscreen graphics port before being copied to the front window's graphics port. This happens repeatedly while the slider is being moved. The continual erasing and redrawing of this composite animated image is thus not visible to the user, who sees only the smooth, flicker-free final result.

Creating an Offscreen Graphics World

You create an offscreen graphics world with the `NewGWorld` function. `NewGWorld` creates a new offscreen graphics port, a new offscreen pixel map, and (on computers which support Color QuickDraw) either a new `GDevice` record or a link to an existing one. `NewGWorld` returns a pointer of type `GWorldPtr` which points to a colour graphics port:

```
typedef CGrafPtr GWorldPtr;
```

When you use `NewGWorld`, you can specify a pixel depth, a boundary rectangle (which also becomes the port rectangle), a colour table, a `GDevice` record, and option flags for memory allocation. Passing 0 as the pixel depth, the window's port rectangle as the offscreen world's boundary rectangle, `NULL` for both the colour table and the `GDevice` record and 0 as the options flags:

- Provides your application with the default behaviour of NewGWorld.
- Supports computers running only basic QuickDraw.

- Allows QuickDraw to optimise the `CopyBits`, `CopyMask`, and `CopyDeepMask` routines used to copy the image into the window's port rectangle.

Setting the Graphics Port for an Offscreen Graphics World

Before drawing into the offscreen graphics port, you should save the graphics port for the front window by calling `GetGWorld`, which saves the current graphics port *and* its `GDevice` record. The offscreen graphics world should then be made the current port by a call to `SetGWorld`. After drawing into the offscreen graphics world, you use `SetGWorld` to restore the active window as the current graphics port.

Note that `SetGWorld` sets the port specified in its `port` parameter as the current port and the device specified in its `gdh` parameter as the current device.

`GetGWorld` and `SetGWorld` save and restore both basic and colour graphics ports.

Preparing to Draw Into an Offscreen Graphics World

After setting the offscreen graphics world as the current port, you should use the `GetGWorldPixMap` function to get a handle to the offscreen pixel map. This is required as the parameter in a call to the `LockPixels` function, which you must call before drawing to, or copying from, an offscreen graphics world.

`LockPixels` prevents the base address of an offscreen pixel image from being moved while you draw into it or copy from it. If the base address for an offscreen pixel image has not been purged by the Memory Manager, or if its base address is not purgeable, `LockPixels` returns `true`. If `LockPixels` returns `false`, your application should either call the `UpdateGWorld` function to reallocate the offscreen pixel image and then reconstruct it, or draw directly into an onscreen graphics port.

`GetGWorldPixMap` and Basic QuickDraw

Note that on a system running only basic QuickDraw, `GetGWorldPixMap` returns the handle to a 1-bit pixel map that your application can supply as a parameter to the other routines related to offscreen graphics worlds described in this section. On a basic QuickDraw system, however, your application should not supply this handle to Color QuickDraw routines.

Copying an Offscreen Image into a Window

After drawing the image in the offscreen graphics world, your application should call `SetGWorld` to restore the active window as the current graphics port.

The image is copied from the offscreen graphics world into the window using `CopyBits` (or, if masking is required, `CopyMask` or `CopyDeepMask`). Specify the offscreen graphics world as the source image for `CopyBits` and specify the window as its destination. Note that `CopyBits` expects its source and destination parameters to be pointers to bitmaps. Accordingly, you must coerce the offscreen graphic's world's `GWorldPtr` data type to a data structure of type `GrafPtr`. Similarly, whenever a colour graphics port is your destination, you must coerce the window's `CGrafPtr` data type to data type `GrafPtr`.[1]

As long as you are drawing into an offscreen graphics world or copying an image from it, you must leave its pixel image locked. When you are finished drawing into, and copying from, an offscreen graphics world, call `UnlockPixels`. (Calling `UnlockPixels` will assist in preventing heap fragmentation.)

[1]As a related matter, note that the `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle, whereas the `baseAddr` field for an onscreen pixel map contains a pointer. You must use the `GetPixBaseAddr` function to obtain a pointer to the `PixMap` record for an offscreen graphics world.

Updating an Offscreen Graphics World

If, for example, you are using an offscreen graphics world to support the window updating process, you can use `UpdateGWorld` to carry certain changes affecting the window (for example, resizing, changes to the pixel depth of the screen, or modifications to the colour table) through to the offscreen graphics world. `UpdateGWorld` allows you to change the pixel depth, boundary rectangle, or colour table for an existing offscreen graphics world without recreating it and redrawing its contents.

Disposing of an Offscreen Graphics World

Call `DisposeGWorld` when your application no longer needs the offscreen graphics world.

Pictures

Introduction

QuickDraw provides a simple set of routines for recording a collection of its drawing commands and then playing the recording back later. Such a collection of drawing commands, as well as the resulting image, is called a **picture.** Pictures provide a common medium for the sharing of image data. They make it easier for your application to draw complex images defined in other applications, and vice versa.

Pictures can be created in colour or black-and-white. Macintoshes using only basic QuickDraw use black-and-white to display pictures created in colour.

When you use `OpenCPicture` or `OpenPicture`[2] to begin defining a picture, QuickDraw collects your subsequent drawing commands in a data structure of type `Picture`. By using `DrawPicture`, you can draw onscreen the picture defined by the instructions stored in the `Picture` record.

Picture Formats

During QuickDraw's evolution, three different formats have evolved for the data contained in a `Picture` record:

- The original format, the **version 1 format**, which is created by the `OpenPicture` function on machines without Color QuickDraw or whenever the current graphics port is a basic graphics port. Pictures created in this format support only black-and-white drawing operations at 72 dpi (dots per inch).

- The **version 2 format**, which is created by the `OpenPicture` function on machines with Color QuickDraw when the current graphics port is a colour graphics port. Pictures created in this format support colour drawing operations at 72 dpi.

- The **extended version 2 format**, which is created by the `OpenCPicture` function on all Macintosh computers running System 7, including those supporting only basic QuickDraw. This format permits your application to specify resolutions for pictures in colour or black-and-white.

Generally, your application should create pictures in the extended version 2 format.

[2]The `OpenPicture` function, which is similar to the `OpenCPicture` function, was created for earlier versions of the system software. Because of its support for higher resolutions, you should use `OpenCPicture` rather than `OpenPicture` to create a picture.

The `Picture` Record

The `Picture` record is as follows:

```
type
Picture = record
  picSize:   integer;  {For a version 1 picture: its size.}
  picFrame:  Rect;     {Bounding rectangle for the picture.}
  …                    {Picture definition (variable length).}
  end;

PicPtr = ^Picture;
PicHandle = ^PicPtr;
```

Field Descriptions

`picSize` The information in this field is useful only for version 1 pictures, which cannot exceed 32 KB in size. Version 2 and extended version 2 pictures can be larger than 32 KB. To maintain compatibility with the version 1 picture format, the `picSize` field was not changed for version 2 or extended version 2 picture formats.

(You should use the Memory Manager function `GetHandleSize` to determine the size of a picture in memory, the File Manager function `PBGetFInfo` to determine the size of a picture in a file of type `'PICT'`, and the Resource Manager function `MaxSizeResource` to determine the size of a picture in a resource of type `'PICT'`.)

`picFrame` Contains the bounding rectangle for the picture. `DrawPicture` uses this rectangle to scale the picture when you draw into a differently sized rectangle.

... Compact drawing commands and picture comments constitute the rest of the record, which is of variable length.

Opcodes: Drawing Commands and Picture Comments

The variable length field in a `Picture` record contains data in the form of **opcodes**, which are values that `DrawPicture` uses to determine what objects to draw or what mode to change for subsequent drawing.

In addition to **compact drawing commands**, opcodes can also specify **picture comments**, which are created using `PicComment`. A picture comment contains data or commands for special processing by output devices, such as PostScript printers. If your application requires capability beyond that provided by QuickDraw drawing routines, `PicComment` allows your application to pass data or commands direct to the output device.

You typically use QuickDraw commands when drawing to the screen and picture comments to include special drawing commands for printers only.

Colour Pictures in Basic Graphics Ports

You can use Color QuickDraw drawing commands to create a colour picture on a computer supporting Color QuickDraw. If the user were to cut the picture and paste it into an application that draws into a basic graphics port, the picture would lose some detail, but should be sufficient for most purposes.

`'PICT'` Files, `'PICT'` Resources, and `'PICT'` Scrap Format

QuickDraw provides routines for creating and drawing pictures. File Manager and Resource Manager routines are used to read pictures from, and write pictures to, a disk. Scrap Manager routines are used to read pictures from, and write pictures to, the scrap[3].

[3] See Chapter 16 — Scrap.

A picture can be stored in the data fork of a file of type 'PICT'. A picture can also be stored as a 'PICT' resource in the resource fork of any file type. Note that the data fork of a 'PICT' file contains a 512-byte header that applications can use for their own purposes.

For each application, the Scrap Manager maintains a storage area to hold the last data cut or copied by the user. The area that is available to your application for this purpose is called the **scrap**. All applications that support copy-and-paste operations read data from, and write data to, the scrap. The 'PICT' scrap format is one of two standard scrap formats. (The other is 'TEXT'.)

The Picture Utilities

In addition to the QuickDraw routines for creating and drawing pictures, system software provides a group of routines called the **Picture Utilities** for examining the content of pictures. You typically use the Picture Utilities before displaying a picture.

The Picture utilities allow you to gather colour, comment, font, resolution, and other information about pictures. You might use the Picture Utilities, for example, to determine the 256 most-used colours in a picture, and then use the Palette Manager to make those colours available for the window in which the application needs to draw the picture.

The Picture Utilities also collect information from black-and-white pictures and bitmaps. They are supported in System 7 even by computers running only basic QuickDraw. However, when collecting colour information on a computer running only basic QuickDraw, the Picture Utilities return NIL instead of handles to Palette and ColorTable records.

Creating Pictures

Use the OpenCPicture function to begin defining a picture. OpenCPicture collects your subsequent drawing commands in a new Picture record. To complete the collection of drawing (and picture comment) commands which define your picture, call ClosePicture.

You pass information to OpenCPicture in the form of an OpenCPicParams record:

```
type
OpenCPicParams = record
  srcRect:    Rect;     {Optimal bounding rectangle.}
  hRes:       Fixed;    {Best horizontal resolution.}
  vRes:       Fixed;    {Best vertical resolution.}
  version:    integer;  {Set to -2.}
  reserved1:  integer;  {(Reserved.  Set to 0.)}
  reserved2:  longint;  {(Reserved.  Set to 0.)}
  end;
```

This record provides a simple mechanism for specifying resolutions when creating images. For example, applications that create pictures from scanned images can specify resolutions higher than 72 dpi.

Clipping Region. You should always use ClipRect to specify a clipping region appropriate to your picture before calling OpenCPicture. If you do not specify a clipping region, OpenCPicture uses the clipping region specified in the current graphics port. If this region is very large (as it is when the graphics port is initialised, being set to the size of the coordinate plane by that initialisation) and you scale the picture when drawing it, the clipping region can become invalid when DrawPicture scales the clipping region, in which case your picture will not be drawn. On the other hand, if the graphics port specifies a small clipping region, part of your drawing may be clipped when you draw it. Setting the clipping region equal to the port rectangle of the current graphics port always sets a valid clipping region.

When the picture has been drawn with QuickDraw drawing commands, a call to ClosePicture concludes the picture definition.

Opening and Drawing Pictures

Using File Manager routines, your application can retrieve pictures saved in `'PICT'` files.[4] Using the `GetPicture` function, your application can retrieve pictures saved in the resource forks of other file types. Using the Scrap Manager function `GetScrap`, your application can retrieve pictures stored in the scrap.

When the picture is retrieved, `DrawPicture` is called to draw the picture. The second parameter taken by `DrawPicture` is the destination rectangle. This rectangle should be specified in coordinates local to the current graphics port. `DrawPicture` shrinks or stretches the picture as necessary to make it fit into this rectangle.

When you are finished using a picture stored as a `'PICT'` resource, you should use the resource Manager routine `ReleaseResource` to release its memory.

Saving Pictures

After creating or changing pictures, your application should allow the user to save them. To save a picture in a `'PICT'` file, you should use the appropriate File Manager routines.[4] (Remember that the first 512 bytes of a `'PICT'` file are reserved for your application's own purposes.) To save pictures in a `'PICT'` resource, you should use the appropriate Resource Manager routines. To place a picture in the Scrap (for example, to respond to the user choosing the **Copy** command to copy a picture to the clipboard), you should use the Scrap Manager function `PutScrap`.

