CHAPTER 8

Cursor Utilities

Contents

Cursor Utilities

This chapter describes the utilities that your application uses to draw and manipulate the cursor on the screen. You should read this chapter to find out how to implement cursors in your application. For example, you should change the arrow cursor to an I-beam cursor when it's over text and to an animated cursor when a medium-length process is under way.

Cursors are defined in resources; the routines in this chapter automatically call the Resource Manager as necessary. For more information about resources, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. Color cursors are defined in resources as well, though they use Color QuickDraw. For information about Color QuickDraw, see the chapter "Color QuickDraw" in this book.

This chapter describes how to

- create and display black-and-white and color cursors
- change the cursor's shape over different areas on the screen
- display an animated cursor

About the Cursor

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'`) resource. The cursor is an integral part of the Macintosh user interface. The user manipulates the cursor with the mouse to select objects or areas on the screen. (It appears only on the screen and never in an offscreen graphics port.) The user moves the cursor across the screen by moving the mouse. Most actions take place only when the user positions the cursor over an object on the screen, then clicks (presses and releases the mouse button). For example, a user might point at a document icon created by your application and click to select it, then choose the Open command from the File menu by pointing at it with the mouse button depressed and then releasing the mouse button.

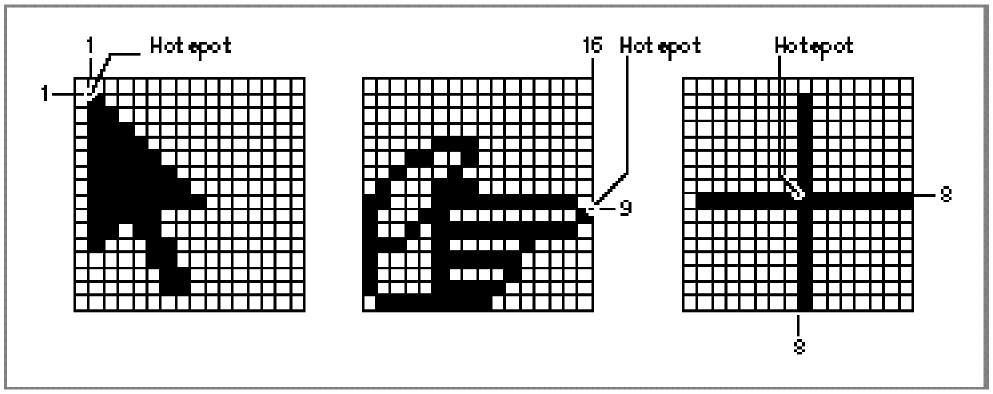
You use a cursor in the content area of your application's windows to allow the user to select all or part of the content. Your application also uses the cursor in the scroll bar area of its windows to adjust the position of the document's contents in the window area. You can change the shape of the cursor to indicate that a user is over a certain kind of content, such as text, or to provide feedback about the status of the computer system.

Note
Some Macintosh user manuals call the cursor a *pointer* because it points to a location on the screen. To avoid confusion with other meanings of *pointer, Inside Macintosh* uses the alternate term *cursor.* ◆

Basic QuickDraw supplies a predefined cursor in the global variable named `arrow`; this is the standard arrow cursor.

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 portion of the pointer that must be positioned over a screen object before mouse clicks can have an effect on that object. For example, when the user presses the mouse button, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates. Figure 8-1 illustrates three cursors and their hot spot points.

Figure 8-1 Hot spots in cursors



The hot spot is a point (not a bit) in the bit image for the cursor. Imagine the rectangle with corners (0,0) and (16,16) containing the cursor's bit image, as in each of the examples in Figure 8-1; each hot spot is defined in the local coordinate systems of these rectangles. For the arrow cursor in this figure, local coordinates (1,1) designate the hot spot. A hot spot of (8,8) is in the center of the crosshairs cursor in Figure 8-1. Notice that the hot spot for the pointing hand cursor has a horizontal coordinate of 16 and a vertical coordinate of 9.

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 doesn't need to do anything to move the cursor.

Your application should change the cursor shape depending on where the user positions it on the screen. For example, when the cursor is in your application's menu bar, the cursor should usually have an arrow shape. When the user moves the cursor over a text document, your application should change the cursor's shape to an I-beam, which indicates where the insertion point will move if the user clicks. When it's over graphic objects, the cursor may have different shapes depending on the type of graphic and the operation that the user is attempting to complete. You should change the cursor shape *only* to provide information to the user. In other words, don't change its shape randomly.

In general, you should always make the cursor visible in your application. To maintain a stable and consistent environment, the user should have access to the cursor. There are a few cases when the cursor may not be visible. For example, in an application where the user is entering text, the insertion point should blink and the cursor should not be visible. If the cursor and the insertion point were both visible, it might confuse the user about where the input would appear. Or, if the user is viewing a slide show in a presentation software application, the cursor need not be visible. However, whenever the user needs access to the cursor, a simple move of the mouse should make the cursor visible again.

When the cursor is used for choosing and selecting, it should remain black. You may want to display a color cursor when the user is drawing or typing in color. The cursor shouldn’t contain more than one color at a time, with the exception of a multicolored paintbrush cursor. It’s hard for the eye to distinguish small areas of color. Make sure that the hot spot can be seen when it’s placed on a background of a similar color. This can be accomplished by changing the color of the cursor or by adding a one-pixel outline in a contrasting color.

When your application is performing an operation that will take at least a couple of seconds, and more time than a user might expect, you need to provide feedback to the user that the operation is in progress. If the operation will last a second or two (a short operation), change the cursor to the wristwatch cursor. If the operation takes several seconds (a medium-length operation) and the user can do nothing in your application but stop the operation, wait until it is completed, or switch to another application, you need to display an animated cursor. This lets the user know that the computer system hasn’t crashed—it’s just busy. If the operation will take longer than several seconds (a lengthy operation), your application should display a status indicator to show the user the estimated total time and the elapsing time of the operation.

For more information about displaying cursors and status indicators in your application, see *Macintosh Human Interface Guidelines*.

Using the Cursor Utilities

This section describes how you can

- create cursors
- change the shape of the cursor
- animate a cursor to indicate that a medium-length process is taking place

To implement cursors, you need to

- define black-and-white cursors as 'CURS' resources in the resource file of your application
- define color cursors in 'crsr' resources—if you want to display color cursors—in the resource file of your application
- define 'acur' resources—if you want to display animated cursors—in the resource file of your application
- initialize the Cursor Utilities by using the InitCursor and InitCursorCtl procedures when your application starts up
- use the SetCursor or SetCCursor procedure to change the cursor shape as necessary
- animate the cursor by using the SpinCursor or RotateCursor procedure

You use 'CURS' resources to create black-and-white cursors for display on black-and-white and color screens. You use 'crsr' resources to create color cursors for 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.

Before using the routines that handle color cursors—namely, the GetCCursor, SetCCursor, and DisposeCCursor routines—you must test for the existence of Color QuickDraw by using the Gestalt function with the GestaltQuickDrawVersion selector. If the value returned in the response parameter is equal to or greater than the value of the constant gestalt32BitQD, then the system supports Color QuickDraw. Both basic and Color QuickDraw support all other routines described in this chapter.

Initializing the Cursor

When your application starts up, the Finder sets the cursor to a wristwatch; this indicates that an operation is in progress. When your application nears completion of its initialization tasks, it should call the InitCursor procedure to change the cursor from a wristwatch to an arrow, as shown in the application-defined procedure DoInit in Listing 8-1.

Listing 8-1 Initializing the Cursor Utilities

```
PROCEDURE DoInit;
BEGIN
   DoSetUpHeap;       {perform Memory Manager initialization here}
   InitGraf(@thePort);{initialize basic QuickDraw}
   InitFonts;         {initialize Font Manager}
   InitWindows;       {initialize Window Manager & other Toolbox }
                      { managers here}
                      {perform all other initializations here}
   InitCursor;        {set cursor to an arrow instead of a }
                      { wristwatch}
   InitCursorCtl(NIL);{load resources for animated cursor with }
                      { resource ID 0}
END; {of DoInit}
```

If your application uses an animated cursor to indicate that an operation of medium length is under way, it should also call the `InitCursorCtl` procedure to load its `'acur'` resource and associated `'CURS'` resources, as illustrated in Listing 8-1.

Changing the Appearance of the Cursor

Whenever the user moves the mouse, the mouse driver, the Event Manager, and your application are responsible for providing feedback to the user. The mouse driver performs low-level functions, such as continually polling the mouse for its location and status and maintaining the current location of the mouse in a global variable. Whenever the user moves the mouse, a low-level interrupt routine of the mouse driver moves the cursor displayed on the screen and aligns the hot spot of the cursor with the new mouse location. This section describes how to use the `GetCursor` and `SetCursor` routines to change the appearance of a black-and-white cursor when it is in different areas of the screen. (To change the cursor to a color cursor, your application must use the `GetCCursor` function, described on page 8-26, and the `SetCCursor` procedure, described on page 8-26.)

Your application is responsible for setting the initial appearance of the cursor, for restoring the cursor after the Event Manager function `WaitNextEvent` returns, and for changing the appearance of the cursor as appropriate for your application. For example, most applications set the cursor to the I-beam 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 a scroll bar of a document. 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. `WaitNextEvent` is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The mouse driver and your application control the shape and appearance of the cursor. A cursor can be any 256-pixel image, defined by a 16-by-16 pixel square. The mouse driver displays the current cursor, which your application can change by using the `SetCursor` or `SetCCursor` procedure.

