# Technote 1196

## Cursor Components

---

**CONTENTS**



---

This Technote describes the building and use of cursor components. Cursor components allow you to build custom color cursors completely under your control. Unlike previous color cursors on the Macintosh, these components allow much more flexibility.

Cursors created utilizing this component are pretty unlimited. They could be extra large, thousands or millions of colors, transparent, situationally intelligent, and/or animated. The design is contrained only by your imagination.

Cursor components are built in two parts. The main part is the component itself, which can be built by expanding on the sample code sections provided in this Technote. The second part is the application that instantiates and controls the cursor. The second half of this technote will provide guidance on ways to do this. The last section of this technote will address cursor flicker and present a solution to the problem.

# Prerequisites and Resources

The Cursor Component SDK provides a sample application from which most of this Technotes code samples are taken. You should study this application and its cursor components carefully. These are very good starting points.

When building an application, there will be two parts (at least): the application itself and the cursor component(s). The simplest way to put together the components is as code resources. To this end, the SDK sample projects provide a good starting point. (Duplicating the cursor component projects and working from there would seem to be the simplest approach.) The application is much simpler. A short study of the cursor code shows that it is very portable, and can be worked into almost any existing application.

Developers will need Universal Interfaces 3.3 to develop cursor components. This is due to new functionality that has been added to QuickDraw headers and interfaces to support cursor components.

One last note: cursor components are available only with Mac OS 9.0 and later; therefore, you should always have a fallback if you plan deployment on systems prior to Mac OS 9.0. Additionally, currently cursor components are not supported under Mac OS X and Carbon. This may change in the future, and this Note will be updated as necessary.

Back to top

# Limitations

Currently, on Mac OS 9, Cursor Components have a few limitations. Due to ATI RAGE 128 video hardware architecture the cursor can exhibit flicker. This flicker varies with different monitor settings and from system to system. See the Fighting Flicker for more information of preventing this problem.

Additionally, Cursor Components do not function with PowerBooks due to adverse interactions with the "mouse trails" feature. There is no workaround, although this may be fixed in the future.

Finally, all cursor drawing and erasing happens at interrupt time. (This will be discussed later in this Technote.) Thus, you should use particular care in designing the drawing and erasing portion of the component, relying only on interrupt-safe functions for these sections. The Interrupt-Safe Routines Technote is a good source for this information.

Back to top

# Building Components

The component is built as a code resource. The main parts are the actual cursor code file and the resource data. In most cases, image data is stored in the resource, while behavior is defined by the code file.