Gathering Picture Information

`GetPictInfo` may be used to gather information about a single picture, and `GetPixMapInfo` may be used to gather colour information about a single pixel map or bitmap. Each of these functions returns colour and resolution information in a `PictInfo` record. A `PictInfo` record can also contain information about the drawing objects, fonts, and comments in a picture.

Cursors

Introduction

A **cursor** is a 256-pixel, black-and-white image in a 16-by-16 pixel square usually defined by an application in a cursor (`'CURS'`) or colour cursor (`'crsr'`) resource.

Cursor Movement, Hot Spot, Visibility, Colour and Shape

Cursor Movement

Whenever the user moves the mouse, the low-level interrupt-driven mouse routines move the cursor to a new location on the screen. Your application does not need to do anything to move the cursor.

Cursor Hot Spot

One point in the cursor's image is designated as the **hot spot**, which in turn points to a location on the screen. The hot spot is the part of the pointer that must be positioned over a screen object before mouse clicks can have an effect on that object. Fig 1 illustrates two cursors and their hot spot points. Note that the hot spot is a point, not a bit.

[4]The demonstration program at Chapter 14 — Files shows how to read pictures from, and save pictures to, files of type `'PICT'`.

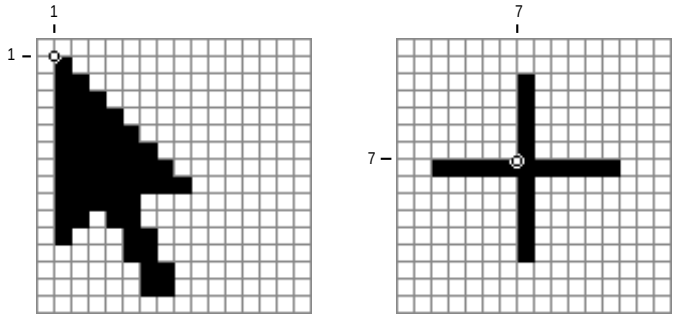

FIG 1 - HOT SPOTS IN CURSORS

Cursor Visibility

In general, you should always make the cursor visible to your application, although there are a few cases where the cursor should not be visible. For example, in a text-editing application, the cursor should be made invisible, and the insertion point made to blink, when the user begins entering text. In such cases, the cursor should be made visible again only when the user moves the mouse.

Cursor Colour

When the cursor is used for choosing or selecting, it should remain black. You may want to display a colour cursor when the user is drawing or typing in colour. To ensure visibility over any background, colour cursors should generally be outlined in black.

Cursor Shape

Your application should change the shape of the cursor in the following circumstances:

- To indicate that the user is over a certain area of the screen. For example, when the cursor is in the menu bar, it should usually have an arrow shape. When the user moves the cursor over a text document, your application should change the cursor to the I-beam shape.

- To provide feedback about the status of the computer system. For example, if an operation will take a second or two, you should provide feedback to the user by changing the cursor to the wristwatch cursor (see Fig 2). If the operation takes several seconds and the user can do nothing in your application but stop the operation, wait until it is completed, or switch to another application, you should display an animated cursor.[5]

The System file in the System Folder contains `'CURS'` resources for the common cursors shown at Fig 2.

FIG 2 - THE I-BEAM, CROSSHAIRS, PLUS SIGN, AND WRISTWATCH CURSORS

The following constants represent the `'CURS'` resource IDs for the cursors shown at Fig 2:

```
iBeamCursor  = 1   Used in text editing.
crossCursor  = 2   Often used for manipulating graphics.
plusCursor   = 3   Often used for selecting field in an array.
watchCursor  = 4   Used when a short operation is in progress.
```

[5]If the operation takes longer than several seconds, you should display a status indicator to show the user the total and elapsed time for the operation. (See Chapter 21 — Miscellany.)

Creating Custom Non-Animated Cursors Resources

To create custom non-animated cursors, you need to:

- Define black-and-white cursors as `'CURS'` resources in the resource file of your application. (You use `'CURS'` resources to create black-and-white cursors for display on black-and-white or colour screens).

- If you want to display colour cursors, define colour cursors in `'crsr'` resources in the resource file of your application. (You use `'crsr'` resources to create colour cursors to display on systems supporting Color QuickDraw. Each `'crsr'` resource also contains a black and white image that Color QuickDraw displays on black and white screens.)[6]

Changing Cursor Shape and Hiding Cursors

Changing Cursor Shape

To change cursor shape, your application must get a handle to the relevant cursor (either a custom cursor or one of the system cursors shown at Fig 2) by specifying its resource ID in a call to `GetCursor` or `GetCCursor`. `GetCursor` returns a handle to a `Cursor` record. `GetCCursor` returns a handle to a `CCrsr` record. The address of the `Cursor` or `CCrsr` record is then used in a call to `SetCursor` or `SetCCursor` to change the cursor shape.

Your application is responsible for setting the initial appearance of the cursor and for changing the appearance of the cursor as appropriate for your application.

In Response to Mouse-Moved Events. For example, most applications set the cursor to the I-beam shape when the cursor is inside a text-editing area of a document, and they change the cursor to an arrow when the cursor is inside the scroll bars. Your application can achieve this effect by requesting that the Event Manager report mouse-moved events if the user moves the cursor out of a region you specify in the `mouseRgn` parameter to the `WaitNextEvent` function. Then, when a mouse-moved event is detected in your main event loop, you can use `SetCursor` or `SetCCursor` to change the cursor to the appropriate shape.[7]

In Response to Resume Events. Your application also needs to adjust the cursor in response to resume events.

Hiding Cursors

You can remove the cursor image from the screen using `HideCursor`. You can hide the cursor temporarily using `ObscureCursor` or you can hide the cursor in a given rectangle by using `ShieldCursor`. To display a hidden cursor, use `ShowCursor`. Note, however, that you do not need to explicitly show the cursor after your application uses `ObscureCursor` because the cursor automatically reappears when the user moves the mouse again.

Creating an Animated Cursor

To create an animated cursor, you should:

- Create a series of `'CURS'` resources that make up the "frames" of the animation. (Typically, an animated cursor uses four to seven frames.)

[6]Before using the routines which handle colour cursors (that is, `GetCCursor`, `SetCCursor`, and `DisposeCCursor`) you should test for the existence of Color QuickDraw using the `Gestalt` function. Both basic and Color QuickDraw support all other routines described in this chapter.

[7]Note that your application may also have to accommodate the cursor shape changing requirements of, say, dialog boxes with editable text items.as well as its main windows.

- Create an 'acur' resource. (The 'acur' resource collects and orders your 'CURS' frames into a single animation. It specifies the IDs of the resources and the sequence for displaying them in your animation.)

- Load the 'acur' resource into an application-defined structure which replicates the structure of an 'acur' resource, for example:

```
Acur = record
  n:      integer;  {Number of cursors ("frames of film").}
  index:  integer;  {Next frame to show <for internal use>.}
  frame1: integer;  {'CURS' resource id for frame #1.}
  fill1:  integer;  {<for internal use>.}
  frame2: integer;  {'CURS' resource id for frame #2.}
  fill2:  integer;  {<for internal use>.}
  frameN: integer;  {'CURS' resource id for frame #N.}
  fillN:  integer;  {<for internal use>.}
  end;

acurPtr = ^Acur;
acurHandle = ^acurPtr;
```

- Load the 'CURS' resources using GetCursor and assign handles to the resulting Cursor structures to the elements of the frame field.

- Call SetCursor to display each cursor, that is, each "frame", in rapid succession, returning to the first frame after the last frame has been displayed. This can be achieved by incrementing the frame at each null event (which means, of course, that the sleep parameter in the WaitNextEvent call must be set to the required interval between frame updates)[8].

Icons

Icons and the Finder

As stated at Chapter 7 — Finder Interface, the Finder uses **icons** to graphically represent objects, such as files and directories, on the desktop. Chapter 7 also introduced the subject of **icon families**, and stated that your application should provide the Finder with a family of specially designed icons for the application file itself and for each of the document types created by the application.

The provision of a family of icon types for each desktop object, rather than just one icon type, enables the Finder to automatically select the appropriate family member to display depending on the icon size specified by the user and the bit depth of the display device. Chapter 7 described the components of an icon family used by the Finder as follows:

Icon	Size (Pixels)	Resource in Which Defined
Large black-and-white icon, and mask	32 by 32	Icon list ('ICN#').
Small black-and-white icon, and mask	16 by 16	Small icon list ('ics#')
Large colour icon with 4 bits of colour data per pixel	32 by 32	Large 4-bit colour icon ('icl4')
Small colour icon with 4 bits of colour data per pixel	16 by 16	Small 4-bit colour icon ('ics4')
Large colour icon with 8 bits of colour data per pixel	32 by 32	Large 8-bit colour icon ('icl8')
Small colour icon with 8 bits of colour data per pixel	16 by 16	Small 8-bit colour icon ('ics8')

Other Icons — Icons, Colour Icons, and Small Icons

Icon ('ICON'). The icon is defined in an 'ICON' resource, which contains a bitmap for a 32-by-32 pixel black-and-white icon. Because it is always displayed on a white background, it does not need a mask.

[8]An alternative method for incrementing the frame, using vertical blanking tasks, is demonstrated at Chapter 19 — Custom Control Definition Functions and VBL Tasks. But note that the vertical blanking task method is not recommended.

Colour Icon (`'cicn'`). The colour icon is defined in a `'cicn'` resource, which has a special format which includes a pixel map, a bitmap, and a mask. You can use a `'cicn'` resource to define a colour icon with a width and height between 8 and 64 pixels. You can also define the bit depth for a colour icon resource.

Small Icon (`'SICN'`). The small icon is defined in a `'SICN'` resource. Small icons are 12 by 16 pixels even though they are stored in a resource as 16-by-16 pixel bitmaps. A `'SICN'` resource consists of a list of 16-by-16 pixel bitmaps for black-and-white icons.[9]

Note that the Finder does not use or display these types of icon.

Icons in Windows, Menus, and Alert and Dialog Boxes

The icons provided by your application for the Finder (or the default system-suppled icons used by the Finder if your application does not provide its own icons) are displayed on the desktop. Your application can also display icons in its menus, dialog boxes and windows.

Icons in Windows

You can display icons of any kind in your windows using the appropriate Icon Utilities routines.

Icons in Menus

The Menu Manager allows you to display icons of resource types `'ICON'` (icon) `'cicn'` (colour icon), and `'SICN'` (small icon) in menu items. The procedure is as follows:

- Create the icon resource with a resource ID between 257 and 511. Subtract 256 from the resource ID to get a value called the **icon number**. Specify the icon number in the Icon field of the menu item definition.

- For an icon (`'ICON'`), specify `$1D` in the keyboard equivalent field of the menu item definition to indicate to the Menu Manager that the icon should be reduced to fit into a 16-by-16 pixel rectangle. Otherwise, specify a value of `$00`, or a value greater than `$20`, in the keyboard equivalent field to cause the Menu Manager to expand the item's rectangle so as to display the icon at its normal 32-by-32 pixel size. (A value greater than `$20` in the keyboard equivalent field specifies the item's keyboard equivalent.)

- For a colour icon (`'cicn'`), specify `$00` or a value greater than `$20` in the keyboard equivalent field. The Menu Manager automatically enlarges the enclosing rectangle of the menu item according to the rectangle specified in the `'cicn'` resource. (Colour icons, unlike icons, can be any height or width between 8 and 64 pixels.)

- For a small icon (`'SICN'`), specify `$1E` in the keyboard equivalent field. This indicates that the item has an icon defined by a `'SICN'` resource. The Menu Manager plots the icon in a 16-by-16 pixel rectangle.

The Menu Manager will then automatically display the icon whenever you display the menu using the `MenuSelect` function. The Menu Manager first looks for a `'cicn'` resource with the resource ID calculated from the icon number and displays that icon if it is found.[10] If a `'cicn'` resource is not found (or if the computer does not have Color QuickDraw) and the keyboard equivalent field specifies `$1E`, the Menu Manager looks for a `'SICN'` resource with the calculated resource ID. Otherwise, the Menu Manager searches for an `'ICON'` resource and plots it in either a 32-by-32 pixel rectangle or a 16-by-16 bit rectangle, depending on the value in the menu item's keyboard equivalent field.[11]

---

[9] Typically, only the Finder and the Standard File Package use small icons.

[10] A colour icon (`'cicn'`) resource contains a bitmap as well as a pixel map, which accounts for black-and-white displays.

[11] Note that, for the Apple and Application menus, the Menu Manager either automatically reduces the icon to fit within the enclosing rectangle of the menu item or uses the appropriate icon from the application's icon family, such as the `'icl8'` resource, if one is available.