Figure 8-2 shows the standard arrow cursor. You initialize the cursor to the standard arrow cursor when you use the `InitCursor` procedure, as shown in Listing 8-1. As shown in Figure 8-2, the hot spot for the arrow cursor is at location (1,1).

Figure 8-2 The standard arrow cursor

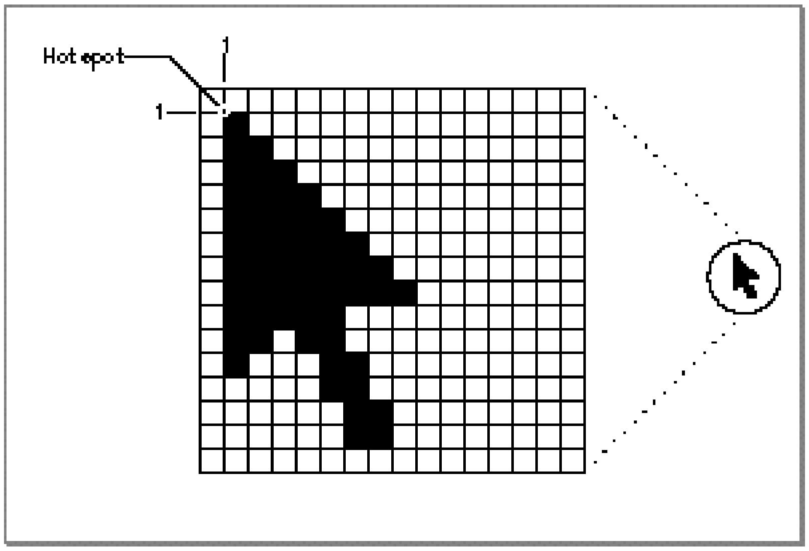
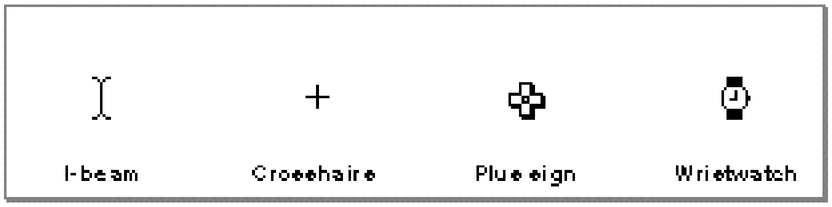


Figure 8-3 shows four other common cursors that are available to your application: the I-beam, crosshairs, plus sign, and wristwatch cursors.

Figure 8-3 The I-beam, crosshairs, plus sign, and wristwatch cursors



The I-beam, crosshairs, plus sign, and wristwatch cursors are defined as resources, and your application can get a handle to any of these cursors by specifying their corresponding resource IDs to the `GetCursor` function. These constants specify the resource IDs for these common cursors:

```
CONST iBeamCursor = 1;  {used in text editing}
      crossCursor = 2;  {often used for manipulating graphics}
      plusCursor  = 3;  {often used for selecting fields in }
                        { an array}
```

```
    watchCursor = 4;  {used when a short operation is in }
                      { progress}
```

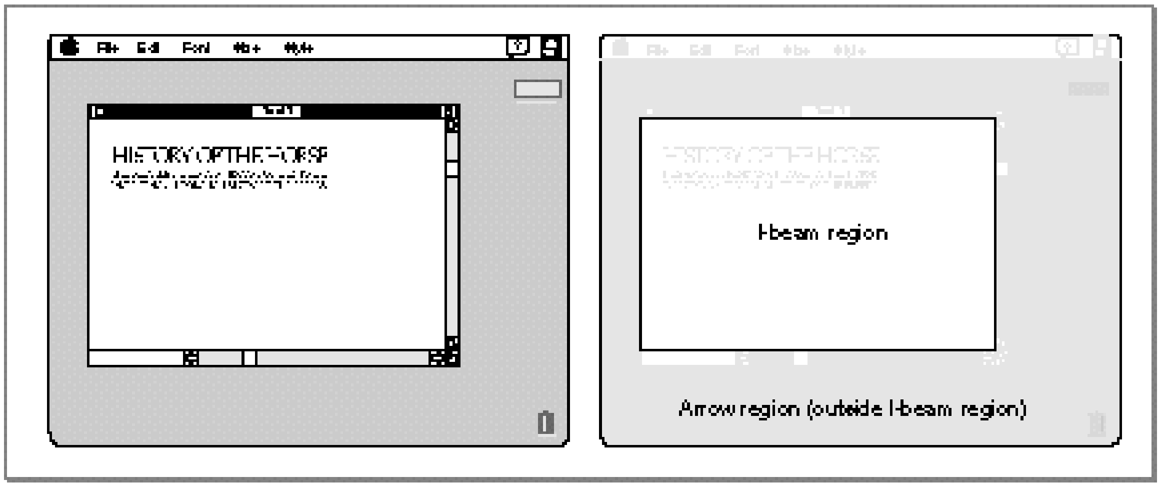
After you use the `GetCursor` function to obtain a handle to one of these cursors or to one defined by your own application in a `'CURS'` resource, you can change the appearance of the cursor by using the `SetCursor` procedure.

Your application usually needs to change the shape of the cursor as the user moves the cursor to different areas within a document. Your application can use mouse-moved events to help accomplish this. Your application also needs to adjust the cursor in response to resume events. Most applications adjust the cursor once through the event loop in response to almost all events.

You can request that the Event Manager report mouse-moved events whenever the cursor is outside of a specified region that you pass as a parameter to the `WaitNextEvent` function. (If you specify an empty region or a `NIL` handle to the `WaitNextEvent` function, `WaitNextEvent` does not report mouse-moved events.)

If you specify a nonempty region in the `mouseRgn` parameter to the `WaitNextEvent` function, `WaitNextEvent` returns a mouse-moved event whenever the cursor is outside of that region. For example, Figure 8-4 shows a document window. Your application might define two regions: a region that encloses the text area of the window (the I-beam region), and a region that defines the scroll bars and all other areas outside the text area (the arrow region). If your application has specified the I-beam region to `WaitNextEvent`, the mouse driver continues to display the I-beam cursor until the user moves the cursor out of the region.

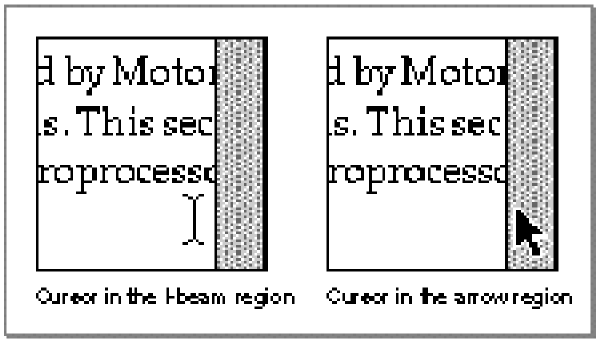
Figure 8-4 A window and its arrow and I-beam regions



When the user moves the cursor out of the I-beam region, `WaitNextEvent` reports a mouse-moved event. Your application can then change the I-beam cursor to the arrow cursor and change the `mouseRgn` parameter to the area defined by the scroll bars and all other areas outside of the I-beam region. The cursor remains an arrow until the user moves the cursor out of the arrow region, at which point your application receives a mouse-moved event.

Figure 8-5 shows how an application might change the cursor from the I-beam cursor to the arrow cursor after receiving a mouse-moved event.

Figure 8-5 Changing the cursor from the I-beam cursor to the arrow cursor



Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise, it will continue to receive mouse-moved events as long as the cursor position is outside the original region.

Listing 8-2 shows an application-defined routine called `MyAdjustCursor`. After receiving any event other than a high-level event, the application's event loop (described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) calls `MyAdjustCursor` to adjust the cursor.

Listing 8-2 Changing the cursor

```
PROCEDURE MyAdjustCursor (mouse: Point; VAR region: RgnHandle);
VAR
   window:        WindowPtr;
   arrowRgn:      RgnHandle;
   iBeamRgn:      RgnHandle;
   iBeamRect:     Rect;
   myData:        MyDocRecHnd;
   windowType:    Integer;
BEGIN
window := FrontWindow;
{Determine the type of window--document, modeless, etc.}
```

```
windowType := MyGetWindowType(window);
CASE windowType OF
   kMyDocWindow:
   BEGIN
      {initialize regions for arrow and I-beam}
      arrowRgn := NewRgn;
      ibeamRgn := NewRgn;
      {set arrow region to large region at first}
      SetRectRgn(arrowRgn, -32768, -32768, 32766, 32766);
      {calculate I-beam region}
      {first get the document's TextEdit view rectangle}
      myData := MyDocRecHnd(GetWRefCon(window));
      iBeamRect := myData^^.editRec^^.viewRect;
      SetPort(window);
      WITH iBeamRect DO
      BEGIN
         LocalToGlobal(topLeft);
         LocalToGlobal(botRight);
      END;
      RectRgn(iBeamRgn, iBeamRect);
      WITH window^.portBits.bounds DO
         SetOrigin(-left, -top);
      {intersect I-beam region with window's visible region}
      SectRgn(iBeamRgn, window^.visRgn, iBeamRgn);
      SetOrigin(0,0);
      {calculate arrow region by subtracting I-beam region}
      DiffRgn(arrowRgn, iBeamRgn, arrowRgn);
      {change the cursor and region parameter as necessary}
      IF PtInRgn(mouse, iBeamRgn) THEN {cursor is in I-beam rgn}
      BEGIN
         SetCursor(GetCursor(iBeamCursor)^^);      {set to I-beam}
         CopyRgn(iBeamRgn, region);     {update the region param}
      END;
      {update cursor if in arrow region}
      IF PtInRgn(mouse, arrowRgn) THEN {cursor is in arrow rgn}
      BEGIN
         SetCursor(arrow);            {set cursor to the arrow}
         CopyRgn(arrowRgn, region);    {update the region param}
      END;
      DisposeRgn(iBeamRgn);
      DisposeRgn(arrowRgn);
   END; {of kMyDocWindow}
   kMyGlobalChangesID:
      MyCalcCursorRgnForModelessDialogBox(window, region);
   kNil:
   BEGIN
      MySetRegionNoWindows(kNil, region);
      SetCursor(arrow);
   END;
   END; {of CASE}
END;
```

The MyAdjustCursor procedure sets the cursor appropriately, according to whether a document window or modeless dialog box is active.