The cursor component API consists two basic sections. The first section, the component manager, can be adopted straight from the code below and provides one main and four basic functions. This section handles the main function, dispatching, opening and closing the component, and responding to version information requests. The code provided will suit most needs. Be sure to look at the allocation of the `CursorGlobalsRecord`. It should be adapted to the specific needs of your cursor component. Normally, you can keep the standard return of `TRUE` for all the cases in the `ComponentCanDo` functions, but if you do not support all cases you should return `FALSE` for those that you do not. An example implementation of the component manager section is shown below. As stated previously, this section is fairly straightforward and the code below can be utilized with very little modification.

```
struct CursorGlobalsRecord
{
    ComponentInstance       self;
    Rect                    bounds;
    Point                   hotSpot;
    UInt32                  animIndex;
    UInt32                  deviceCount;
    CursorDeviceRecord      deviceList[12]; // fixed 12 displays max
};

typedef struct CursorGlobalsRecord      CursorGlobalsRecord;
typedef CursorGlobalsRecord *           CursorGlobalsPtr;
typedef CursorGlobalsPtr *              CursorGlobalsHnd;

// ---------------------------------------------------------------------

// Routine descriptors
RoutineDescriptor   ComponentCanDoRD =
    BUILD_ROUTINE_DESCRIPTOR (uppCallComponentCanDoProcInfo,
                              (ProcPtr)ComponentCanDo);
RoutineDescriptor   ComponentGetVersionRD =
    BUILD_ROUTINE_DESCRIPTOR (uppCallComponentVersionProcInfo,
                              (ProcPtr)ComponentGetVersion);
// Describe the main entry
RoutineDescriptor   mainRD =
    BUILD_ROUTINE_DESCRIPTOR (uppComponentRoutineProcInfo, main);

pascal ComponentResult
main
(ComponentParameters *params,char **storage)
{
    ComponentResult result = noErr;

    // If the selector is less than zero, it's a Component manager selector.
    if (params->what < 0)
    {
        switch (params->what)
        {
            case kComponentOpenSelect:
                return CallWithStoragePI(storage,
                                         params,
                                         ComponentOpen,
                                         uppCallComponentOpenProcInfo);

            case kComponentCloseSelect:
                return CallWithStoragePI(storage,
                                         params,
                                         ComponentClose,
                                         uppCallComponentCloseProcInfo);

            case kComponentCanDoSelect:
                return CallComponentFunction(params,
                                             &ComponentCanDoRD);

            case kComponentVersionSelect:
                return CallComponentFunction(params,
                                             &ComponentGetVersionRD);

            default:
                return noErr;
```

```
        }
    }

    switch (params->what)
    {
        case kCursorComponentInit:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorInitialize,
                                     uppCursorInitProcInfo);

        case kCursorComponentGetInfo:
            return CallWithStoragePI(storage,
                                     params,
                                     GetCursorInfo,
                                     uppGetCursorInfoProcInfo);

        case kCursorComponentSetData:
            return CallWithStoragePI(storage,
                                     params,
                                     SetCursorData,
                                     uppSetCursorDataProcInfo);

        case kCursorComponentReconfigure:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorReconfigure,
                                     uppCursorReconfigureProcInfo);

        case kCursorComponentDraw:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorDraw,
                                     uppCursorDrawProcInfo);

        case kCursorComponentErase:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorErase,
                                     uppCursorEraseProcInfo);

        case kCursorComponentMove:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorMove,
                                     uppCursorMoveProcInfo);

        case kCursorComponentAnimate:
            return CallWithStoragePI(storage,
                                     params,
                                     CursorAnimate,
                                               uppCursorAnimateProcInfo);

        default:
            result = unimpErr;
            break;
    }

    return result;

}
```

```
pascal ComponentResult
ComponentOpen
(CursorGlobalsHnd storage, ComponentInstance self)
{
#pragma unused (storage)
    ComponentResult result = noErr;

    storage =
        (CursorGlobalsHnd)NewHandleClear(sizeof(CursorGlobalsRecord));
    if (storage != nil )
    {
        SetComponentInstanceStorage(self, (Handle)storage);

        // keep an instance of ourself
        (*storage)->self = self;

    }
    else
        result = MemError();

    return result;
}

pascal ComponentResult
ComponentClose
(CursorGlobalsHnd storage, ComponentInstance self)
{
    if (storage != nil)
    {
        DisposeDeviceRecords(storage);
        DisposeHandle((Handle)storage);
    }

    return(noErr);
}

pascal ComponentResult
ComponentCanDo
(short selector)
{
    switch (selector)
    {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
        case kComponentCanDoSelect:
        case kComponentVersionSelect:
        case kCursorComponentInit:
        case kCursorComponentGetInfo:
        case kCursorComponentSetData:
        case kCursorComponentReconfigure:
        case kCursorComponentDraw:
        case kCursorComponentErase:
        case kCursorComponentMove:
            return true;

        default:
            return false;
    }
}

/* Version information */
#define CURSOR_COMPONENT_REV       1
```

```
#define kCursorComponentVersion    1

pascal ComponentResult
ComponentGetVersion
(void)
{
    // Cursor Component interface version in hi word,
    // code rev in lo word
    return ((kCursorComponentVersion << 16) | CURSOR_COMPONENT_REV);
}
```

The second section of the component is the cursor component functions. This consists of eight basic functions that handle the main processing, drawing and erasing of your cursor. We will cover each functional area in detail, but you are advised to look closely at the examples provided for basic structure and functionality.