Displaying Other Icon Types. To display an icon of a resource type other than `'ICON'`, `'cicn'`, and `'SICN'` in your menu items, you must write your own menu definition procedure.

Icons in Alert and Dialog Boxes

The Dialog Manager allows you to display icons of resource types `'ICON'` (icon) and `'cicn'` (colour icon) in alert and dialog boxes. The procedure is to define an item of type `Icon` and provide the resource ID of the icon in the item list (`'DITL'`) resource for the dialog. This will cause the Dialog Manager to automatically display the icon whenever you display the alert or dialog box using Dialog Manager routines.

If you provide a colour icon (`'cicn'`) resource with the same resource ID as an icon (`'ICON'`) resource, the Dialog Manager displays the colour icon instead of the black-and-white icon.

Ordinarily, you would use the `Alert` function, which does not automatically draw a system-supplied alert icon in the alert box, when you wish to display an alert containing your own icon (for example, in your application's **About...** alert box). If you invoke an alert box with the `NoteAlert`, `CautionAlert` or `StopAlert` functions, rather than the `Alert` function, the Dialog Manager draws the system-supplied black-and-white icon as well as your icon. Since your icon is drawn last, you can obscure the system-suppled icon by positioning your icon at the same coordinates.

Displaying Other Icon Types. To display an icon of a resource type other than `'ICON'` and `'cicn'` in a dialog box, you must define an item of type `userItem` and use the appropriate Icon Utilities routine to draw the icons.

Drawing and Manipulating Icons

The Icon Utilities allow your application (and the system software) to draw and manipulate icons of any standard resource type in windows and, subject to the limitations and requirements previously described, in menus and dialog boxes.

You need to use Icon Utilities routines only if:

- You wish to draw icons in your application's windows.
- You wish to draw icons which are not recognised by the Menu Manager and the Dialog Manager in, respectively, menu items and dialog boxes.

Preamble - Icon Families, Suites, and Caches

Icon Families. You can define individual icons of resource types `'ICON'`, `'cicn'`, and `'SICN'` that are not part of an icon family and use Icon Utilities routines to draw them as required. However, to display an icon effectively at a variety of sizes and bit depths, you should provide an icon family[12] in the same way that you provide icon families for the Finder. The advantage of providing an icon family is that you can then leave it to routines such as `PlotIconID`, which are used to draw icons, to automatically determine which icon in the icon family is best suited to the specified destination rectangle and current display bit depth.

Icon Suites. Some Icon Utilities routines take as a parameter a handle to an **icon suite**. An icon suite typically consists of one or more handles to icon resources from a single icon family which have been read into memory. The `GetIconSuite` function may be used to get a handle to an icon suite, which can then be passed to routines such as `PlotIconSuite` to draw that icon in the icon suite best suited to the destination rectangle and current display bit depth. An icon suite can contain handles to each of the six icon resources that an icon family can contain, or it can contain handles to only a subset of the icon resources in an icon family. For best results, an icon suite should always include a resource of type

[12] Each icon in an icon family shares the same resource ID as other icons in the family but has its own resource type identifying the icon data it contains.

'ICN#' in addition to any other large icons you provide and a resource of type 'ics#' in addition to any other small icons you provide.[13]

Icon Cache. An **icon cache** is like an icon suite except that it also contains a pointer to an application-defined **icon getter function** and a pointer to data that is associated with the icon suite. You can pass a handle to an icon cache to any of the Icon Utilities routines which accept a handle to an icon suite. An icon cache typically does not contain handles to the icon resources for all icon family members. Instead, if the icon cache does not contain an entry for a specific type of icon in an icon family, the Icon Utilities routines call your application's icon getter function to retrieve the data for that icon type.

Drawing an Icon Directly From a Resource

To draw an icon from an icon family without first creating an icon suite, use the PlotIconID function. PlotIconID determines, from the size of the specified destination rectangle and the current bit depth of the display device, which icon to draw. The icon drawn is as follows;

Destination Rectangle Size	Icon Drawn
Width or height greater than or equal to 32.	The 32-by-32 pixel icon with the appropriate bit depth.
Less than 32 by 32 pixels and greater than 16 pixels wide or 12 pixels high.	The 16-by-16 pixel icon with the appropriate bit depth.
Height less than or equal to 12 pixels or width less than or equal to 16 pixels.	The 12-by-16 pixel icon with the appropriate bit depth.

Icon Stretching and Shrinking. Depending on the size of the rectangle, PlotIconID may stretch or shrink the icon to fit. To draw icons without stretching them, PlotIconID requires that the destination rectangle have the same dimensions as one of the standard icons.

Icon Alignment and Transform. In addition to destination rectangle and resource ID parameters, PlotIconID takes **alignment** and **transform** parameters. Icon Utilities routines can automatically align an icon within its destination rectangle. (For example, an icon which is taller than it is wide can be aligned to either the right or left of its destination rectangle.) These routines can also transform the appearance of the icon in standard ways analogous to Finder states for icons.

Variables of type IconAlignmentType and IconTransformType should be declared and assigned values representing alignment and transform requirements. Constants, such as atAbsoluteCenter and ttNone, are available to specify alignment and transform requirements.

Getting an Icon Suite and Drawing One of Its Icons

The GetIconSuite function, with the constant svAllAvailableData specified in the third parameter, is used to get all icons from an icon family with a specified resource ID and to collect the handles to the data for each icon into an icon suite. An icon from this suite may then be drawn using PlotIconSuite which, like PlotIconID, takes destination rectangle, alignment and transform parameters and stretches or shrinks the icon if necessary.

Drawing Specific Icons From an Icon Family

If you need to plot a specific icon from an icon family rather than use the Icon Utilities to automatically select a family member, you must first create an icon suite which contains only the icon of the desired resource type together with its corresponding mask. Constants such as svLarge4Bit (an icon selector mask for an 'icl4' icon) are used as the third parameter of the GetIconSuite call to retrieve the required family member. You can then use PlotIconSuite to plot the icon.

[13] When you create an icon suite from icon family resources, the associated resource file should remain open while you use Icon Utilities routines.

Drawing Icons That Are Not Part of an Icon Family

To draw icons of resource type `'ICON'` and `'cicn'` in menu items and dialog boxes, and icons of resource type `'SICN'` in menu items, you use Menu Manager and Dialog Manager routines such as `SetItemIcon` and `SetDialogItem`.

To draw resources of resource type `'ICON'`, `'cicn'`, and `'SICN'` in your application's windows, you use the following routines:

Resource Type	Routine to Get Icon	Routines to Draw Icon
`'ICON'`	`GetIcon`	`PlotIconHandle` `PlotIcon`
`'cicn'`	`GetCIcon`	`PlotCIconHandle` `PlotCIcon`
`'SICN'`	`GetResource`	`PlotSICNHandle`

The routines in this list ending in `Handle` allow you to specify alignment and transforms for the icon.

Manipulating Icons

The `GetIconFromSuite` function may be used to get a handle to the pixel data for a specific icon from an icon suite. You can then use this handle to manipulate the icon data, for example, to alter its colour or add three-dimensional shading.

The Icon Utilities also include routines which allow you to perform an action on one or more icons in an icon suite and to perform hit testing on icons.

Main Constants, Data Types and Routines — Offscreen Graphics Worlds

Constants

Flags for `GWorldFlags` Parameter

```
pixPurgeBit         = 0  {Set to make base address for offscreen pixel image purgeable.}
noNewDeviceBit      = 1  {Set to not create a new GDevice record for offscreen world.}
pixelsPurgeableBit  = 6  {Set to make base address for pixel image purgeable.}
pixelsLockedBit     = 7  {Set to lock base address for offscreen pixel image.}
```

Data Types

```
GWorldPtr = CGrafPtr;
GWorldFlags = longint;
```

Routines

Creating, Altering, and Disposing of Offscreen Graphics Worlds

```
function    NewGWorld(var offscreenGWorld: GWorldPtr; PixelDepth: integer;
            var boundsRect: Rect; cTable: CTabHandle; aGDevice: GDHandle;
            flags: GWorldFlags): QDErr;
function    UpdateGWorld(var offscreenGWorld: GWorldPtr; pixelDepth: integer;
            var boundsRect: Rect; cTable: CTabHandle; aGDevice: GDHandle;
            flags: GWorldFlags): GWorldFlags;
procedure   DisposeGWorld(offscreenGWorld: GWorldPtr);
```

Saving and Restoring Graphics Ports and Offscreen Graphics Worlds

```
procedure   GetGWorld(var port: CGrafPtr; var gdh: GDHandle);
procedure   SetGWorld(port: CGrafPtr; gdh: GDHandle);
```

Managing an Offscreen Graphics World's Pixel Image

```
function   GetGWorldPixMap(offscreenGWorld: GWorldPtr): PixMapHandle;
function   LockPixels(pm: PixMapHandle): boolean;
procedure  UnlockPixels(pm: PixMapHandle);
procedure  AllowPurgePixels(pm: PixMapHandle);
procedure  NoPurgePixels(pm: PixMapHandle);
function   GetPixelsState(pm: PixMapHandle): GWorldFlags;
procedure  SetPixelsState(pm: PixMapHandle; state: GWorldFlags);
function   GetPixBaseAddr(pm: PixMapHandle): Ptr;
function   PixMap32Bit(pmHandle: PixMapHandle): boolean;
```

Main Constants, Data Types and Routines — Pictures

Constants

Verbs for the `GetPictInfo` , `GetPixMapInfo` , and `NewPictInfo` calls

```
returnColorTable       = $0001Return a ColorTable record.
returnPalette          = $0002Return a Palette record.
recordComments         = $0004Return comment information.
recordFontInfo         = $0008Return font information.
suppressBlackAndWhite  = $0010Do not include black and white.
```

Data Types

Picture

```
type
Picture = record
  picSize:         integer;          {For a version 1 picture: its size.}
  picFrame:        Rect;             {Bounding rectangle for the picture.}
  …                                  {Picture definition (variable length).}
  end;

PicPtr = ^Picture;
PicHandle = ^PicPtr;
```

OpenCPicParams

```
OpenCPicParams = record
  srcRect:         Rect;             {Optimal bounding rectangle.}
  hRes:            Fixed;            {Best horizontal resolution.}
  vRes:            Fixed;            {Best vertical resolution.}
  version:         integer;          {Set to -2.}
  reserved1:       integer;          {(Reserved.  Set to 0.)}
  reserved2:       longint;          {(Reserved.  Set to 0.)}
  end;
```

PictInfo

```
PictInfo = record
  version:         integer;          {This is always zero, for now.}
  uniqueColors:    longint;          {The number of actual colors in picture(s)/pixmap(s)}
  thePalette:      PaletteHandle;    {Handle to the palette information.}
  theColorTable:   CTabHandle;       {Handle to the color table.}
  hRes:            Fixed;            {Maximum horizontal resolution for all the pixmaps.}
  vRes:            Fixed;            {Maximum vertical resolution for all the pixmaps.}
  depth:           integer;          {Maximum depth for all the pixmaps (in the picture).}
  sourceRect:      Rect;             {P frame rectangle (this contains entire picture).}
  textCount:       longint;          {Total number of text strings in the picture.}
  lineCount:       longint;          {Total number of lines in the picture.}
  rectCount:       longint;          {Total number of rectangles in the picture.}
  rRectCount:      longint;          {Total number of round rectangles in the picture.}
  ovalCount:       longint;          {Total number of ovals in the picture.}
  arcCount:        longint;          {Total number of arcs in the picture.}
  polyCount:       longint;          {Total number of polygons in the picture.}
  regionCount:     longint;          {Total number of regions in the picture.}
  bitMapCount:     longint;          {Total number of bitmaps in the picture.}
  pixMapCount:     longint;          {Total number of pixmaps in the picture.}
  commentCount:    longint;          {Total number of comments in the picture.}
```

```
  uniqueComments:  longint;           {The number of unique comments in the picture.}
  commentHandle:   CommentSpecHandle; {Handle to all the comment information.}
  uniqueFonts:     longint;           {The number of unique fonts in the picture.}
  fontHandle:      FontSpecHandle;    {Handle to the FontSpec information.}
  fontNamesHandle: Handle;            {Handle to the font names.}
  reserved1:       longint;
  reserved2:       longint;
  end;

  PictInfoPtr = ^PictInfo;
  PictInfoHandle = ^PictInfoPtr;
```

CommentSpec

```
CommentSpec = record
  count:           integer;           {Number of occurrances of this comment ID.}
  ID:              integer;           {ID for the comment in the picture.}
  end;

CommentSpecPtr = ^CommentSpec;
CommentSpecHandle = ^CommentSpecPtr;
```