For a document window, MyAdjustCursor defines two regions, specified by the arrowRgn and iBeamRgn variables. If the cursor is inside the region described by the arrowRgn variable, MyAdjustCursor sets the cursor to the arrow cursor and returns the region described by arrowRgn. Similarly, if the cursor is inside the region described by the iBeamRgn variable, MyAdjustCursor sets the cursor to the I-beam cursor and returns the region described by iBeamRgn.

The MyAdjustCursor procedure calculates the two regions by first setting the arrow region to the largest possible region. It then sets the I-beam region to the region described by the document's TextEdit view rectangle. This region typically corresponds to the content area of the window minus the scroll bars. (If your application doesn't use TextEdit for its document window, then set this region as appropriate to your application.) The MyAdjustCursor routine adjusts the I-beam region so that it includes only the part of the content area that is in the window's visible region (for example, to take into account any floating windows that might be over the window). The code in this listing sets the arrow region to include the entire screen except for the region occupied by the I-beam region. (TextEdit is described in *Inside Macintosh: Text*.)

The MyAdjustCursor procedure then determines which region the cursor is in and sets the cursor and region parameter appropriately.

For modeless dialog boxes, MyAdjustCursor calls its own routine to appropriately adjust the cursor for the modeless dialog box. The MyAdjustCursor procedure also appropriately adjusts the cursor if no windows are currently open.

Your application should normally hide the cursor when the user is typing. You can remove the cursor image from the screen by using either the HideCursor or Hide_Cursor procedure. You can hide the cursor temporarily by using the ObscureCursor procedure, or you can hide the cursor in a given rectangle by using the ShieldCursor procedure. To display a hidden cursor, use the ShowCursor or Show_Cursor procedure. Note that you do not need to explicitly show the cursor after your application uses the `ObscureCursor` procedure; instead, the cursor automatically reappears when the user moves the mouse again. These procedures are described in "Hiding and Showing Cursors" beginning on page 8-28.

Creating an Animated Cursor

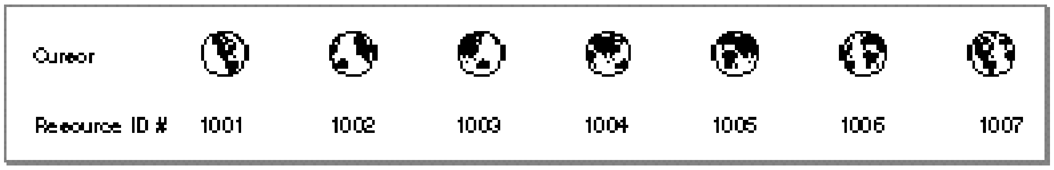
Your application should display an animated cursor when performing a medium-length operation that might cause the user to think that the computer has stopped working. To create an animated cursor, you should

- create a series of `'CURS'` resources that make up the "frames" of the animation
- create an `'acur'` resource with a resource ID of 0
- pass the value `NIL` to the `InitCursorCtl` procedure once in your program code to load these resources
- use either the `RotateCursor` or `SpinCursor` procedure when your application is busy with its task

Note
An alternate, but more code-intensive, method of creating and displaying an animated cursor is shown in the chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*. ◆

Typically, an animated cursor uses four to seven frames. For example, the seven `'CURS'` resources in Figure 8-6 constitute the seven frames of a globe cursor that spins. To create these resources, your application typically uses a high-level utility such as ResEdit, which is available from APDA.

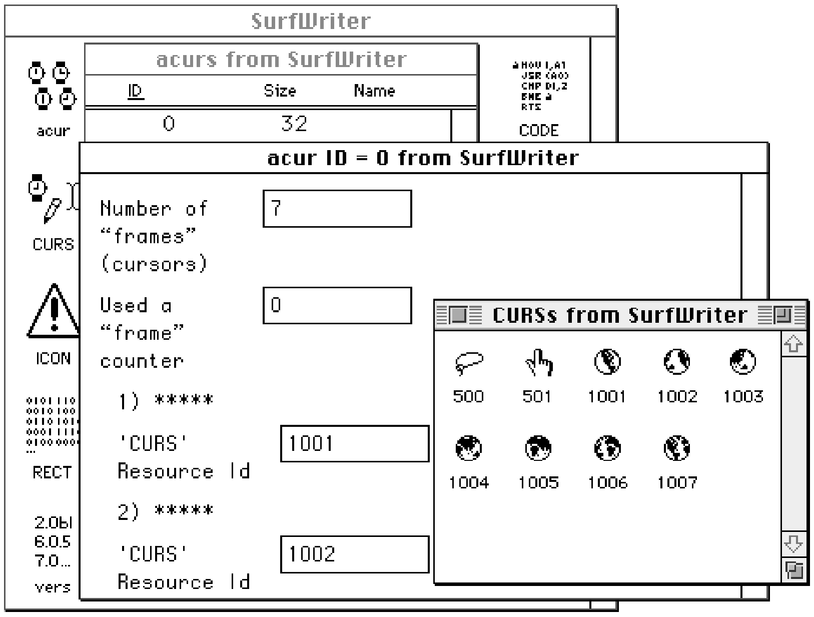
Figure 8-6 The `'CURS'` resources for an animated globe cursor



To collect and order your `'CURS'` frames into a single animation, you must create an `'acur'` resource. This resource specifies the IDs of the `'CURS'` resources and the sequence for displaying them in your animation. If your application uses only one spinning cursor, give your `'acur'` resource a resource ID of 0.

Figure 8-7 shows how the `'CURS'` resources for the spinning globe cursor are specified in an `'acur'` resource using ResEdit.

Figure 8-7 An `'acur'` resource for an animated cursor



acurs from SurfWriter

ID	Size	Name
0	32	

acur ID = 0 from SurfWriter

Number of "frames" (cursors): 7

Used a "frame" counter: 0

1) *****

'CURS' Resource Id: 1001

2) *****

'CURS' Resource Id: 1002

To load the `'acur'` resource and its associated `'CURS'` resources, use the `InitCursorCtl` procedure once prior to calling the `RotateCursor` or `SpinCursor` procedure. If you pass `NIL` to `InitCursorCtl`, then it automatically loads the `'acur'` resource that has an ID of 0 in your application's resource file. If you wish to use multiple animated cursors, you must create multiple `'acur'` resources—that is, one for each series of `'CURS'` resources. Prior to displaying one of your animated cursors with `RotateCursor` or `SpinCursor`, you must call the Resource Manager function `GetResource` to return a handle to its `'acur'` resource. Your application must coerce that handle to one of type `acurHandle`, and then pass this handle to the `InitCursorCtl` procedure. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about `GetResource`.

When you call `RotateCursor` or `SpinCursor`, one frame—that is, one `'CURS'` resource—is displayed. When you pass a positive value to the procedure the next time you call it, the next frame specified in the `'acur'` resource is displayed. A negative value passed to either procedure displays the previous frame listed in the `'acur'` resource. The distinction between `RotateCursor` and `SpinCursor` is that your application maintains an index for changing the cursor when calling `RotateCursor`, but your application does not maintain an index for changing the cursor when calling `SpinCursor`; instead, your application must determine the proper interval for calling `SpinCursor`.

Listing 8-3 shows an application-defined routine called `MyRotateCursor`. When the application calling `MyRotateCursor` starts on a medium-length operation and needs to indicate to the user that the operation is in progress, the application sets its global variable `gDone` to `FALSE` and repeatedly calls `MyRotateCursor` until the operation is complete and `gDone` becomes `TRUE`.

Listing 8-3 Animating a cursor with the `RotateCursor` procedure

```
PROCEDURE MyRotateCursor;
BEGIN
   IF NOT gDone THEN
   BEGIN
      RotateCursor(TickCount);
   END;
END;
```

Listing 8-3 uses the Event Manager function `TickCount` to maintain an index for `RotateCursor` to use when displaying the frames for an animated cursor. (A tick is approximately 1/60 of a second; `TickCount` returns the number of ticks since the computer started up.) When the value passed as a parameter to `RotateCursor` is a multiple of 32, then `RotateCursor` displays the next frame in the animation.

Listing 8-4 shows an application-defined routine called `MySpinCursor`. As you see in Listing 8-4, the application does not maintain an index for displaying the frames for an animated cursor. Instead, every time `SpinCursor` is called, the next frame in the animation is displayed.