## CursorInitialize

This function should create a cursor image, mask and save under buffers, and other info for each device. It will be called once when the component is opened and again when display devices are reconfigured or removed. The function `CreateDeviceRecords`, included in the sample and shown below, shows an example of handling the tasks of creating the image, mask and save under buffers for each device. This is important since cursors can span multiple devices, and devices can have different color depths resulting in different buffer pixel formats. If the buffers are successfully allocated, `CreateDeviceRecords` calls back to `SetCursorData` to set up the new cursors. Notice also the call to `InitializeCursorFlush`. This is designed to correct a flicker condition that may occur in some system configurations. See the Fighting Flicker for more information. Below is an example of a basic implementation of `CursorInitialize` and `CreateDeviceRecords`.

```
pascal ComponentResult
CursorInitialize
(CursorGlobalsHnd storage)
{
    ComponentResult result;
    Rect            theRect = {0, 0, 16, 16};
    Point           hotSpot = {0, 0};

    if (storage != nil)
    {
        result = InitializeCursorFlush();
        if (result == noErr)
        {

            (*storage)->bounds          = theRect;
            (*storage)->hotSpot         = hotSpot;
            (*storage)->deviceCount     = 0;
            result = CreateDeviceRecords(storage);
        }
    }
    return result;
}

static OSErr
CreateDeviceRecords
(CursorGlobalsHnd cursorGlobals)
{
    OSErr               err;
```

```
GDHandle            theDevice;
GWorldPtr           theWorld;
PrivateCursorData   data;
UInt32              counter;
SInt16              depth;
Rect                bounds;
short               index;

bounds = (*cursorGlobals)->bounds;

// start with the first device
theDevice = DMGetFirstScreenDevice(true);
while (theDevice != nil)
{
    counter = (*cursorGlobals)->deviceCount;
    depth = (*(*theDevice)->gdPMap)->pixelSize;

    (*cursorGlobals)->deviceList[counter].device = nil;
    (*cursorGlobals)->deviceList[counter].saveUnder = nil;

    for (index = 0; index < kNumberOfAnimations; index++)
    {
        (*cursorGlobals)->deviceList[counter].cursorImage[index] = nil;
        // create GWorlds to hold images of cursor expanded to the bitdepth
        // of this device

        err = CreateImageWorld(&theWorld, depth, &bounds, theDevice);
        if (err == noErr)
           (*cursorGlobals)->deviceList[counter].cursorImage[index]=
                theWorld;
    }

    if (err == noErr)
    {
        // create a GWorld to hold mask
        err = CreateImageWorld(&theWorld, depth, &bounds, theDevice);
        if (err == noErr && theWorld != nil)
        {
            (*cursorGlobals)->deviceList[counter].maskImage = theWorld;

            // create a GWorld to hold saved screen image
            err = CreateImageWorld(&theWorld, depth, &bounds, theDevice);
            if (err == noErr && theWorld != nil)
            {
                (*cursorGlobals)->deviceList[counter].saveUnder = theWorld;
                (*cursorGlobals)->deviceList[counter].device = theDevice;

                // increment number of instances counter
                (*cursorGlobals)->deviceCount++;
            }
        }
    }
    theDevice = DMGetNextScreenDevice(theDevice, true);
}

if (err != noErr)
{
    for (index = 0; index < kNumberOfAnimations; index++)
    {
        if ((*cursorGlobals)->deviceList[counter].cursorImage[index] != nil)
            DisposeGWorld
                ((*cursorGlobals)->deviceList[counter].cursorImage[index]);
```

```
        }

        if ((*cursorGlobals)->deviceList[counter].maskImage != nil)
            DisposeGWorld((*cursorGlobals)->deviceList[counter].maskImage);

        if ((*cursorGlobals)->deviceList[counter].saveUnder != nil)
            DisposeGWorld((*cursorGlobals)->deviceList[counter].saveUnder);
    }
    else
    {
        // fill out the cursor data
        err = SetCursorData(cursorGlobals, &data);
    }
    return err;
}
```