FontSpec

```
FontSpec = record
  pictFontID:      integer;           {ID of the font in the picture.}
  sysFontID:       integer;           {ID of the same font in the current system file.}
  size: array [0..3] of longint;      {Bit array of all the sizes found (1..127)}
                                     {(bit 0 means > 127).}
  style:           integer;           {Combined style of all occurrances of the font.}
  nameOffset:      longint;           {Offset into the fontNamesHdl handle for the font's}
                                     {name.}
  end;

FontSpecPtr = ^FontSpec;
FontSpecHandle = ^FontSpecPtr;
```

Routines

Creating and Disposing of Pictures

```
function   OpenCPicture(var newHeader: OpenCPicParams): PicHandle;
function   OpenPicture(var picFrame: Rect): PicHandle;
procedure  PicComment(kind: integer; dataSize: integer; dataHandle: Handle);
procedure  ClosePicture;
procedure  KillPicture(myPicture: PicHandle);
```

Drawing Pictures

```
procedure  DrawPicture(myPicture: PicHandle; var dstRect: Rect);
function   GetPicture(pictureID: integer): PicHandle;
```

Collecting Picture Information

```
function   GetPictInfo(thePictHandle: PicHandle; var thePictInfo: PictInfo; verb: integer;
           colorsRequested: integer; colorPickMethod: integer; version: integer): OSErr;
function   GetPixMapInfo(thePixMapHandle: PixMapHandle; var thePictInfo: PictInfo;
           verb: integer; colorsRequested: integer; colorPickMethod: integer;
           version: integer): OSErr;
function   NewPictInfo(var thePictInfoID: PictInfoID; verb: integer; colorsRequested: integer;
           colorPickMethod: integer; version: integer): OSErr;
function   RecordPictInfo(thePictInfoID: PictInfoID; thePictHandle: PicHandle): OSErr;
function   RecordPixMapInfo(thePictInfoID: PictInfoID; thePixMapHandle: PixMapHandle): OSErr;
function   RetrievePictInfo(thePictInfoID: PictInfoID; var thePictInfo: PictInfo;
           colorsRequested: integer): OSErr;
function   DisposePictInfo(thePictInfoID: PictInfoID): OSErr;
```

Main Constants, Data Types and Routines — Cursors

Constants

```
iBeamCursor   = 1
crossCursor   = 2
plusCursor    = 3
watchCursor   = 4
```

Data Types

Cursor

```
Cursor = record
  data:         Bits16;
  mask:         Bits16;
  hotSpot:      Point;
  end;

  CursPtr = ^Cursor;
  CursHandle = ^CursPtr;
```

CCrsr

```
CCrsr = record
  crsrType:     integer;      {Type of cursor.}
  crsrMap:      PixMapHandle; {The cursor's pixmap.}
  crsrData:     Handle;       {Cursor's data.}
  crsrXData:    Handle;       {Expanded cursor data.}
  crsrXValid:   integer;      {Depth of expanded data (0 if none).}
  crsrXHandle:  Handle;       {Future use.}
  crsr1Data:    Bits16;       {One-bit cursor.}
  crsrMask:     Bits16;       {Cursor's mask.}
  crsrHotSpot:  Point;        {Cursor's hotspot.}
  crsrXTable:   longint;      {Private.}
  crsrID:       longint;      {Private.}
  end;

CCrsrPtr = ^CCrsr;
CCrsrHandle = ^CCrsrPtr;
```

Routines

Initialising Cursors

```
procedure  InitCursor;
procedure  InitCursorCtl(newCursors: UNIV acurHandle);
```

Changing Black-and-White Cursors

```
function   GetCursor(cursorID: integer): CursHandle;
procedure  SetCursor(var crsr: Cursor);
```

Changing Colour Cursors

```
function   GetCCursor(crsrID: integer): CCrsrHandle;
procedure  SetCCursor(cCrsr: CCrsrHandle);
procedure  AllocCursor;
procedure  DisposeCCursor(cCrsr: CCrsrHandle);
```

Hiding and Showing Cursors

```
procedure  HideCursor;
procedure  ShowCursor;
procedure  ObscureCursor;
procedure  ShieldCursor(var shieldRect: Rect; offsetPt: Point);
```

Main Constants, Data Types and Routines — Icons

Constants

Types for Icon Families

```
kLarge1BitMask              = 'ICN#';
kLarge4BitData              = 'icl4';
kLarge8BitData              = 'icl8';
kSmall1BitMask              = 'ics#';
kSmall4BitData              = 'ics4';
kSmall8BitData              = 'ics8';
kMini1BitMask               = 'icm#';
kMini4BitData               = 'icm4';
kMini8BitData               = 'icm8';
```

`IconAlignmentType` **Values**

```
kAlignNone                  = $0;
kAlignVerticalCenter        = $1;
kAlignTop                   = $2;
kAlignBottom                = $3;
kAlignHorizontalCenter      = $4;
kAlignAbsoluteCenter        = kAlignVerticalCenter + kAlignHorizontalCenter;
kAlignCenterTop             = kAlignTop + kAlignHorizontalCenter;
kAlignCenterBottom          = kAlignBottom + kAlignHorizontalCenter;
kAlignLeft                  = $8;
kAlignCenterLeft            = kAlignVerticalCenter + kAlignLeft;
kAlignTopLeft               = kAlignTop + kAlignLeft;
kAlignBottomLeft            = kAlignBottom + kAlignLeft;
kAlignRight                 = $C;
kAlignCenterRight           = kAlignVerticalCenter + kAlignRight;
kAlignTopRight              = kAlignTop + kAlignRight;
kAlignBottomRight           = kAlignBottom + kAlignRight;
```

`IconTransformType` **Values**

```
kTransformNone              = $0;
kTransformDisabled          = $1;
kTransformOffline           = $2;
kTransformOpen              = $3;
kTransformLabel1            = $0100;
kTransformLabel2            = $0200;
kTransformLabel3            = $0300;
kTransformLabel4            = $0400;
kTransformLabel5            = $0500;
kTransformLabel6            = $0600;
kTransformLabel7            = $0700;
kTransformSelected          = $4000;
kTransformSelectedDisabled  = kTransformSelected + kTransformDisabled;
kTransformSelectedOffline   = kTransformSelected + kTransformOffline;
kTransformSelectedOpen      = kTransformSelected + kTransformOpen;
```

`IconSelectorValue` **Masks**

```
kSelectorLarge1Bit          = $00000001;   {'ICN#' resource.}
kSelectorLarge4Bit          = $00000002;   {'icl4' resource.}
kSelectorLarge8Bit          = $00000004;   {'icl8' resource.}
kSelectorSmall1Bit          = $00000100;   {'ics#' resource.}
kSelectorSmall4Bit          = $00000200;   {'ics4' resource.}
kSelectorSmall8Bit          = $00000400;   {'ics8' resource.}
kSelectorMini1Bit           = $00010000;   {'ism#' resource.}
kSelectorMini4Bit           = $00020000;   {'icm4' resource.}
kSelectorMini8Bit           = $00040000;   {'icm8' resource.}
kSelectorAllLargeData       = $000000FF;   {'ICN#', 'icl4', and 'icl8' resources.}
kSelectorAllSmallData       = $0000FF00;   {'ics#', 'ics4', and 'ics8' resources.}
kSelectorAllMiniData        = $00FF0000;   {'icm#', 'icm4', and 'icm8' resources.}
kSelectorAll1BitData        = kSelectorLarge1Bit + kSelectorSmall1Bit + kSelectorMini1Bit;
kSelectorAll4BitData        = kSelectorLarge4Bit + kSelectorSmall4Bit + kSelectorMini4Bit;
kSelectorAll8BitData        = kSelectorLarge8Bit + kSelectorSmall8Bit + kSelectorMini8Bit;
kSelectorAllAvailableData   = $FFFFFFFF;   {All resources of given ID.}
```

Data Types

```
IconAlignmentType = SInt16;
IconTransformType = SInt16;
```

CIcon

```
CIcon = record
  iconPMap:     PixMap;                 {The icon's pixMap}
  iconMask:     BitMap;                 {The icon's mask}
  iconBMap:     BitMap;                 {The icon's bitMap}
  iconData:     Handle;                 {The icon's data}
  iconMaskData: array [0..0] of SInt16; {Icon's mask and BitMap data}
  end;

CIconPtr = ^CIcon;
CIconHandle = ^CIconPtr;
```

Routines

Drawing Icons From Resources

```
function   PlotIconID(var theRect: Rect; align: IconAlignmentType;
           transform: IconTransformType; theResID: SInt16): OSErr;
procedure  PlotIcon(var theRect: Rect; theIcon: Handle);
function   PlotIconHandle(var theRect: Rect; align: IconAlignmentType;
           transform: IconTransformType; theIcon: Handle): OSErr;
procedure  PlotCIcon(var theRect: Rect; theIcon: CIconHandle);
function   PlotCIconHandle(var theRect: Rect; align: IconAlignmentType;
           transform: IconTransformType; theCIcon: CIconHandle): OSErr;
function   PlotSICNHandle(var theRect: Rect; align: IconAlignmentType;
           transform: IconTransformType; theSICN: Handle): OSErr;
```

Getting Icons From Resources Which do Not Belong to an Icon Family

```
function   GetIcon(iconID: SInt16): Handle;
function   GetCIcon(iconID: SInt16): CIconHandle;
```

Disposing of Icons

```
procedure  DisposeCIcon(theIcon: CIconHandle);
```

Creating an Icon Suite

```
function   GetIconSuite(var theIconSuite: Handle; theResID: Sint16;
           selector: IconSelectorValue): OSErr;
function   NewIconSuite(var theIconSuite: Handle): OSErr;
function   AddIconToSuite(theIconData: Handle; theSuite: Handle; theType: ResType): OSErr;
```

Getting Icons From an Icon Suite

```
function   GetIconFromSuite(var theIconData: Handle; theSuite: Handle;
           theType: ResType): OSErr;
```

Drawing Icons From an Icon Suite

```
function   PlotIconSuite(var theRect: Rect; align: IconAlignmentType; transform:
           IconTransformType; theIconSuite: Handle): OSErr;
```

Performing Operations on Icons in an Icon Suite

```
function   ForEachIconDo(theSuite: Handle; selector: IconSelectorValue; action: IconActionUPP;
           yourDataPtr: UNIV Ptr): OSErr;
```

Disposing of Icon Suites

```
function   DisposeIconSuite(theIconSuite: Handle; disposeData: boolean): OSErr;
```

Converting an Icon Mask to a Region

```
function   IconIDToRgn(theRgn: RgnHandle; var iconRect: Rect; align: IconAlignmentType;
           iconID: SInt16): OSErr;
```

```
function   IconSuiteToRgn(theRgn: RgnHandle; var iconRect: Rect;
           align: IconAlignmentType; theIconSuite: Handle): OSErr;
```

Determining Whether a Point or Rectangle is Within an Icon

```
function   PtInIconID(testPt: Point; var iconRect: Rect; align: IconAlignmentType;
           iconID: SInt16): boolean;
function   PtInIconSuite(testPt: Point; var iconRect: Rect; align: IconAlignmentType;
           theIconSuite: Handle): boolean;
function   RectInIconID(var testRect: Rect; var iconRect: Rect;
           align: IconAlignmentType; iconID: SInt16): boolean;
function   RectInIconSuite(var testRect: Rect; var iconRect: Rect;
           align: IconAlignmentType; theIconSuite: Handle): boolean;
```

Working With Icon Caches

```
function   MakeIconCache(var theHandle: Handle; makeIcon: IconGetterUPP;
           yourDataPtr: UNIV Ptr): OSErr;
function   LoadIconCache(var theRect: Rect; align: IconAlignmentType;
           transform: IconTransformType; theIconCache: Handle): OSErr;
```