Listing 8-4 Animating a cursor with the `SpinCursor` procedure

```
PROCEDURE MySpinCursor;
BEGIN
   IF NOT gDone THEN
      SpinCursor(0);
   END;
```

If the operation takes less than a second or two, your application can simply use the `SetCursor` procedure to display the cursor with the resource ID represented by the `watchCursor` constant. If the operation will take longer than several seconds (a lengthy operation), your application should display a status indicator in a dialog box to show the user the estimated total time and the elapsing time of the operation. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about creating and displaying dialog boxes.

Cursor Utilities Reference

This section describes the data structures, routines, and resources that are specific to cursors. “Data Structures” shows the Pascal data structures for the `Bits16` array and the `Cursor`, `CCrsr`, `Cursors`, and `Acur` records. “Routines” describes the routines for initializing cursors, managing black-and-white cursors, managing color cursors, hiding and showing cursors, and displaying animated cursors. “Resources” describes the cursor resource, the color cursor resource, and the animated cursor resource. The constants that represent values for the standard cursors are listed in “Summary of Cursor Utilities.”

Data Structures

Your application typically does not create the data structures described in this section. Although you can create a `Cursor` record and its associated `Bits16` array in your program code, it is usually easier to create a black-and-white cursor in a cursor resource, which is described on page 8-33. Similarly, you can create a `CCrsr` record in your program code, but it is usually easier to create a color cursor in a color cursor resource, which is described on page 8-34. The `Cursors` data type contains the standard cursors you can display. Finally, you usually list animated cursors in an animated cursor resource, which is described on page 8-36, instead of creating them in an `Acur` record.

Bits16

The `Bits16` array is used by the `Cursor` record to hold a black-and-white, 16-by-16 pixel square image.

```
Bits16 = ARRAY[0..15] OF Integer;
```

Cursor

Your application typically does not create `Cursor` records, which are data structures of type `Cursor`. Although you can create a `Cursor` record and its associated `Bits16` array in your program code, it is usually easier to create a black-and-white cursor in a cursor resource, which is described on page 8-33.

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'`) resource. When your application uses the `GetCursor` function (described on page 8-24) to get a cursor from a `'CURS'` resource, `GetCursor` uses the Resource Manager to load the resource into memory as a `Cursor` record. Your application can then display the color cursor by using the `SetCursor` procedure, which is described on page 8-25.

A `Cursor` record is defined as follows:

```
TYPE CursPtr    = ^Cursor;
CursHandle      = ^CursPtr;
Cursor =
   RECORD
      data:     Bits16;  {cursor image}
      mask:     Bits16;  {cursor mask}
      hotSpot: Point;    {point aligned with mouse}
   END;
```

Field descriptions

`data`	Cursor image data, which must begin on a word boundary. The `Bits16` data type for this field is described in the preceding section.
`mask`	The cursor's mask, whose effects are shown in Table 8-1. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape. The `Bits16` data type for this field is described in the preceding section.
`hotSpot`	A point in the image that aligns with the mouse location.This field aligns a point (not a bit) in the image with the mouse location on the screen. Whenever the user moves the mouse, the low-level interrupt-driven mouse routines move the cursor. When the user clicks, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates.

The cursor appears on the screen as a 16-by-16 pixel square. The appearance of each bit of the square is determined by the corresponding bits in the data and mask and, if the mask bit is 0, by the pixel under the cursor, as shown in Table 8-1.

Table 8-1 Cursor appearance

Data	Mask	Resulting pixel on screen
0	1	White
1	1	Black
0	0	Same as pixel under cursor
1	0	Inverse of pixel under cursor

Notice that if all mask bits are 0, the cursor is completely transparent, in that the image under the cursor can still be viewed. Pixels under the white part of the cursor appear unchanged; under the black part of the cursor, black pixels show through as white.

Basic QuickDraw supplies a predefined cursor in the global variable named `arrow`; this is the standard arrow cursor.

CCrsr

Your application typically does not create `CCrsr` records, which are data structures of type `CCrsr`. Although you can create a `CCrsr` record, it is usually easier to create a color cursor in a color cursor resource, which is described on page 8-34.

A color cursor is a 256-pixel color image in a 16-by-16 pixel square usually defined in a color cursor (`'crsr'`) resource. When your application uses the `GetCCursor` function (described on page 8-26) to get a color cursor from a `'crsr'` resource, `GetCCursor` uses the Resource Manager to load the resource into memory as a `CCrsr` record. Your application can then display the color cursor by using the `SetCCursor` procedure, which is described on page 8-26.

The `CCrsr` record is substantially different from the `Cursor` record described in the preceding section; the fields `crsr1Data`, `crsrMask`, and `crsrHotSpot` in the `CCrsr` record are the only ones that have counterparts in the `Cursor` record. A `CCrsr` record is defined as follows:

```
TYPE CCrsrHandle =   ^CCrsrPtr;
CCrsrPtr =           ^CCrsr;
CCrsr =
   RECORD
      crsrType:      Integer;       {type of cursor}
      crsrMap:       PixMapHandle;  {the cursor's PixMap record}
      crsrData:      Handle;        {cursor's data}
      crsrXData:     Handle;        {expanded cursor data}
      crsrXValid:    Integer;       {depth of expanded data}
      crsrXHandle:   Handle;        {reserved for future use}
      crsr1Data:     Bits16;        {1-bit cursor}
      crsrMask:      Bits16;        {cursor's mask}
      crsrHotSpot:   Point;         {cursor's hot spot}
      crsrXTable:    LongInt;       {private}
      crsrID:        LongInt;       {ctSeed for expanded cursor}
   END;
```

Field descriptions

`crsrType`	The type of cursor. Possible values are $8000 for a black-and-white cursor and $8001 for a color cursor.
`crsrMap`	A handle to the `PixMap` record defining the cursor's characteristics. `PixMap` records are described in the chapter "Color QuickDraw" in this book.
`crsrData`	A handle to the cursor's pixel data.
`crsrXData`	A handle to the expanded pixel image used internally by Color QuickDraw.
`crsrXValid`	The depth of the expanded cursor image. If you change the cursor's data or color table, you should set this field to 0 to cause the cursor to be re-expanded. You should never set it to any other values.
`crsrXHandle`	Reserved for future use.
`crsr1Data`	A 16-by-16 pixel image with a pixel depth of 1 to be displayed when the cursor is on screens with pixel depths of 1 or 2 bits.
`crsrMask`	The cursor's mask data. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape. The same 1-bit mask is used with images specified by the `crsrData` and `crsr1Data` fields.
`crsrHotSpot`	The cursor's hot spot.
`crsrXTable`	Reserved for future use.
`crsrID`	The color table seed for the cursor.

The first four fields of the `CCrsr` record are similar to the first four fields of the `PixPat` record, and are used in the same manner by Color QuickDraw. See the chapter "Color QuickDraw" in this book for information about `PixPat` records.

The display of a cursor involves a relationship between a mask, stored in the `crsrMask` field with the same format used for 1-bit cursor masks, and an image. There are two possible sources for a color cursor's image. When the cursor is on a screen whose depth is 1 or 2 bits per pixel, the image for the cursor is taken from the `crsr1Data` field, which contains bitmap cursor data (similar to the bitmap in a `'CURS'` resource).

When the screen depth is greater than 2 bits per pixel, the `crsrMap` field and the `crsrData` field define the image. The pixels within the mask replace the destination pixels. Color QuickDraw transfers the pixels outside the mask into the destination pixels using the XOR Boolean transfer mode. Therefore, if pixels outside the mask are white, the destination pixels aren't changed. If pixels outside the mask are all black, the destination pixels are inverted. All other values outside of the mask cause unpredictable results. See the discussion of Boolean transfer modes in the chapter "Color QuickDraw" in this book for more information about the XOR Boolean transfer mode.

To work properly, a color cursor's image should contain white pixels (R = G = B = $FFFF) for the transparent part of the image, and black pixels (R = G = B = $0000) for the part of the image to be inverted, in addition to the other colors in the cursor's image. Thus, to define a cursor that contains two colors, it's necessary to use a 2-bit cursor image (that is, a four-color image).

If your application changes the value of your color cursor data or its color table, it should set the `crsrXValid` field to 0 to indicate that the color cursor's data needs to be re-expanded, and it should assign a new unique value to the `crsrID` field (unique values can be obtained using the Color Manager function `GetCTSeed`, which is described in *Inside Macintosh: Advanced Color Imaging*. Then your application should call `SetCCursor` to display the changed color cursor.

Cursors

When passing a value to the `Show_Cursor` procedure (described on page 8-30), you can use the `Cursors` data type to represent the kind of cursor to show. The `Cursors` data type is defined as follows:

```
TYPE Cursors =       {values to pass to Show_Cursor}
  (HIDDEN_CURSOR,    {the current cursor}
   I_BEAM_CURSOR,    {the I-beam cursor; to select text}
   CROSS_CURSOR,     {the crosshairs cursor; to draw }
                     { graphics}
   PLUS_CURSOR,      {the plus sign cursor; to select }
                     { cells}
   WATCH_CURSOR,     {the wristwatch cursor; to }
                     { indicate a short operation in }
                     { progress}
   ARROW_CURSOR);    {the standard cursor}
```

Acur

Your application typically does not create `Acur` records, which are data structures of type `Acur`. Although you can create an `Acur` record, which specifies the `'CURS'` resources to use in an animated cursor sequence, it is usually easier to create an animated cursor (`'acur'`) resource, which is described on page 8-36.