## GetCursorInfo

This function fills out the `CursorInfo` structure passed in. The structure pointer should be checked for validity and filled with the appropriate information. The version will be `kCursorComponentsVersion`. The capabilities field can either be 0 for minimal capabilities or `cursorDoesAnimate` to indicate that the cursor will be animating (i.e., changing image data on per-frame basis). `animateDuration` should be set to the number of ticks between frames. At this time only constant-rate animation is supported (i.e., each frame has the same duration). The last two fields, `bounds` and `hotspot`, should be appropriate for the cursor and must be zero based. Remember that you will be drawing the entire bounds at interrupt time. Cursors that are too large will cause slow overall machine performance while they are displayed. An example of `GetCursorInfo` is shown below.

```
pascal ComponentResult
GetCursorInfo
(CursorGlobalsHnd storage, CursorInfo *info)
{
    OSErr       err = noErr;

    if (info != nil)
    {
        info->version               = kCursorComponentsVersion;
        // use zero for minimal capabilities
        info->capabilities          = cursorDoesAnimate
        // number of ticks between animations (zero for none)
        info->animateDuration       = 15;
        info->bounds                = (*storage)->bounds;
        info->hotspot               = (*storage)->hotSpot;
    }
    else
        err = paramErr;
    return err;
}
```

## SetCursorData

This function enables an application to pass data to the cursor component. The first parameter is a handle to the `CursorGlobalsRecord`. The second parameter is a pointer to private data that is passed in by the application. This function's implementation can range from nothing to as complex as needed. The following example implementation shows using `SetCursorData` to change the cursor image via pictures from a resource file.

```
pascal ComponentResult
SetCursorData
(CursorGlobalsHnd storage, PrivateCursorPtr data)
{
    ComponentResult     result = -1;
    PicHandle           cursorPict;
    PicHandle           maskPict;
    GWorldPtr           imageWorld;
    GWorldPtr           oldPort;
    GDHandle            oldGD;
    Rect                theRect;
    short               resFile;
    short               c;
    short               index;

    // hide the cursor since we are going to change it
    HideCursor();

    // wack the values to those known by cursor so demo app works
    data->pictureID = kCursorPictureID;
    data->hotSpot.h = (short)(((*storage)->bounds.right -
                          (*storage)->bounds.left) / 2);
    data->hotSpot.v = (short)(((*storage)->bounds.bottom -
                          (*storage)->bounds.top) / 2);

    resFile = OpenComponentResFile((Component)(*storage)->self);
    if (resFile != -1)
    {
        cursorPict = GetPicture((short)data->pictureID);
        maskPict = GetPicture((short)(data->pictureID + kNumberOfAnimations));
        if (cursorPict != nil && maskPict != nil)
        {
            theRect = (*storage)->bounds;
            GetGWorld(&oldPort, &oldGD);

            for (c = 0; c < (*storage)->deviceCount; c++)
            {
                for (index = 0; index < kNumberOfAnimations; index++)
                {
                    imageWorld = (*storage)->deviceList[c].cursorImage[index];
                    cursorPict = GetPicture((short)(data->pictureID + index));
                    if (imageWorld != nil && cursorPict != nil)
                    {
                        SetGWorld(imageWorld, nil);
                        DrawPicture(cursorPict, &theRect);
                    }
                }
                imageWorld = (*storage)->deviceList[c].maskImage;
                if (imageWorld != nil)
                {
                    SetGWorld(imageWorld, nil);
```

```
                    DrawPicture(maskPict, &theRect);
                }
            }
            SetPort((GrafPtr)oldPort);
            SetGDevice(oldGD);

            (*storage)->hotSpot = data->hotSpot;

            // Now tell Quickdraw we changed
            result = CursorComponentChanged((*storage)->self);
        }
        CloseComponentResFile(resFile);
    }
    // show the new cursor
    ShowCursor();

    return result;
}
```