Demonstration Program

```
1  { ############################################################################################
2  // GWorldPicCursIconPascal.p
3  // ############################################################################################
4  //
5  // This program:
6  //
7  // •   Opens a window in which the results of various drawing operations are displayed,
8  //     and in which regions are established for a cursor shape change demonstration.
9  //
10 // •   Demonstrates offscreen graphics world, picture, cursor, animated cursor, and icon
11 //     operations as a result of the user choosing items from a Demonstration menu.
12 //
13 // •   Quits when the user chooses Quit or clicks the window's close box.
14 //
15 // The program utilises the following resources:
16 //
17 // •   'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
18 //
19 // •   A 'WIND' resource (purgeable) (initially visible).
20 //
21 // •   An 'acur' resource (purgeable).
22 //
23 // •   'CURS' resources associated with the 'acur' resource (purgeable).
24 //
25 // •   An 'ALRT' resource (purgeable) and associated 'DITL' resource (purgeable) for an
26 //     About GWorldPicCursIcon… alert box, which is used to demonstrate the display of
27 //     icons in alert boxes.
28 //
29 // •   'ICON', 'cicn', and 'SICN' resources (purgeable) for the display of icons in menu
30 //     items and the About GWorldPicCursIcon… alert box.
31 //
32 // •   A 'SIZE' resource with the acceptSuspendResumeEvents  & is32BitCompatible flags set.
33 //
34 // ############################################################################################ }
35
36 program GWorldPicCursIconPascal(input, output);
37
38 { …… include the following Universal Interfaces }
39
40 uses
41
42   Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
43   Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, QDOffscreen, Resources, Icons,
44   GestaltEqu, PictUtils, SegLoad, Sound;
45
46 { …… define the following constants }
47
48 const
49
50 mApple = 128;
```

```
  iAbout = 1;
mFile = 129;
  iQuit = 11;
mDemonstration = 131;
  iWithoutOffScreenGWorld = 1;
  iWithOffScreenGWorld = 2;
  iPicture = 3;
  iCursor = 4;
  iAnimatedCursor = 5;
  iIcon = 6;

rAlert = 128;
rMenubar = 128;
rWindow = 128;
rBeachBallCursor = 128;
rIcon = 257;

kBeachBallTickInterval = 5;

kMaxLong = $7FFFFFFF;

{ ……… type definitions }

type

animCurs = record
  numberOfFrames : integer;
  whichFrame : integer;
  frame : array [0..8] of CursHandle;
  end;
animCursPtr = ^animCurs;
animCursHandle = ^animCursPtr;

{ ……… global variables }

var

gDone : boolean;
gWindowPtr : WindowPtr;
gSleepTime : longint;
gCursorRegion : RgnHandle;
gInBackground : boolean;
gCursorRegionsActive : boolean;
gAnimCursHdl : animCursHandle;
gAnimCursActive : boolean;
gAnimCursTickInterval : integer;
gAnimCursLastTick : longint;
menubarHdl : Handle;
menuHdl : MenuHandle;

{ ############################################################################ DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ################################################################################ DoIcon }

procedure DoIcon;

  var
  theErr : OSErr;
  response : longint;
```

```
  finalTicks : UInt32;
  a : integer;
  theRect : Rect;
  iconHdl : Handle;
  cIconHdl : CIconHandle;

  begin
  BackColor(whiteColor);
  FillRect(gWindowPtr^.portRect, qd.white);

  SetRect(theRect, 2, 130, 34, 162);

  theErr := Gestalt(gestaltQuickdrawVersion, response);
  if (response < gestalt8BitQD)
    thenbegin
        iconHdl := GetIcon(rIcon);
        for a := 1 to 19 do
          begin
          PlotIcon(theRect, iconHdl);
          InsetRect(theRect, a*(-1), a*(-2));
          OffsetRect(theRect, a*4, 0);
          Delay(20, finalTicks);
          end
        end
    elsebegin
        cIconHdl := GetCIcon(rIcon);
        for a := 1 to 19 do
          begin
          PlotCIcon(theRect, cIconHdl);
          InsetRect(theRect, a*(-1), a*(-2));
          OffsetRect(theRect, a*4, 0);
          Delay(20, finalTicks);
          end;
        DisposeCIcon(cIconHdl);
        end;
  end;
    {of procedure DoIcon}

{ ############################################################################ ReleaseAnimCursor }

procedure ReleaseAnimCursor;

  var
  a : integer;

  begin
  for a := 0 to (gAnimCursHdl^^.numberOfFrames - 1) do
    ReleaseResource(Handle(gAnimCursHdl^^.frame[a]));

  ReleaseResource(Handle(gAnimCursHdl));
  end;
    {of procedure ReleaseAnimCursor}

{ ################################################################################ SpinAnimCursor }

procedure SpinAnimCursor;

  var
  newTick : longint;

  begin
  newTick := TickCount;
  if (newTick < (gAnimCursLastTick + gAnimCursTickInterval)) then
    Exit(SpinAnimCursor);

  SetCursor(gAnimCursHdl^^.frame[gAnimCursHdl^^.whichFrame]^^);
  gAnimCursHdl^^.whichFrame := gAnimCursHdl^^.whichFrame + 1;
  if (gAnimCursHdl^^.whichFrame = gAnimCursHdl^^.numberOfFrames) then
    gAnimCursHdl^^.whichFrame := 0;

  gAnimCursLastTick := newTick;
  end;
    {of procedure SpinAnimCursor}

{ ############################################################################ GetAnimCursor }

function GetAnimCursor(resourceID, tickInterval : integer) : boolean;
```

```
205
206     var
207     cursorID, a : integer;
208     noError : boolean;
209
210     begin
211     noError := false;
212     a := 0;
213
214     gAnimCursHdl := animCursHandle(GetResource('acur', resourceID));
215     if (gAnimCursHdl <> nil) then
216       begin
217       noError := true;
218       while ((a < gAnimCursHdl^^.numberOfFrames)  and noError) do
219         begin
220         cursorID := integer(HiWord(longint(gAnimCursHdl^^.frame[a])));
221         gAnimCursHdl^^.frame[a] := GetCursor(cursorID);
222         if (gAnimCursHdl^^.frame[a] <> nil)
223           then a := a + 1
224           else noError := false;
225         end;
226       end;
227
228     if (noError) then
229       begin
230       gAnimCursTickInterval := tickInterval;
231       gAnimCursLastTick := TickCount;
232       gAnimCursHdl^^.whichFrame := 0;
233       end;
234
235     GetAnimCursor := noError;
236     end;
237       {of function GetAnimCursor}
238
239   { ############################################################################ DoAnimCursor }
240
241   procedure DoAnimCursor;
242
243     var
244     animCursResourceID, animCursTickInterval : integer;
245
246     begin
247     BackColor(whiteColor);
248     FillRect(gWindowPtr^.portRect, qd.white);
249
250     animCursResourceID := rBeachBallCursor;
251     animCursTickInterval := kBeachBallTickInterval;
252
253     if (GetAnimCursor(animCursResourceID, animCursTickInterval))
254       thenbegin
255           gAnimCursActive := true;
256           gSleepTime := animCursTickInterval;
257           end
258       elseSysBeep(10);
259     end;
260       {of procedure DoAnimCursor}
261
262   { ############################################################################ ChangeCursor }
263
264   procedure ChangeCursor(gWindowPtr : WindowPtr; cursorRegion : RgnHandle);
265
266     var
267     cursorRect : Rect;
268     arrowCursorRgn : RgnHandle;
269     ibeamCursorRgn : RgnHandle;
270     crossCursorRgn : RgnHandle;
271     plusCursorRgn : RgnHandle;
272     mousePosition : Point;
273
274     begin
275     arrowCursorRgn := NewRgn;
276     ibeamCursorRgn := NewRgn;
277     crossCursorRgn := NewRgn;
278     plusCursorRgn  := NewRgn;
279
280     SetRectRgn(arrowCursorRgn, -32768, -32768, 32766, 32766);
281
```

```
282     cursorRect := gWindowPtr^.portRect;
283     LocalToGlobal(cursorRect.topLeft);
284     LocalToGlobal(cursorRect.botRight);
285
286     InsetRect(cursorRect, 40, 40);
287     RectRgn(ibeamCursorRgn, cursorRect);
288     DiffRgn(arrowCursorRgn, ibeamCursorRgn, arrowCursorRgn);
289
290     InsetRect(cursorRect, 40, 40);
291     RectRgn(crossCursorRgn, cursorRect);
292     DiffRgn(ibeamCursorRgn, crossCursorRgn, ibeamCursorRgn);
293
294     InsetRect(cursorRect, 40, 40);
295     RectRgn(plusCursorRgn, cursorRect);
296     DiffRgn(crossCursorRgn, plusCursorRgn, crossCursorRgn);
297
298     GetMouse(mousePosition);
299     LocalToGlobal(mousePosition);
300
301     if (PtInRgn(mousePosition, ibeamCursorRgn)) then
302       begin
303       SetCursor(GetCursor(iBeamCursor)^^);
304       CopyRgn(ibeamCursorRgn, cursorRegion);
305       end
306     else if (PtInRgn(mousePosition, crossCursorRgn)) then
307       begin
308       SetCursor(GetCursor(crossCursor)^^);
309       CopyRgn(crossCursorRgn, cursorRegion);
310       end
311     else if (PtInRgn(mousePosition, plusCursorRgn)) then
312       begin
313       SetCursor(GetCursor(plusCursor)^^);
314       CopyRgn(plusCursorRgn, cursorRegion);
315       end
316     else
317       begin
318       SetCursor(qd.arrow);
319       CopyRgn(arrowCursorRgn, cursorRegion);
320       end;
321
322     DisposeRgn(arrowCursorRgn);
323     DisposeRgn(ibeamCursorRgn);
324     DisposeRgn(crossCursorRgn);
325     DisposeRgn(plusCursorRgn);
326     end;
327       {of procedure ChangeCursor}
328
329   { ############################################################################## DoCursor }
330
331   procedure DoCursor;
332
333     var
334     cursorRect : Rect;
335     a : integer;
336
337     begin
338     BackColor(whiteColor);
339     FillRect(gWindowPtr^.portRect, qd.white);
340
341     cursorRect := gWindowPtr^.portRect;
342     PenPat(qd.gray);
343     PenSize(1, 1);
344     ForeColor(redColor);
345
346     for a := 0 to 2 do
347       begin
348       InsetRect(cursorRect, 40, 40);
349       FrameRect(cursorRect);
350       end;
351
352     MoveTo(10, 20);
353     DrawString('Arrow cursor region');
354     MoveTo(50, 60);
355     DrawString('IBeam cursor region');
356     MoveTo(90, 100);
357     DrawString('Cross cursor region');
358     MoveTo(130, 140);
```

```
  DrawString('Plus cursor region');

  gCursorRegionsActive := true;
  gCursorRegion := NewRgn;
  end;
    {of procedure DoCursor}

{ ############################################################################## DoPicture }

procedure DoPicture;

  var
  pictureRect : Rect;
  picParams : OpenCPicParams;
  pictureHdl : PicHandle;
  trianglePoly : PolyHandle;
  pictureInfo : PictInfo;
  pictInfoString : string;
  ignored : OSErr;

  begin
  BackColor(whiteColor);
  FillRect(gWindowPtr^.portRect, qd.white);

  pictureRect := gWindowPtr^.portRect;
  InsetRect(pictureRect, 50, 50);

  picParams.srcRect := pictureRect;
  picParams.hRes := $00480000;
  picParams.vRes := $00480000;
  picParams.version := -2;

  pictureHdl := OpenCPicture(picParams);

  ClipRect(gWindowPtr^.portRect);

  ForeColor(blueColor);
  FillRect(pictureRect, qd.dkGray);
  ForeColor(yellowColor);
  FillOval(pictureRect, qd.gray);

  trianglePoly := OpenPoly;
  MoveTo(pictureRect.left, pictureRect.bottom);
  LineTo(trunc(pictureRect.left + ((pictureRect.right - pictureRect.left) / 2)),
           pictureRect.top);
  LineTo(pictureRect.right, pictureRect.bottom);
  ClosePoly;

  PenPat(qd.black);
  ForeColor(redColor);
  PaintPoly(trianglePoly);
  KillPoly(trianglePoly);

  ForeColor(blackColor);
  TextSize(30);
  TextFont(systemFont);
  MoveTo(115, 230);
  DrawString('Recorded Picture');
  ForeColor(whiteColor);
  MoveTo(112, 227);
  DrawString('Recorded Picture');

  ClosePicture;

  DrawPicture(pictureHdl, pictureRect);

  SetWTitle(gWindowPtr, 'Click Mouse for Picture Information');

  while not (Button) do ;

  FillRect(gWindowPtr^.portRect, qd.white);
  SetWTitle(gWindowPtr, 'Offscreen Graphics Worlds, Pictures and Cursors');

  TextFont(1);
  TextSize(10);

  ignored := GetPictInfo(pictureHdl, pictureInfo, returnPalette, 8, systemMethod, 0);
```

```
436     ForeColor(blackColor);
437     MoveTo(180, 50);
438     DrawString('Some Picture Information:');
439
440     MoveTo(180, 80);
441     DrawString('TextStrings: ');
442     NumToString(pictureInfo.textCount, pictInfoString);
443     DrawString(pictInfoString);
444
445     MoveTo(180, 95);
446     DrawString('Rectangles: ');
447     NumToString(pictureInfo.rectCount, pictInfoString);
448     DrawString(pictInfoString);
449
450     MoveTo(180, 110);
451     DrawString('Round Rectangles: ');
452     NumToString(pictureInfo.rRectCount, pictInfoString);
453     DrawString(pictInfoString);
454
455     MoveTo(180, 125);
456     DrawString('Ovals: ');
457     NumToString(pictureInfo.ovalCount, pictInfoString);
458     DrawString(pictInfoString);
459
460     MoveTo(180, 140);
461     DrawString('Arcs: ');
462     NumToString(pictureInfo.arcCount, pictInfoString);
463     DrawString(pictInfoString);
464
465     MoveTo(180, 155);
466     DrawString('Polygons: ');
467     NumToString(pictureInfo.polyCount, pictInfoString);
468     DrawString(pictInfoString);
469
470     MoveTo(180, 170);
471     DrawString('Unique Fonts: ');
472     NumToString(pictureInfo.uniqueFonts, pictInfoString);
473     DrawString(pictInfoString);
474
475     KillPicture(pictureHdl);
476
477     TextFont(1);
478     TextSize(10);
479
480     end;
481       {of procedure DoPicture}
482
483   { ############################################################################ DoGWorldDrawing }
484
485   procedure DoGWorldDrawing;
486
487     var
488     a, b, c, i, j : integer;
489     theRect : Rect;
490
491     begin
492     PenPat(qd.black);
493     PenSize(1, 1);
494
495     for a := 0 to 7 do
496       for i := 0 to 15 do
497         begin
498         b := i * 30 + 12;
499         for j := 0 to 15 do
500           begin
501           c := j * 18 + 5;
502           SetRect(theRect, b+a, c+a, b+28-a, c+16-a);
503           if (a < 3)
504             then  ForeColor(redColor)
505             else  if ((a > 2) and (a < 6))
506                   then  ForeColor(greenColor)
507                   else  if(a > 5) then
508                         ForeColor(blueColor);
509           FrameRect(theRect);
510           end;
511             {of j-for loop}
512         end;
```

```
513           {of i-for loop}
514     end;
515       {of procedure DoGWorldDrawing}
516
517   { ############################################################ DoWithoutOffScreenGWorld }
518
519   procedure DoWithoutOffScreenGWorld;
520
521     begin
522     BackColor(whiteColor);
523     FillRect(gWindowPtr^.portRect, qd.white);
524
525     DoGWorldDrawing;
526     end;
527       {of procedure DoWithoutOffScreenGWorld}
528
529   { ############################################################## DoWithOffScreenGWorld }
530
531   procedure DoWithOffScreenGWorld;
532
533     var
534     windowPortPtr : CGrafPtr;
535     deviceHdl : GDHandle;
536     quickDrawErr : QDErr;
537     gworldPortPtr : GWorldPtr;
538     gworldPixMapHdl : PixMapHandle;
539     lockPixResult : boolean;
540     sourceRect, destRect : Rect;
541
542     begin
543     BackColor(whiteColor);
544     FillRect(gWindowPtr^.portRect, qd.white);
545
546     ForeColor(blackColor);
547     MoveTo(130, 140);
548     DrawString('Please Wait.  Drawing in offscreen graphics port.');
549
550     SetCursor(GetCursor(watchCursor)^^);
551
552     GetGWorld(windowPortPtr, deviceHdl);
553
554     quickDrawErr := NewGWorld(gworldPortPtr, 0, gWindowPtr^.portRect, nil, nil, 0);
555     if ((gworldPortPtr = nil) or (quickDrawErr <> noErr)) then
556       begin
557       SysBeep(10);
558       Exit(DoWithOffScreenGWorld);
559       end;
560
561     SetGWorld(gworldPortPtr, nil);
562
563     gworldPixMapHdl := GetGWorldPixMap(gworldPortPtr);
564
565     lockPixResult := LockPixels(gworldPixMapHdl);
566     if not (lockPixResult) then
567       begin
568       SysBeep(10);
569       Exit(DoWithOffScreenGWorld);
570       end;
571
572     EraseRect(gworldPortPtr^.portRect);
573
574     DoGWorldDrawing;
575
576     SetGWorld(windowPortPtr, deviceHdl);
577
578     sourceRect := gworldPortPtr^.portRect;
579     destRect := windowPortPtr^.portRect;
580
581     CopyBits(GrafPtr(gworldPortPtr)^.portBits, GrafPtr(windowPortPtr)^.portBits,
582              sourceRect, destRect, srcCopy, nil);
583
584     if (QDError <> noErr) then
585       SysBeep(10);
586
587     UnlockPixels(gworldPixMapHdl);
588     DisposeGWorld(gworldPortPtr);
589
```

```
  SetCursor(qd.arrow);

  end;
    {of procedure DoWithOffScreenGWorld}

{ ############################################################################## DoIdle }

procedure DoIdle;

  begin
  if (gAnimCursActive = true) then
    SpinAnimCursor;
  end;
    {of procedure DoIdle}

{ ######################################################################## DoDemonstrationMenu }

procedure DoDemonstrationMenu(menuItem : integer);

  begin
  case (menuItem) of

    iWithoutOffScreenGWorld:
      begin
      DoWithoutOffScreenGWorld;
      end;

    iWithOffScreenGWorld:
      begin
      DoWithOffScreenGWorld;
      end;

    iPicture:
      begin
      DoPicture;
      end;

    iCursor:
      begin
      DoCursor;
      end;

    iAnimatedCursor:
      begin
      DoAnimCursor;
      end;

    iIcon:
      begin
      DoIcon;
      end;
    end;
      {of case statement}
  end;
    {of procedure DoDemonstrationMenu}

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

procedure DoMenuChoice(menuChoice : longint);

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

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

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

  if (gAnimCursActive = true) then
    begin
    gAnimCursActive := false;
    SetCursor(qd.arrow);
```

```
667       ReleaseAnimCursor;
668       gSleepTime := kMaxLong;
669       end;
670
671     if (gCursorRegionsActive = true) then
672       begin
673       gCursorRegionsActive := false;
674       DisposeRgn(gCursorRegion);
675       gCursorRegion := nil;
676       end;
677
678     case (menuID) of
679
680       mApple:
681         begin
682         if (menuItem = iAbout)
683           then  ignored := Alert(rAlert, nil)
684           else  begin
685               GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
686               daDriverRefNum := OpenDeskAcc(itemName);
687               end;
688         end;
689
690       mFile:
691         begin
692         if (menuItem = iQuit) then
693           gDone := true;
694         end;
695
696       mDemonstration:
697         begin
698         DoDemonstrationMenu(menuItem);
699         end;
700       end;
701         {of case statement}
702
703     HiliteMenu(0);
704     end;
705       {of procedure DoMenuChoice}
706
707   { ############################################################################## DoOSEvent }
708
709   procedure DoOSEvent(var eventRec : EventRecord);
710
711     begin
712     case (BAnd(BSR(eventRec.message, 24), $000000FF)) of
713
714       suspendResumeMessage:
715         begin
716         if (BAnd(eventRec.message, resumeFlag) = 1)
717           then gInBackground := false
718           else gInBackground := true;
719         end;
720
721       mouseMovedMessage:
722         begin
723         if (gCursorRegionsActive) then
724           ChangeCursor(gWindowPtr, gCursorRegion);
725         end;
726       end;
727         {of case statement}
728     end;
729       {of procedure DoOSEvent}
730
731   { ############################################################################ DoMouseDown }
732
733   procedure DoMouseDown(var eventRec : EventRecord);
734
735     var
736     theWindowPtr : WindowPtr;
737     partCode : integer;
738
739     begin
740     partCode := FindWindow(eventRec.where, theWindowPtr);
741
742     case (partCode) of
743
```

```
744       inMenuBar:
745         begin
746         DoMenuChoice(MenuSelect(eventRec.where));
747         end;
748
749       inSysWindow:
750         begin
751         SystemClick(eventRec, theWindowPtr);
752         end;
753
754       inContent:
755         begin
756         if (theWindowPtr <> FrontWindow) then
757           SelectWindow(theWindowPtr);
758         end;
759
760       inDrag:
761         begin
762         DragWindow(theWindowPtr, eventRec.where, qd.screenBits.bounds);
763         end;
764
765       inGoAway:
766         begin
767         if (TrackGoAway(theWindowPtr, eventRec.where)) then
768           gDone := true;
769         end;
770       end;
771         {of case statement}
772     end;
773       {of procedure DoMouseDown}
774
775   { ############################################################################## DoEvents }
776
777   procedure DoEvents(var eventRec : EventRecord);
778
779     var
780     theWindowPtr : WindowPtr;
781     charCode : char;
782
783     begin
784     theWindowPtr := WindowPtr(eventRec.message);
785
786     case (eventRec.what) of
787
788       mouseDown:
789         begin
790         DoMouseDown(eventRec);
791         end;
792
793       keyDown, autoKey:
794         begin
795         charCode := chr(BAnd(eventRec.message, charCodeMask));
796         if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
797           DoMenuChoice(MenuKey(charCode));
798         end;
799
800       updateEvt:
801         begin
802         BeginUpdate(theWindowPtr);
803         EndUpdate(theWindowPtr);
804         end;
805
806       osEvt:
807         begin
808         DoOSEvent(eventRec);
809         end;
810
811       end;
812         {of case statement}
813     end;
814       {of procedure DoEvents}
815
816   { ############################################################################## EventLoop }
817
818   procedure EventLoop;
819
820     var
```

```
821     eventRec : EventRecord;
822     gotEvent : boolean;
823
824     begin
825     gDone := false;
826     gSleepTime := kMaxLong;
827     gCursorRegion := nil;
828
829     while not (gDone) do
830       begin
831       gotEvent := WaitNextEvent(everyEvent, eventRec, gSleepTime, gCursorRegion);
832       if (gotEvent)
833         then DoEvents(eventRec)
834         else DoIdle;
835       end;
836     end;
837       {of procedure EventLoop}
838
839   { ################################################################ start of main program }
840
841   begin
842
843   gCursorRegionsActive := false;
844   gAnimCursActive := false;
845
846     { ……… initialise managers }
847
848     DoInitManagers;
849
850     { ……… set up menu bar and menus }
851
852     menubarHdl := GetNewMBar(rMenubar);
853     if (menubarHdl = nil) then
854       ExitToShell;
855     SetMenuBar(menubarHdl);
856     DrawMenuBar;
857     menuHdl := GetMenuHandle(mApple);
858     if (menuHdl = nil)
859       then ExitToShell
860       else AppendResMenu(menuHdl, 'DRVR');
861
862     { ……… open window }
863
864     gWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
865     if (gWindowPtr = nil) then
866       ExitToShell;
867
868     SetPort(gWindowPtr);
869     TextSize(10);
870
871     { ……… enter event loop }
872
873     EventLoop;
874
875   end.
876     {of main program}
877
878   { ############################################################################################ }
```