When your application uses the `InitCursorCtl` procedure (described on page 8-22), the Resource Manager loads an animated cursor resource into memory as an `Acur` record, which in turn is used by the `RotateCursor` procedure or `SpinCursor` procedure (both described on page 8-32) when sequencing through `'CURS'` resources.

An Acur resource is defined as follows:

```
TYPE acurPtr = ^Acur;
acurHandle = ^acurPtr;
Acur =
   RECORD
      n:       Integer;    {number of cursors ("frames")}
      index:   Integer;    {reserved}
      frame1:  Integer;    {'CURS' resource ID for frame #1}
      fill1:   Integer;    {reserved}
      frame2:  Integer;    {'CURS' resource ID for frame #2}
      fill2:   Integer;    {reserved}
      frameN:  Integer;    {'CURS' resource ID for frame #N}
      fillN:   Integer;    {reserved}
   END;
```

Field descriptions

`n`	The number of frames in the animated cursor.
`index`	Used by basic QuickDraw to create the animation.
`frame1`	The resource ID of the cursor (`'CURS'`) resource for the first frame sequence of the animation. The cursor resource is described on page 8-33.
`fill1`	Reserved.
`frame2`	The resource ID of the cursor resource for the next frame in the sequence of the animation.
`fill2`	Reserved.
`frameN`	The resource ID of the cursor resource for the last frame used in the sequence of the animation.
`fillN`	Reserved.

Routines

This section describes the routines you use to initialize the cursor, manage a black-and-white cursor, manage a color cursor, hide and show the cursor, and display an animated cursor.

Initializing Cursors

When your application starts up, the Finder sets the cursor to a wristwatch; this indicates that a short operation is in progress. When your application nears completion of its initialization tasks, it should call the `InitCursor` procedure to change the cursor from a wristwatch to an arrow.

If your application uses an animated cursor to indicate that an operation of medium length is under way, it should also call the `InitCursorCtl` procedure to load its `'acur'` resource and associated `'CURS'` resources.

InitCursor

You use the `InitCursor` procedure to set the current cursor to the standard arrow and make it visible.

```
PROCEDURE InitCursor;
```

DESCRIPTION

The `InitCursor` procedure sets the current cursor to the standard arrow and sets the cursor level to 0, making the cursor visible. (A value of –1 makes the cursor invisible.) The cursor level keeps track of the number of times the cursor has been hidden to compensate for nested calls to the `HideCursor` and `ShowCursor` procedures.

SEE ALSO

For a description of the `HideCursor` procedure, see page 8-28. For a description of the `ShowCursor` procedure, see page 8-30. Listing 8-1 on page 8-6 illustrates how to use the `InitCursor` procedure.

InitCursorCtl

To load the resources necessary for displaying an animated cursor, use the `InitCursorCtl` procedure.

```
PROCEDURE InitCursorCtl (newCursors: UNIV acurHandle);
```

`newCursors`
: A handle to an `Acur` record (described on page 8-20) that specifies the cursor resources you want to use in your animation. If you specify `NIL` in this parameter, `InitCursorCtl` loads the animated cursor resource (described on page 8-36) with resource ID 0—as well as the cursor resources (described on page 8-33) specified therein—from your application's resource file.

DESCRIPTION

The `InitCursorCtl` procedure loads the cursor resources for an animated cursor sequence into memory. Your application should call the `InitCursorCtl` procedure once prior to calling the `RotateCursor` procedure (described on page 8-32) or the `SpinCursor` procedure (described on page 8-32).

If your application passes `NIL` in the `newCursors` parameter, `InitCursorCtl` loads the `'acur'` resource with resource ID 0, as well as the `'CURS'` resources whose resource IDs are specified in the `'acur'` resource. If any of the resources cannot be loaded, the cursor does not change when you call `RotateCursor` or `SpinCursor`. Otherwise, the `RotateCursor` procedure and the `SpinCursor` procedure display in sequence the cursors specified in these resources.

If your application does not pass `NIL` in the `newCursors` parameter, it must pass a handle to an `Acur` record. Your application can use the Resource Manager function `GetResource` to obtain a handle to an `'acur'` resource, which your application should then coerce to a handle of type `acurHandle` when passing it to `InitCursorCtl`.

If your application calls the `RotateCursor` or `SpinCursor` procedure without calling `InitCursorCtl`, `RotateCursor` and `SpinCursor` automatically call `InitCursorCtl`. However, since you won't know the state of memory, any memory allocated by the Resource Manager for animating cursors may load into an undesirable location, possibly causing fragmentation. Calling the `InitCursorCtl` procedure during your initialization process has the advantage of causing the memory allocation when you can control its location. For information on using the `InitCursorCtl` procedure during your initialization process, see "Initializing the Cursor" on page 8-6.

SPECIAL CONSIDERATION

If you want to use multiple `'acur'` resources repeatedly during the execution of your application, be aware that the `InitCursorCtl` procedure changes each `frameN` and `fillN` integer pair within the `Acur` record in memory to a handle to the corresponding `'CURS'` resource, which is also in memory. Thus, if the `newCursors` parameter is not `NIL` when your application calls the `InitCursorCtl` procedure, your application must guarantee that `newCursors` always points to a fresh copy of an `'acur'` resource.

SEE ALSO

Listing 8-1 on page 8-6 illustrates how to initialize an animated cursor by using the `InitCursorCtl` procedure. Listing 8-3 on page 8-15 shows how to animate the cursor with the `RotateCursor` procedure, and Listing 8-4 on page 8-15 shows how to animate the cursor with the `SpinCursor` procedure.

Changing Black-and-White Cursors

When you use the `InitCursor` procedure described on page 8-22, the cursor changes from a wristwatch to an arrow. You can change the cursor to another shape by using the `GetCursor` function to load another cursor into memory and then using the `SetCursor` procedure to display it on the screen.

GetCursor

You use the `GetCursor` function to load a cursor resource (described on page 8-33) into memory. You can then display the cursor specified in this resource by calling the `SetCursor` procedure (described in the next section).

```
FUNCTION GetCursor (cursorID: Integer): CursHandle;
```

`cursorID` The resource ID for the cursor you want to display. You can supply one of these constants to get a handle to one of the standard cursors:

```
CONST
   iBeamCursor = 1;  {to select text}
   crossCursor = 2;  {to draw graphics}
   plusCursor  = 3;  {to select cells}
   watchCursor = 4;  {to indicate a short operation }
                     { in progress}
```

DESCRIPTION

The `GetCursor` function returns a handle to a `Cursor` record (described on page 8-16) for the cursor with the resource ID that you specify in the `cursorID` parameter. If the resource can't be read into memory, `GetCursor` returns `NIL`.

To get a handle to a color cursor, use the `GetCCursor` function, which is described on page 8-26.

SEE ALSO

Listing 8-2 on page 8-10 illustrates how to use the `GetCursor` and `SetCursor` routines to change the cursor's shape.

SetCursor

After using the `GetCursor` function to return a handle to a cursor as described in the preceding section, you can use the `SetCursor` procedure to make that cursor the current cursor.

```
PROCEDURE SetCursor (crsr: Cursor);
```

`crsr`	A `Cursor` record, as described on page 8-16.

DESCRIPTION

The `SetCursor` procedure displays the cursor you specify in the `crsr` parameter. If the cursor is hidden, it remains hidden and attain its new appearance only when it's uncovered. If the cursor is already visible, it changes to the new appearance immediately.

You need to use the `InitCursor` procedure (described on page 8-22) to initialize the standard arrow cursor and make it visible on the screen before you can call `SetCursor` to change the cursor's appearance.

To display a color cursor, you must use the `SetCCursor` procedure, which is described on page 8-26.

SEE ALSO

Listing 8-2 on page 8-10 illustrates how to use the `GetCursor` and `SetCursor` routines to change the cursor's shape.

Changing Color Cursors

This section describes how to create and display color cursors on the screen. It might be useful to display a color cursor when the user is drawing or typing in color. For example, the insertion point could appear in the color that is being used. Except for multicolored paintbrush cursors, the cursor shouldn't contain more than one color at once because it's hard for the eye to distinguish small areas of color.

To display a color cursor, you load the cursor resource into memory using the `GetCCursor` function. Then you specify the cursor to display on the screen using the `SetCCursor` procedure. Use the `DisposeCCursor` procedure to release the memory used by the color cursor. Although you should never need to do so (because Color QuickDraw handles this), the `AllocCursor` procedure reallocates cursor memory.

GetCCursor

You use the `GetCCursor` function to load a color cursor resource into memory.

```
FUNCTION GetCCursor (crsrID: Integer): CCrsrHandle;
```

`crsrID` The resource ID of the cursor that you want to display.

DESCRIPTION

The `GetCCursor` function creates a new `CCrsr` record and initializes it using the information in the `'crsr'` resource with the specified ID. The `GetCCursor` function returns a handle to the new `CCrsr` record. You can then display this cursor on the screen by calling `SetCCursor`. If a resource with the specified ID isn't found, then this function returns a `NIL` handle.

Since the `GetCCursor` function creates a new `CCrsr` record each time it is called, your application shouldn't call the `GetCCursor` function before each call to the `SetCCursor` procedure (unlike the way `GetCursor` and `SetCursor` are normally used). The `GetCCursor` function doesn't dispose of or detach the resource, so resources of type `'crsr'` should typically be purgeable. You should call the `DisposeCCursor` procedure (described on page 8-27) when you are finished using the color cursor created with `GetCCursor`.