## CursorReconfigure

This function is called when the graphics environment has changed. The component should now rebuild its cursors (if required) to correspond to the new environment. A simple example implementation is below. This example calls `DisposeDeviceRecords`, also shown below, which is the de-allocation function for `CreateDeviceRecords` (shown above).

```
pascal ComponentResult
CursorReconfigure
(CursorGlobalsHnd storage)
{
    ComponentResult     result;

    if (storage != nil)
    {
        // dispose of old device records if they exist
        if ((*storage)->deviceCount > 0)
            DisposeDeviceRecords(storage);

        result = CreateDeviceRecords(storage);
    }
    return result;
}

static void
DisposeDeviceRecords
(CursorGlobalsHnd cursorGlobals)
{
    SInt16          index;
    SInt16          cursorIndex;

    for (index = 0; index < (*cursorGlobals)->deviceCount; index++)
    {
        (*cursorGlobals)->deviceList[index].device = nil;

        for (cursorIndex = 0; cursorIndex < kNumberOfAnimations; cursorIndex++)
        {
        if ((*cursorGlobals)->deviceList[index].cursorImage[cursorIndex] != nil)
                DisposeGWorld
```

```
                    ((*cursorGlobals)->deviceList[index].cursorImage[cursorIndex]);
        }
        if ((*cursorGlobals)->deviceList[index].saveUnder != nil)
        {
            DisposeGWorld((*cursorGlobals)->deviceList[index].saveUnder);
            (*cursorGlobals)->deviceList[index].saveUnder = nil;
        }
    }
    (*cursorGlobals)->deviceCount = 0;
}
```

## CursorDraw

This is the core of the cursor component, and it will be called every time the cursor is drawn for every display that the cursor spans. Additionally, since cursors are drawn at interrupt time, this function must not move memory (call Memory Manager routines). The first parameter, as usual, is a handle to `CursorGlobalsRecord`. Following this is a `GDHandle` for the `GDevice` on which to draw the cursor. The component should use this information to find the correct cursor images to draw. This is handled in the example by the `FindDeviceRecordIndex` function. This function, like the others in this example, works together with `FindDeviceRecordIndex` locating the specific index that corresponds to the cursor image data for the device passed in. The component developer is free to implement this functionality any way you prefer. The last parameter is the point at which to draw the cursor (in locate coordinates). This can be negative if the origin of the cursor is on an adjacent display.