Demonstration Program Comments

When this program is run, the user should:

- Invoke the demonstrations by choosing items from the Demonstration menu and the About GWorldPictCursIcon…item in the Apple menu.

- Note that both the About GWorldPicCursIcon… item in the Apple menu and the Icons item in the Demonstration menu contain icons.

  The resource ID for the 'SICN', 'ICON', and 'cicn' resources associated with these menu items is 257.

  $1E is specified in the keyboard equivalent field of the menu item definition for the About GWorldPicCursIcon… item. This means that the 'SICN' resource with ID 257 will be

displayed on black-and-white Macintoshes, and the 'cicn' resource with the same ID, scaled down to 16-by-16 pixels, will be displayed on Macintoshes with Color QuickDraw.

$36 (the ASCII character code for 6) is specified in the keyboard equivalent field of the menu item definition for the Icon item. This means that the Menu Manager will automatically enlarge the the menu item's enclosing rectangle to accommodate the 32-by-32 pixel colour icon, that the 'ICON' resource with ID 257 will be displayed on black-and-white Macintoshes, and that the 'cicn' resource with the same ID will be displayed on Macintoshes with Color QuickDraw. It also means that the Command-key equivalent will appear in the menu item along with the icon.

If the display device in a Color Quickdraw environment is set to pixel depths of 1 or 2, the bitmap (black-and-white) component of the colour icon resource will be displayed.

- Click outside and inside the window when the cursor and animated cursor demonstrations have been invoked.The constant declaration block

Lines 50-60 establish constants related to menu IDs and menu item numbers. Lines 62-66 establish constants related to alert, menu bar, window, cursor, and icon resources. Line 68 establishes a constant for the interval between frame changes for an animated cursor. Line 70 sets kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The type declaration block

Lines 76-82 define a data type which is identical to the structure of an 'acur' resource.

The variable declaration block

gDone controls exit from the main event loop and thus program termination. gWindowPtr will be assigned the pointer to the window utilised by the demonstration.