SEE ALSO

For a description of the `'crsr'` resource format, see page 8-34. For a description of the `CCrsr` record, see page 8-18. For a description of the `SetCCursor` procedure, see the next section.

SetCCursor

You use the `SetCCursor` procedure to specify a color cursor for display on the screen.

```
PROCEDURE  SetCCursor (cCrsr: CCrsrHandle);
```

`cCrsr` A handle to the color cursor to be displayed.

DESCRIPTION

The `SetCCursor` procedure allows your application to set a color cursor for display on the screen. At the time the cursor is set, it's expanded to the current screen depth so that it can be drawn rapidly. You must call `GetCCursor` before you call `SetCCursor`; however, you can make several subsequent calls to `SetCCursor` once `GetCCursor` creates the `CCrsr` record.

If your application has changed the cursor's data or its color table, it must also invalidate the `crsrXValid` and `crsrID` fields of the `CCrsr` record before calling `SetCCursor`.

DisposeCCursor

You use the `DisposeCCursor` procedure to dispose of all records allocated by the `GetCCursor` function. The `DisposeCCursor` procedure is also available as the `DisposCCursor` procedure.

```
PROCEDURE DisposeCCursor (cCrsr: CCrsrHandle);
```

`cCrsr`	A handle to the color cursor to be disposed of.

DESCRIPTION

The `DisposeCCursor` procedure disposes of memory allocated by the `GetCCursor` function. You should use `DisposeCCursor` for each call to the `GetCCursor` function (described on page 8-26).

AllocCursor

Although you typically won't need to, you can use the `AllocCursor` procedure to reallocate cursor memory.

```
PROCEDURE AllocCursor;
```

DESCRIPTION

Under normal circumstances, you should never need to use this procedure, since Color QuickDraw handles reallocation of cursor memory.

Hiding and Showing Cursors

You can remove the cursor image from the screen by using either the `HideCursor` or `Hide_Cursor` procedure. You can hide the cursor temporarily by using the `ObscureCursor` procedure, or you can hide the cursor in a given rectangle by using the `ShieldCursor` procedure. Your application should hide the cursor when the user is typing, for example. To display a cursor hidden by the `HideCursor`, `Hide_Cursor`, or `ObscureCursor` procedure, use the `ShowCursor` or `Show_Cursor` procedure. (When you use `ObscureCursor` to hide the cursor, the cursor is redisplayed automatically the next time the user moves the mouse.)

HideCursor

You can use the `HideCursor` procedure to remove the cursor from the screen.

```
PROCEDURE HideCursor;
```

DESCRIPTION

The `HideCursor` procedure removes the cursor from the screen, restores the bits under the cursor image, and decrements the cursor level (which `InitCursor` initialized to 0). You might want to use `HideCursor` when the user is using the keyboard to create content in one of your application's windows. Every call to `HideCursor` should be balanced by a subsequent call to the `ShowCursor` procedure, which is described on page 8-30.

Hide_Cursor

You can use the `Hide_Cursor` procedure to hide the cursor if it is visible on the screen. The `Hide_Cursor` procedure is functionally the same as the `HideCursor` procedure described in the preceding section.

```
PROCEDURE Hide_Cursor;
```

DESCRIPTION

The `Hide_Cursor` procedure calls the `HideCursor` procedure to remove the cursor's image from the screen and decrements the cursor level by 1. Every call to `Hide_Cursor` should be balanced by a subsequent call to the `Show_Cursor` procedure, which is described on page 8-30. Before using `Hide_Cursor`, you must use the `InitCursorCtl` procedure, which is described on page 8-22.

ObscureCursor

You use the `ObscureCursor` procedure to hide the cursor until the next time the user moves the mouse.

```
PROCEDURE ObscureCursor;
```

DESCRIPTION

The `ObscureCursor` procedure temporarily hides the cursor; the cursor is redisplayed the next time the user moves the mouse. Your application normally calls `ObscureCursor` when the user begins to type. Unlike `HideCursor` (which is described on page 8-28), `ObscureCursor` has no effect on the cursor level and must not be balanced by a call to `ShowCursor`.

ShieldCursor

You can use the `ShieldCursor` procedure to hide the cursor in a rectangle.

```
PROCEDURE ShieldCursor (shieldRect: Rect; offsetPt: Point);
```

`shieldRect`
: A rectangle in which the cursor is hidden whenever the cursor intersects the rectangle. The rectangle may be specified in global or local coordinates. If you are using global coordinates, pass (0,0) in the `offsetPt` parameter. If you are using the local coordinates of a graphics port, pass the coordinates for the upper-left corner of the graphics port's boundary rectangle in the `offsetPt` parameter.

`offsetPt`
: A point value for the offset of the rectangle. Like the basic QuickDraw procedure `LocalToGlobal`, the `ShieldCursor` procedure offsets the coordinates of the rectangle by the coordinates of this point.

DESCRIPTION

If the cursor and the given rectangle intersect, `ShieldCursor` hides the cursor. If they don't intersect, the cursor remains visible while the mouse isn't moving, but is hidden when the mouse moves. This procedure may be useful when using a feature such as QuickTime to display content in a specified rectangle. When a QuickTime movie is animating, the cursor should not be visible in front of the movie.

The `ShieldCursor` procedure decrements the cursor level and should be balanced by a call to the `ShowCursor` procedure, which is described in the next section.

ShowCursor

You use the `ShowCursor` procedure to display a cursor hidden by the `HideCursor` or `ShieldCursor` procedure.

```
PROCEDURE ShowCursor;
```

DESCRIPTION

The `ShowCursor` procedure increments the cursor level, which may have been decremented by the `HideCursor` or `ShieldCursor` procedure, and displays the cursor on the screen when the level is 0. A call to the `ShowCursor` procedure should balance each previous call to the `HideCursor` or `ShieldCursor` procedure. The level isn't incremented beyond 0, so extra calls to `ShowCursor` have no effect.

Low-level interrupt-driven routines link the cursor with the mouse position, so that if the cursor level is 0 (visible), the cursor automatically follows the mouse.

If the cursor has been changed with the `SetCursor` procedure while hidden, `ShowCursor` displays the new cursor.

SEE ALSO

For a description of the `HideCursor` procedure, see page 8-28. The `ShieldCursor` procedure is described on page 8-29, and the `SetCursor` procedure is described on page 8-25.

Show_Cursor

You use the `Show_Cursor` procedure to display the cursor on the screen if you have used the `Hide_Cursor` procedure (described on page 8-28) to remove the cursor from the screen.

```
PROCEDURE Show_Cursor (cursorKind: Cursors);
```

`cursorKind`
The kind of cursor to show. To specify one of the standard cursors, you can use one of these values defined by the `Cursors` data type.

```
TYPE Cursors =      {values to pass Show_Cursor}
  (HIDDEN_CURSOR,   {the current cursor}
   I_BEAM_CURSOR,   {the I-beam cursor; to select text}
   CROSS_CURSOR,    {the crosshairs cursor; to draw }
                    { graphics}
   PLUS_CURSOR,     {the plus sign cursor; to select }
                    { cells}
```

```
            WATCH_CURSOR,    {the wristwatch cursor; to }
                             { indicate a short operation in }
                             { progress}
            ARROW_CURSOR);   {the standard cursor}
```

DESCRIPTION

The `Show_Cursor` procedure increments the cursor level, which may have been decremented by the `Hide_Cursor` procedure, and displays the specified cursor on the screen only if the level becomes 0 (it is never incremented beyond 0). You can specify one of the standard cursors or the current cursor by passing one of the previously listed values in the `cursorKind` parameter. If you specify one of the standard cursors, the `Show_Cursor` procedure calls the `SetCursor` procedure for the specified cursor prior to calling `ShowCursor`. If you specify `HIDDEN_CURSOR`, this procedure just calls `ShowCursor`. Before using `Show_Cursor`, you must use the `InitCursorCtl` procedure, which is described on page 8-22.

SPECIAL CONSIDERATIONS

The value `ARROW_CURSOR` works correctly only if the basic QuickDraw global variables have been set up by using the `InitGraf` procedure, which is described in the chapter "Basic QuickDraw" in this book.

SEE ALSO

Figure 8-3 on page 8-8 illustrates the cursors represented by the `Cursors` data type.

Displaying Animated Cursors

This section describes how to display an animated cursor using the `RotateCursor` procedure or the `SpinCursor` procedure. You use an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working. The two procedures are similar, but you must maintain a counter with the `RotateCursor` procedure.

You need to call the `InitCursorCtl` procedure to load your cursor resources before using the routines described in this section. For information about using the `InitCursorCtl` procedure, see page 8-22.

RotateCursor

You can use the `RotateCursor` procedure to display an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working.

```
PROCEDURE RotateCursor (counter: LongInt);
```

`counter`	An incrementing or decrementing index maintained by your application. When the index is a multiple of 32, the next cursor frame is used in the animation. A positive counter moves forward through the cursor frames, and a negative counter moves backward through the cursor frames.

DESCRIPTION

The `RotateCursor` procedure animates whatever sequence of cursors you set up by using the `InitCursorCtl` procedure. If the value of `counter` is a multiple of 32, the `RotateCursor` procedure calls the `SetCursor` procedure to set the cursor to the next cursor frame. `RotateCursor` does not show the cursor if it is currently hidden. If the cursor is hidden, you can show it by making a call to `ShowCursor` or `Show_Cursor` (both described on page 8-30).