Once the cursor data is found and you have the display and point, the cursor needs to be drawn. It is key to note that this entire drawing sequence occurs at interrupt time; it should be quick and use only interrupt-safe routines. Additionally, the plotted cursor must be clipped to the appropriate device. Lastly, the plotting code absolutely must save the data underneath the cursor image prior to plotting the new one. This can be done all at once, or pixel-by-pixel, as shown in the example. `PlotMaskedImageWithSave` is shown below as an example of how to plot and save the cursor data. The last call in this routine, `CursorFlush`, is required to previously mentioned flicker condition that can occur in cursor components. See the Fighting Flicker section for a full explanation.

```
pascal ComponentResult
CursorDraw
(CursorGlobalsHnd storage, GDHandle device, Point *position)
{
    SInt16      index;

    index = FindDeviceRecordIndex(storage, device);
    if (index != -1)
    {
        PlotMaskedImageWithSave (device, *position,
            (*storage)->deviceList[index].cursorImage
                [(*storage)->animIndex]->portPixMap,
            (*storage)->deviceList[index].maskImage->portPixMap,
            (*storage)->deviceList[index].saveUnder->portPixMap);
    }
    return noErr;
}

static SInt16
FindDeviceRecordIndex
(CursorGlobalsHnd cursorGlobals, GDHandle findDevice)
{
    UInt16          index;
```

```
    for (index = 0; index < (*cursorGlobals)->deviceCount; index++)
    {
        if ((*cursorGlobals)->deviceList[index].device == findDevice)
            return index;
    }
    // didn't find device
    return -1;
}

static void
PlotMaskedImageWithSave
(GDHandle device, Point mouse, PixMapHandle srcPix,
          PixMapHandle mskPix, PixMapHandle savePix)
{
    Rect            theRect;
    UInt32          h;
    UInt32          v;
    UInt32          srcHStart;
    UInt32          srcVStart;
    UInt32          offRowBytes;
    UInt32          dstRowBytes;
    SInt32          srcHeight;
    SInt32          srcWidth;
    UInt32          deviceHeight;
    UInt32          deviceWidth;
    UInt32          offOffset;
    SInt32          xOffset;
    SInt32          yOffset;
    UInt16          depth;
    Ptr             srcBase;
    Ptr             mskBase;
    Ptr             dstBase;
    Ptr             saveBase;
    Ptr             srcData;
    Ptr             mskData;
    Ptr             dstData;
    Ptr             saveData;

    // get base of src, msk, and save
    srcBase = (*srcPix)->baseAddr;
    mskBase = (*mskPix)->baseAddr;
    saveBase = (*savePix)->baseAddr;

    offRowBytes = (*srcPix)->rowBytes & 0x7FFF;

    // get base and rowbytes of destination
    dstBase = (*(*device)->gdPMap)->baseAddr;
    dstRowBytes = (*(*device)->gdPMap)->rowBytes & 0x7FFF;
    depth = (*(*device)->gdPMap)->pixelSize;

    theRect = (*device)->gdRect;
    deviceWidth = theRect.right - theRect.left;
    deviceHeight = theRect.bottom - theRect.top;

    theRect = (*srcPix)->bounds;
    srcWidth = theRect.right - theRect.left;
    srcHeight = theRect.bottom - theRect.top;

    xOffset = mouse.h;
    yOffset = mouse.v;
    srcHStart = 0;
    srcVStart = 0;
```

```
// trim h
if (xOffset < 0)
{
    srcWidth -= -xOffset;
    srcHStart = -xOffset;
    xOffset = 0;
}
else
{
    if ((xOffset + srcWidth) > deviceWidth)
        srcWidth = deviceWidth - xOffset;
}

// trim y
if (yOffset < 0)
{
    srcHeight -= -yOffset;
    srcVStart = -yOffset;
    yOffset = 0;
}
else
{
    if ((yOffset + srcHeight) > deviceHeight)
        srcHeight = deviceHeight - yOffset;
}

switch (depth)
{
    case 1:
    case 2:
    case 4:
    case 8:
        DebugStr("\pDon't handle indexed yet");
        break;

    case 16:
        // need to align this stuff
        {
            UInt16          *srcPixel;
            UInt16          *mskPixel;
            UInt16          *dstPixel;
            UInt16          *savePixel;

            offOffset = (srcVStart * offRowBytes) + (srcHStart << 1);

            srcData = srcBase + offOffset;
            mskData = mskBase + offOffset;
            saveData = saveBase + offOffset;
            dstData = dstBase + (yOffset * dstRowBytes) + (xOffset << 1);

            for (v = 0; v < srcHeight; v++)
            {
                srcPixel = (UInt16 *)srcData;
                mskPixel = (UInt16 *)mskData;
                dstPixel = (UInt16 *)dstData;
                savePixel = (UInt16 *)saveData;

                for (h = 0; h < srcWidth; h++)
                {
                    if (*mskPixel++ == 0x00)
                    {
```

```
                        *savePixel = *dstPixel;
                        *dstPixel = *srcPixel;
                    }

                    dstPixel++;
                    srcPixel++;
                    savePixel++;
                }

                srcData += offRowBytes;
                mskData += offRowBytes;
                dstData += dstRowBytes;
                saveData += offRowBytes;
            }
        }
        break;

    case 32:
        {
            UInt32          *srcPixel;
            UInt32          *mskPixel;
            UInt32          *dstPixel;
            UInt32          *savePixel;

            offOffset = (srcVStart * offRowBytes) + (srcHStart << 2);

            srcData = srcBase + offOffset;
            mskData = mskBase + offOffset;
            saveData = saveBase + offOffset;
            dstData = dstBase + (yOffset * dstRowBytes) + (xOffset << 2);
            for (v = 0; v < srcHeight; v++)
            {
                srcPixel = (UInt32 *)srcData;
                mskPixel = (UInt32 *)mskData;
                dstPixel = (UInt32 *)dstData;
                savePixel = (UInt32 *)saveData;

                for (h = 0; h < srcWidth; h++)
                {
                    if (*mskPixel++ == 0x00)
                    {
                        *savePixel = *dstPixel;
                        *dstPixel = *srcPixel;
                    }
                    srcPixel++;
                    dstPixel++;
                    savePixel++;
                }

                srcData += offRowBytes;
                mskData += offRowBytes;
                saveData += offRowBytes;
                dstData += dstRowBytes;
            }
        }
        break;

    default:
        break;
    }
    CursorFlush(device);
}
```