In this program, the sleep and cursor region parameters in the WaitNextEvent call will be changed during program execution. Hence the global variables gSleepTime and gCursorRegion.

gInBackground relates to foreground/background switching.

gCursorRegionActive and gAnimCursActive will be set to true during, respectively, the cursor and animated cursor demonstrations. gAnimCursHdl will be assigned a handle to the animCurs structure used during the animated cursor demonstration. gAnimCursTickInterval and gAnimCursLastTick also relate to the animated cursor demonstration.

The procedure DoIcon

DoIcon draws an icon in the window at a size and location determined by a bounding rectangle.

Lines 134-135 clear the port rectangle to white. Line 137 sets the initial coordinates of the top, left, bottom and right of the bounding rectangle.

Line 139 tests for the presence of Color QuickDraw. If Color QuickDraw is not present, Lines 141-150 execute. The call to GetIcon reads the specified 'ICON' resource from disk and returns a handle to a 128-byte bit image of the icon. Lines 143-149 use PlotIcon to plot the icon a number of times, with the location, size and shape of the icon being changed each time through the loop.

If Color QuickDraw is present, Lines 151-161 execute. The call to GetCIcon at Line 152 obtains a CIcon data structure and initialises it with data from the specified 'cicn' resource. Lines 153-159 use PlotCIcon to plot the icon a number of times, with the location, size and shape of the icon being changed each time through the loop.

Line 160 removes all data structures created by the call to GetCIcon. This is important because GetCIcon creates a new Cicon data structure each time it is called.

The procedure ReleaseAnimCursor

ReleaseAnimCursor deallocates the memory occupied by the Cursor structures (Lines 173-174) and the 'acur' resource (Line 176).

Recall that releaseAnimCursor is called when the user clicks in the menu bar and that, at the same time, the gAnimCursActive flag is set to false, the cursor is reset to the standard arrow shape, and WaitNextEvent's sleep parameter is reset to the maximum possible value.

The procedure SpinAnimCursor

SpinAnimCursor is called whenever null events are received (that is, in this demonstration, every 5 ticks assuming no other events intervene).

Line 188 assigns the number of ticks since system startup to newTick. Line 189 checks whether 5 ticks have elapsed since Line 230 was executed (first call to SpinAnimCursor) or since SpinAnimCursor last exited (subsequent calls to SpinAnimCursor - see Line 197). If 5 ticks have not elapsed, the function simply returns (Line 190). Otherwise, Line 192 sets the cursor shape to that represented by the handle stored in the specified element of the frame[] field of the animCurs structure. Line 193 increments the frame counter field (whichFrame) of the animCurs structure. If Line 192 set the cursor to the last cursor in the series (Line 194), Line 195 resets the frame counter to 0. Line 197 retrieves and stores the tick count at exit for use at Line 189 next time the function is called.

The function GetAnimCursor

GetAnimatedCursor retrieves the data in the specified 'acur' resource and stores it in an animCurs structure, retrieves the 'CURS' resources specified in the 'acur' resource and assigns the handles to the resulting Cursor structures to elements in an array in the animCurs structure, establishes the frame rate for the cursor, and sets the starting frame number.

Line 213 calls GetResource to read the 'acur' resource into memory and return a handle to the resource. The handle is cast to type animCursHandle and assigned to the global variable gAnimCursHdl (a handle to a structure of type animCurs, which is identical to the structure of an 'acur' resource). If this call is not successful (that is, GetResource returns NIL), the function will simply exit, returning false to DoAnimCursor. If the call is successful, noError is set to true (Line 216) before Line 217 sets up a loop which will cycle once for each of the 'CURS' resources specified in the 'acur' resource - assuming that noError is not set to false at some time during this process.

The ID of each cursor is stored in the high word of the specified element of the frame[] field of the animCurs structure, and this is retrieved at Line 219. The cursor ID is then used in the call to GetCursor at Line 220 to read in the resource from disk (if necessary) and assign the handle to the resulting 68-byte Cursor structure to the specified element of the frame[] field of the animCurs structure. If this pass through the loop was successful, the array index is incremented (Lines 221-222); otherwise, noError is set to false (Line 223), causing the loop and the function to exit, returning false to DoAnimCursor.

Line 229 assigns the ticks value passed to getAnimCursor to a global variable which will be utilised in the function SpinCursor. Line 230 assigns the number of ticks since system startup to another variable which will also be utilised in the function SpinAnimCursor. Line 231 sets the starting frame number.

At this stage, the animated cursor has been initialised and DoIdle will call SpinAnimCursor whenever null events are received.

The procedure DoAnimCursor

DoAnimCursor responds to the user's selection of the Animated Cursor item from the Demonstration menu.

In this demonstration, application-defined functions are utilised to retrieve 'acur' and 'CURS' resources, spin the cursor, and deallocate the memory associated with the animated cursor when the cursor is no longer required. These functions are generic in that they may be used to initialise, spin and release any animated cursor passed to the getAnimCursor function as a formal parameter. A "beach-ball" cursor is utilised in this demonstration. doAnimCursor's major role is simply to call GetAnimCursor with the beach-ball 'acur' resource as a parameter.

Lines 246-247 clear the window to white. Line 249 assigns the resource ID of the beach-ball 'acur' resource to the variable used as the first parameter in the GetAnimCursor call at Line 252. Line 250 assigns a value represented by a constant to the second parameter in the GetAnimCursor call. This value controls the frame rate of the cursor, that is, the number of ticks which must elapse before the next frame (cursor) is displayed. (The best frame rate depends on the type of animated cursor used.)

Line 252 calls the GetAnimCursor function. If the call is successful, the flag gAnimCursActive is set to true (Line 254) and, importantly, the sleep parameter in the WaitNextEvent call is set to the same ticks value as that used to control the cursor's frame rate (Line 255). This latter will cause null events to be generated at that tick interval (assuming, of course, that no other events intervene). Recall that the DoIdle function is called whenever a null event is received and that, if the flag gAnimCursActive is set to true, DoIdle calls the SpinAnimCursor function.

If the call to GetAnimCursor fails, DoAnimCursor simply plays the system alert sound and returns (Lines 257).

COLOUR ANIMATED CURSOR

For a colour animated cursor:

- Replace the 'CURS' resource with a 'crsr' resource.
- Replace Line 79 with:

```
frame : array [0..8] of CCursHandle;
```

- Replace Line 192 with:

```
SetCCursor (gAnimCursHdl^^.frame[gAnimCursHdl^^.whichFrame]^^);
```

- Replace Line 220 with:

```
gAnimCursHdl^^.frame[a] := GetCCursor(cursorID);
```

- Replace Line 174 with:

```
DisposeCIcon (gAnimCursHdl^^.frame[a]^^);
```

The procedure ChangeCursor

ChangeCursor is called whenever a mouse-moved message is reported (see Lines 720-724). Recall that mouse-moved messages are generated only when the mouse is not within the region specified in the last parameter to the WaitNextEvent call.

Lines 274-277 create new empty regions to serve as the regions within which the cursor shape will be changed to, respectively, the system arrow, the system I-beam, the system cross, and the system plus.

Line 279 sets the arrow cursor region to, initially, the boundaries of the coordinate plane. Lines 281-283 establish a rectangle equivalent to the window's port rectangle and change this rectangle's coordinates from local to global coordinates. Line 285 insets this rectangle by 40 pixels all round and Line 286 establishes this as the I-beam region. Line 287, in effect, cuts the rectangle represented by the I-beam region from the arrow region, leaving a hollow arrow region.

Lines 289-295 use the same procedure to establish a rectangular hollow region for the cross cursor and an interior rectangular region for the plus cursor. The result of all this is a rectangular plus cursor region in the centre of the window, surrounded by (but not overlapped by) a hollow rectangular cross cursor region, this surrounded by (but not overlapped by) a hollow rectangular I-beam cursor region, this surrounded by (but not overlapped by) a hollow rectangular arrow cursor region the outside of which equates to the boundaries of the coordinate plane.

Line 297 gets the point representing the mouse's current position. Since GetMouse returns this point in local coordinates, Line 298 converts it to global coordinates.

The next task is to determine the region in which the cursor is currently located (its movement to that region having generated by the mouse-moved event which resulted in the call to this function in the first place). The calls to PtInRgn at Lines 300, 305 and 310 are made for that purpose. Depending on which region is established as the region in which the cursor in currently located, the cursor is set to the appropriate shape and that region is assigned to WaitNextEvent's mouseRgn parameter. This latter means that, since the cursor is now within the region assigned to the mouseRgn parameter, mouse-moved events will cease to be generated until the mouse is moved out of that region.

That accomplished, Lines 321-324 deallocate the memory associated with the regions created earlier in the function.

The procedure DoCursor

DoCursor is called when the user selects Cursors from the Demonstration menu. Its chief purpose is to assign true to the global variable gCursorRegionActive, which will cause a mouse-moved message to result in a call to ChangeCursor (see Lines 720-724). In addition, it draws some rectangles in the window which visually represent to the user some cursor regions which will later be established by the changeCursor function.

Lines 337-338 clear the port rectangle to white. Lines 340-358 draw the rectangles and descriptive text in the window.

Line 360 sets the gCursorRegionsActive flag to true and Line 361 creates an empty region for the last parameter of the WaitNextEvent call.

The procedure DoPicture

DoPicture demonstrates recording and playing back a picture.

Lines 379-383 clear the window to white and establish a rectangle 50 pixels inside the port rectangle. Lines 385-388 assign values to the fields of an OpenCPicParams record. These specify the rectangle established at Line 382, 72 pixels per inch resolution horizontally, and 72 pixels per inch resolution vertically. The version field should always be set to -2. Using this record as its parameter, OpenCPicture initiates the recording of the picture definition (Line 390).

Line 392 establishes the clipping region as equivalent to the port rectangle. (Before this call, the clipping region is very large. In fact, it is as large as the coordinate plane. If the clipping region is very large and you scale the picture while drawing it, the clipping region can become invalid when DrawPicture scales the clipping region - in which case the picture will not be drawn.)

Lines 394-418 "draw" a simple picture comprising a rectangle, an oval, a triangle and some text. (Because of the previous call to OpenCPicture, these drawing instructions are simply "recorded" in the Picture record. Nothing appears in the window.)

Line 420 terminates picture recording and Line 422 draws the picture by "playing back" the "recording" stored in the specified Picture structure.

When the user responds to the invitation to click the mouse (Lines 424-426), Line 434 returns information about the picture in a picture information record. Lines 435-472 extract some of the information from this record and print it in the window.

Line 474 deallocates the memory associated with the picture record.

The procedure DoGWorldDrawing

DoGWorldDrawing is called by both DoWithoutOffScreenWorld and DoWithOffScreenWorld to draw some graphics.

The procedure DoWithoutOffScreenGWorld

DoWithoutOffScreenGWorld is the first demonstration. It is included only as a contrast to the offscreen graphics world demonstration DoWithOffScreenWorld. It simply fills the window's port rectangle with white pixels and then calls DoGWorldDrawing to execute some drawing designed to take a short but nonetheless perceptible period of time.

The procedure DoWithOffScreenGWorld

DoWithOffScreenGWorld demonstrates the use of an offscreen graphics world to execute the same drawing operation as does DoWithoutOffScreenWorld.

At Lines 542-547, the window's port rectangle is cleared to white and some advisory text is drawn in the window indicating that drawing is taking place in an offscreen graphics world. To further indicate to the user that the application has not just drifted away, Line 549 sets the cursor to the system's familiar watch cursor.

Line 551 saves the current graphics world, that is, the current graphics port and the current device.

Line 553 creates an offscreen graphics world. The gworldPortPtr parameter receives a pointer to the offscreen graphics world's graphics port. 0 in the second parameter means that the offscreen world's pixel depth will be set to the deepest device intersecting the rectangle passed as the third parameter. The third parameter becomes the offscreen port's portRect, the offscreen pixel map's bounds and the offscreen device's gdRect value. NIL in the fourth parameter causes the default colour table for the pixel depth to be used. The fifth parameter is set to NIL because the noNewDevice flag is not set. 0 in the sixth parameter means that, in fact, no flags are set.

Line 560 sets the graphics port pointed to by gworldPortPtr as the current graphics port. (When the first parameter is a GWorldPtr, the current device is set to the device attached to the offscreen world and the second paramter is ignored.)

Lines 562-564 reflect the requirement to call LockPixels to prevent the base address of an offscreen pixel image from being moved when it is drawn into or copied from. Line 562 gets a handle to the offscreen world's pixel map and Line 564 locks that buffer in memory.

Line 571 clears the offscreen graphics port before Line 573 calls the application-defined function doGWorldDrawing to draw some graphics in the offscreen port.