SEE ALSO

For an example of using the `RotateCursor` procedure, see Listing 8-3 on page 8-15.

SpinCursor

You can use the `SpinCursor` procedure to display an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working.

```
PROCEDURE SpinCursor (increment: Integer);
```

`increment`	A value that determines the sequencing direction of the cursor. A positive increment moves forward through the cursor frames, and a negative increment moves backward through the cursor frames. A 0 value for the increment resets the counter to 0 and steps to the next cursor frame.

DESCRIPTION

The `SpinCursor` procedure is similar to the `RotateCursor` procedure except that, instead of passing a counter, you pass a value that indicates which direction to spin the cursor. Your application is responsible for determining the proper intervals at which to call `SpinCursor`. Your application specifies the increment to be counted, either positive or negative, and `SpinCursor` adds the increment to its counter. The sign of the increment, not the sign of the accumulated value of the `SpinCursor` counter, determines the cursor's direction of spin.

SEE ALSO

For an example of using the `SpinCursor` procedure, see Listing 8-4 on page 8-15.

Resources

This section describes the cursor (`'CURS'`) resource, the color cursor (`'crsr'`) resource, and the animated cursor (`'acur'`) resource. Your application can use a `'CURS'` resource to create a black-and-white cursor other than the standard cursors or a `'crsr'` resource to create a color cursor to display on color screens. Your application can use an `'acur'` resource to create an animated cursor to display when a medium-length operation is taking place. These resource types should be marked as purgeable. See the discussion of the pointing device in *Macintosh Human Interface Guidelines* for more information on when to use different types of cursors in your application; see also the discussion of color in the same book.

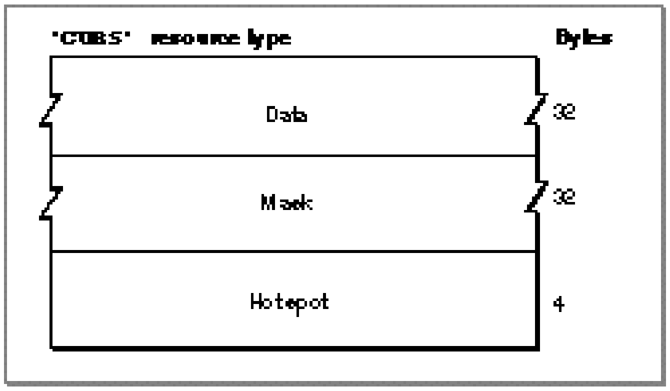
The Cursor Resource

You can use a cursor resource to define a cursor to display in your application. A cursor resource is a resource of type `'CURS'`. All cursor resources must be marked purgeable and must have resource IDs greater than 128. You use the `GetCursor` function (described on page 8-24) to obtain a cursor stored in a `'CURS'` resource. QuickDraw reads the requested resource, copies it, and then alters the copy before passing it to your application.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level utility such as the ResEdit application to create `'CURS'` resources. You can then use the DeRez decompiler to convert your `'CURS'` resources into Rez input when necessary.

The compiled output format for a 'CURS' resource is illustrated in Figure 8-8.

Figure 8-8 Format of a compiled cursor ('CURS') resource



The compiled version of a 'CURS' resource contains the following elements:

- Data. A bitmap for the cursor.
- Mask. A bitmap for the cursor's mask. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape.
- Hot spot. The cursor's hot spot.

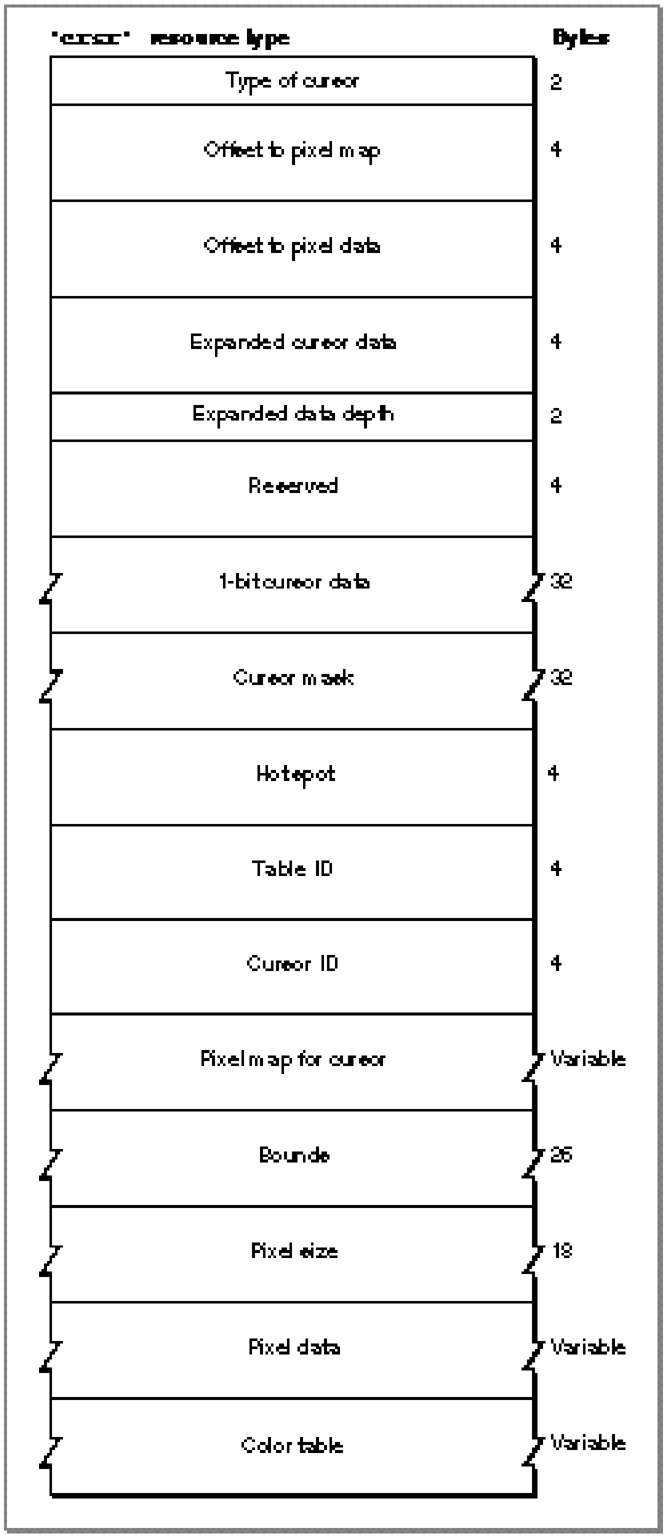
The Color Cursor Resource

You can use a color cursor resource to define a colored cursor to display in your application. A color cursor resource is a resource of type 'crsr'. All color cursor resources must be marked purgeable and must have resource IDs greater than 128. You use the GetCCursor function (described on page 8-26) to obtain a color cursor stored in a 'crsr' resource. Color QuickDraw reads the requested resource, copies it, and then alters the copy before passing it to the application. Each time you call GetCCursor, you get a new copy of the cursor. This means that you should call GetCCursor only once for a color cursor, even if you call the SetCCursor procedure many times.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level utility such as the ResEdit application to create 'crsr' resources. You can then use the DeRez decompiler to convert your 'crsr' resources into Rez input when necessary.

The compiled output format for a 'crsr' resource is illustrated in Figure 8-9.

Figure 8-9 Format of a compiled color cursor (`'crsr'`) resource



The compiled version of a `'crsr'` resource contains the following elements:

- Type of cursor. A value of $8001 identifies this as a color cursor. A value of $8000 identifies this as a black-and-white cursor.
- Offset to `PixMap` record. This offset is from the beginning of the resource data.
- Offset to pixel data. This offset is from the beginning of the resource data.
- Expanded cursor data. This expanded pixel image is used internally by Color QuickDraw.
- Expanded data depth. This is the pixel depth of the expanded cursor image.
- Reserved. The Resource Manager uses this element for storage.
- Cursor data. This field contains a 16-by-16 pixel 1-bit image to be displayed when the cursor is on 1-bit or 2-bit screens.
- Cursor mask. A bitmap for the cursor's mask. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape.
- Hot spot. The cursor's hot spot.
- Table ID. This contains an offset to the color table data from the beginning of the resource data.
- Cursor ID. This contains the cursor's resource ID.
- Pixel map. This pixel map describes the image when drawing the color cursor. The pixel map contains an offset to the color table data from the beginning of the resource.
- Bounds. The boundary rectangle of the cursor.
- Pixel size. The number of pixels per bit in the cursor.
- Pixel data. The data for the cursor.
- Color table. A color table containing the color information for the cursor's pixel map.

The Animated Cursor Resource

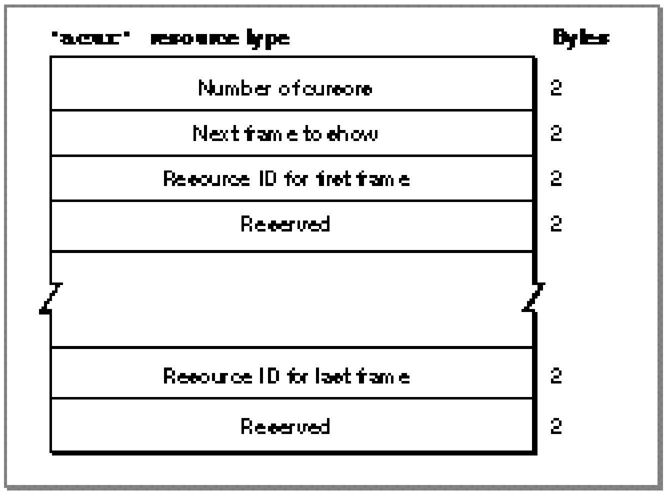
You can use an animated cursor resource to define a set of frames for an animated cursor to display in your application. An animated cursor resource is a resource of type `'acur'`.