## CursorErase

This function really is a drawing function. It just undoes the work done by `CursorDraw` by restoring the save screen image data. The parameters are the same as for `CursorDraw` and have the same caveats. The applicable save data must be found based on the `GDHandle` parameter. `RestoreMaskedImage` (as shown below) restores the clipped data back to the screen. As with other component functions, this must be interrupt safe.

```
pascal ComponentResult
CursorErase
(CursorGlobalsHnd storage, GDHandle device, Point *position)
{
    SInt16          index;

    index = FindDeviceRecordIndex(storage, device);
    if (index != -1)
    {
        RestoreMaskedImage(device, *position,
            (*storage)->deviceList[index].saveUnder->portPixMap,
            (*storage)->deviceList[index].maskImage->portPixMap);
    }
    return noErr;
}

static void
RestoreMaskedImage
(GDHandle device, Point mouse, PixMapHandle srcPix, PixMapHandle mskPix)
{
    Rect            theRect;
    UInt32          h;
    UInt32          v;
    UInt32          srcHStart;
    UInt32          srcVStart;
    UInt32          offRowBytes;
    UInt32          dstRowBytes;
    UInt32          offOffset;
    SInt32          srcHeight;
    SInt32          srcWidth;
    UInt32          deviceHeight;
    UInt32          deviceWidth;
    SInt32          xOffset;
    SInt32          yOffset;
    UInt16          depth;
    Ptr             srcBase;
    Ptr             dstBase;
    Ptr             mskBase;
    Ptr             srcData;
    Ptr             dstData;
    Ptr             mskData;

    // get base of src and mask
    srcBase = (*srcPix)->baseAddr;
    mskBase = (*mskPix)->baseAddr;

    offRowBytes = (*srcPix)->rowBytes & 0x7FFF;

    // get base and rowbytes of destination
```

```
dstBase = (*(*device)->gdPMap)->baseAddr;
dstRowBytes = (*(*device)->gdPMap)->rowBytes & 0x7FFF;
depth = (*(*device)->gdPMap)->pixelSize;

theRect = (*device)->gdRect;
deviceWidth = theRect.right - theRect.left;
deviceHeight = theRect.bottom - theRect.top;

theRect = (*srcPix)->bounds;
srcWidth = theRect.right - theRect.left;
srcHeight = theRect.bottom - theRect.top;

xOffset = mouse.h;
yOffset = mouse.v;
srcHStart = 0;
srcVStart = 0;

// trim h
if (xOffset < 0)
{
    srcWidth -= -xOffset;
    srcHStart = -xOffset;
    xOffset = 0;
}

if ((xOffset + srcWidth) > deviceWidth)
    srcWidth = deviceWidth - xOffset;

// trim y
if (yOffset < 0)
{
    srcHeight -= -yOffset;
    srcVStart = -yOffset;
    yOffset = 0;
}

if ((yOffset + srcHeight) > deviceHeight)
    srcHeight = deviceHeight - yOffset;

switch (depth)
{
    case 1:
    case 2:
    case 4:
    case 8:
        DebugStr("\pDon't handle indexed yet");
        break;

    case 16:
        // need to align this stuff
        {
            UInt16          *srcPixel;
            UInt16          *dstPixel;
            UInt16          *mskPixel;

            offOffset = (srcVStart * offRowBytes) + (srcHStart << 1);

            srcData = srcBase + offOffset;
            mskData = mskBase + offOffset;
            dstData = dstBase + (yOffset * dstRowBytes) + (xOffset << 1);

            for (v = 0; v < srcHeight; v++)
```

```
        {
            srcPixel = (UInt16 *)srcData;
            dstPixel = (UInt16 *)dstData;
            mskPixel = (UInt16 *)mskData;

            for (h = 0; h < srcWidth; h++)
            {
                if (*mskPixel == 0x00)
                {
                    *dstPixel = *srcPixel;
                }

                dstPixel++;
                mskPixel++;
                srcPixel++;
            }

            srcData += offRowBytes;
            mskData += offRowBytes;
            dstData += dstRowBytes;
        }
    }
    break;

case 32:
    {
        UInt32          *srcPixel;
        UInt32          *dstPixel;
        UInt32          *mskPixel;

        offOffset = (srcVStart * offRowBytes) + (srcHStart << 2);

        srcData = srcBase + offOffset;
        mskData = mskBase + offOffset;
        dstData = dstBase + (yOffset * dstRowBytes) + (xOffset << 2);

        for (v = 0; v < srcHeight; v++)
        {
            srcPixel = (UInt32 *)srcData;
            dstPixel = (UInt32 *)dstData;
            mskPixel = (UInt32 *)mskData;

            for (h = 0; h < srcWidth; h++)
            {
                if (*mskPixel == 0x00)
                {
                    *dstPixel = *srcPixel;
                }

                dstPixel++;
                mskPixel++;
                srcPixel++;
            }

            srcData += offRowBytes;
            mskData += offRowBytes;
            dstData += dstRowBytes;
        }
    }
    break;

default:
```

```
            break;
    }
}
```