Line 575 sets the window's graphics port as the current port and sets the current device to that saved at Line 409.

Lines 577-578 establish the source and destination rectangles (required by the CopyBits call at Line 580) as equivalent to the offscreen graphics world and window port rectangles respectively.

The CopyBits call at Line 580 copies the image from the offscreen world to the window. (Note that, because a basic, rather than a colour, graphics port is being drawn to, there is no need to set the foreground colour to black and the background colour to white before the CopyBits call.) Line 583 checks for any error resulting from the last QuickDraw call (in this case, CopyBits).

Line 586 unlocks the offscreen pixel image buffer and Line 587 deallocates all of the memory previously allocated for the offscreen graphics world.

Finally, Line 589 sets the cursor back to the standard arrow cursor.

The procedure DoIdle

DoIdle is called from the main event loop when a null event is received. If the active demonstration is the animated cursor demonstration (Line 599), the application defined function SpinAnimCursor is called (Line 600).

The procedure DoDemonstrationMenu

DoDemonstrationMenu handles choices from the Demonstration menu.

The procedure DoMenuChoice

DoMenuChoice processes Apple and File menu choices to completion and calls a subsidiary function to handle Demonstration menu choices.

Lines 662-675 are invoked if the user chooses a menu item while either the animated cursor demonstration or the normal cursor demonstration is the active demonstration. In these cases:

- If the animated cursor demonstration is currently the active demonstration (Line 662), the flag which indicates this condition is set to false (Line 664), the cursor is set to the standard arrow cursor (Line 665), memory associated with the animated cursor is deallocated (Line 666) and WaitNextEvent's sleep parameter is set to the maximum possible value (Line 667).

- If the normal cursor demonstration is currently the active demonstration (Line 670), the flag which indicates this condition is set to false (Line 672), the cursor region associated with the last parameter of the WaitNextEvent call is disposed of (Line 673) and that parameter is set to NIL (Line 674) to defeat mouse-moved event reporting.

If the user chooses the About… item in the Apple menu, an alert box is invoked (Lines 681-682). (Note that the Icon item in the alert box's 'DITL' resource specifies the icon resource with ID 257.)

The procedure DoOSEvents

DoOSEvents handles Operating System events.

In the event of a mouse-moved event (Line 720), and if the current demonstration is the standard cursors demonstration (Line 722), the application-defined function ChangeCursor is called (Line 723). The function is passed the pointer to the window and a pointer to the region used as the last parameter in the WaitNextEvent call.

(As an aside, note that this cursor shape adjustment strategy differs from that used in the demonstration program at Chapter 2 - Low Level and Operating System events, where the cursor adjustment function was called immediately before the WaitNextEvent call in the main event loop (provided a mouse-moved event had occurred). If the strategy shown in this program is used (that is, call the cursor adjustment function when a mouse-moved event is received), you must also call the cursor adjustment function when a new window is opened and whenever a window activation event is received.)

The procedures DoMouseDown, DoEvents

DoEvents and DoMouseDown perform minimal initial event handling consistent with the satisfactory execution of the demonstration aspects of the program.

The procedure EventLoop

EventLoop contains the main event loop. The event loop terminates when gDone is set to true.

Before the loop is entered, gSleepTime is set to kMaxLong and gCursorRegion is set to NIL (Lines 825-826). Initially, therefore:

- The sleep parameter in the WaitNextEvent call at Line 830 will be set to the maximum possible value, meaning that null events will virtually never occur.
- The mouseRegion parameter in the WaitNextEvent call will cause mouse-moved events not to occur.

Note that, if a null event is received (Line 833), the application-defined function DoIdle is called. (As will be seen, null events will occur every five ticks during the animated cursor demonstration, when WaitNextEvent's sleep parameter will be assigned the constant defined at Line 68.)

The main program block

The main function initialises the system software managers (Line 847), sets up the menus (Lines 851-859), opens a window (Line 863), sets the window's graphics port as the current port for drawing (Line 867) and sets the text size to 10 points (Line 868). The main event loop is then entered (Line 872).

Note that error handling here and in other areas of the program is somewhat rudimentary: the program simply terminates.

Creating Cursor and Icon Resources, and Assigning Icons to Menu Items, Using ResEdit

Creating Cursor and Icon Resources

Creating the 'acur' Resource

The procedure for creating the 'acur' resource is as follows:

- Open GWorldPicCursIcon.µ.rsrc in ResEdit. Choose **Resource/Create New Resource**. A small dialog opens. Click the **acur** item in the scrolling list, and then click the dialog's **OK** button. The **acurs from GWorldPicCursIcon.µ.rsrc** window opens, followed by the **acur ID = 128 from GWorldPicCursIcon.µ.rsrc** window. (ResEdit automatically assigns 128 as the resource ID of the first 'acur' resource you create.)
- Choose **Resource/GetResource Info**. In the **Info for acur = 128 from GWorldPicCursIcon.u.rsrc** window, check the **Purgeable** checkbox. Close the window.
- Enter 8 in the Number of "frames" (cursors) item.
- Enter the 'CURS' resource IDs by successively clicking on the next) ***** item, choosing **Resource/Insert New Field(s)**, and entering the appropriate 'CURS' resource ID in the resulting 'CURS' Resource Id item.
- Close the **acur ID = 128 from GWorldPicCursIcon.u.rsrc** window. Close the **acurs from GWorldPicCursIcon.u.rsrc** window. An acur icon representing the resource just created appears in the **GWorldPicCursIcon.µ.rsrc** window.

Creating the 'CURS' Resources

The procedure for creating the 'CURS' resources is as follows:

- Choose **Resource/Create New Resource**, select **CURS** in the resulting dialog, and click the **OK** button. The **CURSs from GWorldPicCursIcon.u.rsrc** window opens, followed by the **CURS ID = 128 from GWorldPicCursIcon.u.rsrc** window.

- Choose **Resource/GetResource Info**. In the **Info for CURS = 128 from GWorldPicCursIcon.u.rsrc** window, check the **Purgeable** checkbox. Close the window.

- Using the tools in the panel at the left of the **CURS ID = 128 from GWorldPicCursIcon.u.rsrc** window, draw the cursor image in the large centre panel. Then drag the thumbnail of this image in the small box titled Pointer into the small box titled Mask to automatically create the mask. Close the **CURS ID = 128 from GWorldPicCursIcon.u.rsrc** window. A thumbnail image of the cursor, labelled with the resource ID, appears in the **CURSs from GWorldPicCursIcon.u.rsrc** window.

- Choose **Resource/Create New Resource** again. **The CURS ID = 129 from GWorldPicCursIcon.u.rsrc** window opens. Repeat the previous process to create the second 'CURS' resource.

- Create the remaining six 'CURS' resources in the same way. Then close the **CURSs from GWorldPicCursIcon.u.rsrc** window. A CURS icon representing the resources just created appears in the **GWorldPicCursIcon.µ.rsrc** window.

Creating the 'cicn' Resource

The procedure for creating the 'cicn' resource is much the same as for the 'CURS' resources except that:

- **cicn** should be selected in the **Resource/Create New Resource** dialog.

- When the **cicn ID = 128 from GWorldPicCursIcon.u.rsrc** window opens, choose **cicn/Icon Size...** and enter the required width and height of the colour icon in the resulting dialog.

- While the **Info for CURS = 128 from GWorldPicCursIcon.u.rsrc** window is open, change the resource's ID to 257 as well as checking the **Purgeable** checkbox. (When the window is closed, the **cicn ID = 128 from GWorldPicCursIcon.u.rsrc** window becomes the **cicn ID = 257 from GWorldPicCursIcon.u.rsrc** window.)

- After drawing the image, drag the thumbnail of the completed image in the small box titled Color to both the B & W and Mask boxes to automatically create the bitmap version and the mask.

Creating the 'ICON' Resource

The procedure for creating the 'ICON' resource is much the same as for the 'cicn' resource except that **ICON** is selected in the **Resource/Create New Resource** dialog and no mask creation is required.

Creating the 'SICN' Resource

The procedure for creating the 'SICN' resource is much the same as for the 'ICON' resource except that **SICN** is selected in the **Resource/Create New Resource** dialog.

Assigning Icons to Menu Items

About GWorldPicCursIcon... Menu Item

The procedure for assigning the small icon ('SICN') to the **About GWorldPicCursIcon...** menu item in the Apple menu is as follows:

- In the **GWorldPicCursIcon.µ.rsrc** window, double-click the MENU icon. The **MENUs From GWorldPicCursIcon.µ.rsrc** window opens. With the thumbnail of the Apple menu (ID 128) selected, choose **Resource/Open Using Hex Editor**. The **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window opens. The bottom three lines of the display are as follows:

```
000018   576F 726C 6450 6963  WorldPic
000020   4375 7273 4963 6F6C  CursIcon
000028   C900 0000 0000       …¤¤¤¤¤
```

  Note that the second and third words in the bottom row are both `00`. The second word is for the icon resource ID (if any). The third word is for the keyboard equivalent (if any). Close the window.

- In the **MENUs From GWorldPicCursIcon.µ.rsrc** window, double-click the Apple menu thumbnail. The **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window opens. Click the **About GWorldPicCursIcon…** item to highlight it and choose **MENU/Choose Icon…**. Click the **Small Icons (SICN)** radio button. The `'SICN'` resource with ID 257 appears in the list box. Click that item to highlight it, then click the **OK** button. Back in the **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window, note that the **Cmd-Key:** item is now dimmed. (A menu item that has a small icon cannot have a keyboard equivalent.)

- Close the **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window. Notice in the **MENUs From GWorldPicCursIcon.µ.rsrc** window that either the small icon (Color QuickDraw not present) or a scaled down version of the colour icon (Color QuickDraw present) appears in the **About GWorldPicCursIcon…** item in the Apple menu thumbnail.

- With the Apple menu thumbnail selected, choose **Resource/Open Using Hex Editor**. The **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window opens. The bottom three lines of the display are as now as follows:

```
000018   576F 726C 6450 6963  WorldPic
000020   4375 7273 4963 6F6C  CursIcon
000028   C901 1E00 0000       …¤¤¤¤¤
```

  Notice that the keyboard equivalent word is now `1E`, which indicates that the item has an icon defined in a `'SICN'` resource. Note also that the icon resource ID word contains `01` (257-256).[14] Close the **MENU = 128 From GWorldPicCursIcon.µ.rsrc** window.

Icon Menu Item

The procedure for assigning the colour icon (`'cicn'`) to the **Icon** menu item in the **Demonstration** menu is as follows:

- In the **MENUs From GWorldPicCursIcon.µ.rsrc** window, double-click the Demonstration menu thumbnail. The **MENU = 131 From GWorldPicCursIcon.µ.rsrc** window opens. Note that the **Cmd-Key:** item contains 6 (the keyboard equivalent).

- Click the **Icon** menu item to highlight it, and then choose **MENU/Choose Icon…**. In the resulting dialog, click the **Normal Icons (ICON)** radio button. The `'ICON'` resource with ID 257 appears in the list box. Click the icon and then click the **OK** button. Back in the **MENU = 131 From GWorldPicCursIcon.µ.rsrc** window, note that the **Cmd-Key:** item is *not* dimmed. (A menu item that has a normal icon can also have a have a keyboard equivalent.)

- Close the **MENU = 131 From GWorldPicCursIcon.µ.rsrc** window. Notice in the **MENUs From GWorldPicCursIcon.µ.rsrc** window that either the icon (Color QuickDraw not present) or the colour icon (Color QuickDraw present) appears in the **Icon** item in the **Demonstration** menu thumbnail. Note also that the item's enclosing rectangle has been expanded to accommodate the 32-by-32 pixel icon/colour icon, and that the item has a keyboard equivalent.

[14] Recall from Footnote 8 at Chapter 3 — Menus that the Menu Manager adds 256 to the resource ID specified and uses the result as the icon's rersource ID.

- With the **Demonstration** menu thumbnail selected, choose **Resource/Open Using Template**. In the resulting dialog, select **MENU** and click the **OK** button. The **MENU 131= From GWorldPicCursIcon.µ.rsrc** window opens. Scroll down to the last menu item and note the Key equiv item. This item will only accept and display a single character, which is why the Hex Editor was used to display the `0x1E` keyboard equivalent in the **About GWorldPicCursIcon…** item. (The Hex Editor can also be used to enter non-single character keyboard equivalents.) Note also the Icon # item, which contains the icon's resource ID (257-256).

- Close the **MENU 131= From GWorldPicCursIcon.µ.rsrc** window. Close the **MENUs From GWorldPicCursIcon.µ.rsrc** window. Close the **GWorldPicCursIcon.µ.rsrc** window, saving the file.