If you pass `NIL` to `InitCursorCtl` (described on page 8-22), it automatically loads the `'acur'` resource that has an ID of 0 in your application's resource file. If you wish to use multiple `'acur'` resources, you must give them resources IDs greater than 128, and you must use the Resource Manager function `GetResource` to obtain handles to them. You must then coerce their handles to type `acurHandle`, which you pass to `InitCursorCtl`. You use the `SpinCursor` or `RotateCursor` procedure to animate the cursors stored in an `'acur'` resource.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level tool such as the ResEdit application to create `'acur'` resources. You can then use the DeRez decompiler to convert your `'acur'` resources into Rez input when necessary.

The compiled output format for an `'acur'` resource is illustrated in Figure 8-10.

Figure 8-10 Format of a compiled animated cursor (`'acur'`) resource



The compiled version of an `'acur'` resource contains the following elements:

- Number of cursors. The number of frames used to animate the cursor.
- Next frame to show. Reserved.
- Resource ID of the cursor resource that defines the first frame of the animation.
- Reserved.
- Resource ID of the cursor resource that defines the last frame of the animation.
- Reserved.

Summary of Cursor Utilities

Pascal Summary

Constants

```
CONST
   iBeamCursor = 1;  {used in text editing}
   crossCursor = 2;  {often used for manipulating graphics}
   plusCursor  = 3;  {often used for selecting fields in an array}
   watchCursor = 4;  {used to mean a short operation is in progress}
```

Data Types

```
TYPE  Bits16 = ARRAY[0..15] OF Integer;

      CursPtr = ^Cursor;
      CursHandle = ^CursPtr;
      Cursor =
      RECORD
         data:    Bits16;  {cursor image}
         mask:    Bits16;  {cursor mask}
         hotSpot: Point;   {point aligned with mouse}
      END;

      CCrsrPtr = ^CCrsr;
      CCrsrHandle = ^CCrsrPtr;
      CCrsr =
      RECORD
         crsrType:   Integer;       {type of cursor}
         crsrMap:    PixMapHandle;  {the cursor's PixMap record}
         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;       {1-bit cursor}
   crsrMask:    Bits16;       {cursor's mask}
   crsrHotSpot: Point;        {cursor's hot spot}
   crsrXTable:  LongInt;      {private}
   crsrID:      LongInt;      {ctSeed for expanded cursor}
END;

Cursors =             {values to pass to Show_Cursor}
   (HIDDEN_CURSOR,    {the current cursor}
    I_BEAM_CURSOR,    {the I-beam cursor; to select text}
    CROSS_CURSOR,     {the crosshairs cursor; to draw graphics}
    PLUS_CURSOR,      {the plus sign cursor; to select cells}
    WATCH_CURSOR,     {the wristwatch cursor; to indicate a }
                      { short operation in progress}
    ARROW_CURSOR);    {the standard cursor}

acurPtr = ^Acur;
acurHandle = ^acurPtr;
Acur =
RECORD
   n:        Integer;    {number of cursors ("frames")}
   index:    Integer;    {reserved}
   frame1:   Integer;    {'CURS' resource ID for frame #1}
   fill1:    Integer;    {reserved}
   frame2:   Integer;    {'CURS' resource ID for frame #2}
   fill2:    Integer;    {reserved}
   frameN:   Integer;    {'CURS' resource ID for frame #N}
   fillN:    Integer;    {reserved}
END;
```

Routines

Initializing Cursors

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

Changing Black-and-White Cursors

```
FUNCTION GetCursor           (cursorID: Integer): CursHandle;
PROCEDURE SetCursor          (crsr: Cursor);
```

Changing Color Cursors

```
{DisposeCCursor is also spelled as DisposCCursor}
FUNCTION GetCCursor          (cursorID: Integer): CCursHandle;
PROCEDURE SetCCursor         (cCrsr: CCrsrHandle);
PROCEDURE DisposeCCursor     (cCrsr: CCrsrHandle);
PROCEDURE AllocCursor;
```

Hiding and Showing Cursors

```
PROCEDURE HideCursor;
PROCEDURE Hide_Cursor;
PROCEDURE ObscureCursor;
PROCEDURE ShieldCursor       (shieldRect: Rect; offsetPt: Point);
PROCEDURE ShowCursor;
PROCEDURE Show_Cursor        (cursorKind: Cursors);
```

Displaying Animated Cursors

```
PROCEDURE RotateCursor       (counter: LongInt);
PROCEDURE SpinCursor         (increment: Integer);
```

C Summary

Constants

```
enum {
   iBeamCursor =  1,    /* used in text editing */
   crossCursor =  2,    /* often used for manipulating graphics */
   plusCursor =   3,    /* often used for selecting fields in an array */
   watchCursor =  4     /* used to mean a short operation is in progress */
};

enum {                /* values to pass to Show_Cursor */
   HIDDEN_CURSOR,     /* the current cursor */
   I_BEAM_CURSOR,     /* the I-beam cursor; to select tect */
   CROSS_CURSOR,      /* the crosshairs cursor; to draw graphics */
   PLUS_CURSOR,       /* the plus sign cursor; to select cells */
   WATCH_CURSOR,      /* the wristwatch cursor; to indicate a short
                         operation in progress */
   ARROW_CURSOR       /* the standard cursor */
};
typedef unsigned char Cursors;
```

Data Types

```
typedef short Bits16[16];

struct Cursor {
   Bits16   data;    /* cursor image */
   Bits16   mask;    /* cursor mask */
   Point    hotSpot; /* point aligned with mouse */
};
typedef struct Cursor Cursor;
typedef Cursor *CursPtr, **CursHandle;

struct CCrsr {
   short          crsrType;      /* type of cursor */
   PixMapHandle   crsrMap;       /* the cursor's PixMap record */
   Handle         crsrData;      /* cursor's data */
   Handle         crsrXData;     /* expanded cursor data */
   short          crsrXValid;    /* depth of expanded data (0 if none) */
   Handle         crsrXHandle;   /* future use */
   Bits16         crsr1Data;     /* 1-bit cursor */
   Bits16         crsrMask;      /* cursor's mask */
   Point          crsrHotSpot;   /* cursor's hot spot */
   long           crsrXTable;    /* private */
   long           crsrID;        /* ctSeed for expanded cursor */
};
typedef struct CCrsr CCrsr;
typedef CCrsr *CCrsrPtr, **CCrsrHandle;

struct Acur {
   short    n;       /* number of cursors ("frames of film") */
   short    index;   /* reserved */
   short    frame1;  /* 'CURS' resource ID for frame #1 */
   short    fill1;   /* reserved */
   short    frame2;  /* 'CURS' resource ID for frame #2 */
   short    fill2;   /* reserved */
   short    frameN;  /* 'CURS' resource ID for frame #N */
   short    fillN;   /* reserved */
};
typedef struct Acur acur,*acurPtr,**acurHandle;
```

Functions

Initializing Cursors

```
pascal void InitCursor      (void);
pascal void InitCursorCtl   (acurHandle newCursors);
```

Changing Black-and-White Cursors

```
pascal CursHandle GetCursor (short cursorID);
pascal void SetCursor       (const Cursor *crsr);
```

Changing Color Cursors

```
/* DisposeCCursor is also spelled as DisposCCursor */
pascal CCrsrHandle GetCCursor
                            (short crsrID);
pascal void SetCCursor      (CCrsrHandle cCrsr);
pascal void DisposeCCursor  (CCrsrHandle cCrsr);
pascal void AllocCursor     (void);
```

Hiding and Showing Cursors

```
pascal void HideCursor      (void);
pascal void Hide_Cursor     (void);
pascal void ObscureCursor   (void);
pascal void ShieldCursor    (const Rect *shieldRect, Point offsetPt);
pascal void ShowCursor      (void);
pascal void Show_Cursor     (Cursors cursorKind);
```

Displaying Animated Cursors

```
pascal void RotateCursor    (long counter);
pascal void SpinCursor      (short increment);
```

Assembly-Language Summary

Data Structures

Cursor Data Structure

0	`data`	32 bytes	cursor image
32	`mask`	32 bytes	cursor mask
64	`hotSpot`	long	point aligned with mouse

Color Cursor Data Structure

0	`crsrType`	word	type of cursor
2	`crsrMap`	long	the cursor's `PixMap` record
6	`crsrData`	long	cursor's data
10	`crsrXData`	long	expanded cursor data
14	`crsrXValid`	word	depth of expanded data (0 if none)
16	`crsrXHandle`	long	handle for future use
20	`crsr1Data`	16 words	1-bit data
52	`crsrMask`	16 words	1-bit mask
84	`crsrHotSpot`	long	hot spot for cursor
88	`crsrXTable`	long	table ID for expanded data
92	`crsrID`	long	ID for cursor
96	`crsrRec`	long	size of cursor save area

Global Variables

`arrow`	The standard arrow cursor.