## CursorMove

This function must both erase and restore the last image, and draw a new image. To this end, both the old position and new position are provided. Again, this all happens at interrupt time; the `GDHandle` must be used to determine the cursor data to restore and draw the screen and cursor image. All the caveats for the previously described plotting functions apply.

```
pascal ComponentResult
CursorMove
(CursorGlobalsHnd storage, GDHandle device, Point *lastPosition, Point *newPosition)
{
    SInt16          index;

    index = FindDeviceRecordIndex(storage, device);
    if (index != -1)
    {
        RestoreMaskedImage(device, *lastPosition,
            (*storage)->deviceList[index].saveUnder->portPixMap,
            (*storage)->deviceList[index].maskImage->portPixMap);
        PlotMaskedImageWithSave(device, *newPosition,
            (*storage)->deviceList[index].cursorImage
                    [(*storage)->animIndex]->portPixMap,
            (*storage)->deviceList[index].maskImage->portPixMap,
            (*storage)->deviceList[index].saveUnder->portPixMap);
    }
    return noErr;
}
```

## CursorAnimate

The last function in the component API is `CursorAnimate`. It is responsible for indexing through the cursor's various images. You will only need to provide this function if your component handles animation. The animate function is called once for each display. In the example case, we simply index through pre-rendered images so we care only about the first display. If your component renders each frame on the fly, you will most likely require notifications for each display. Once again, the specifics of the animate function are completely defined by your cursor implementation. Below is a simple example of a rotating animation.

```
pascal ComponentResult
CursorAnimate
(CursorGlobalsHnd storage, GDHandle device, Point *lastPosition)
{
#pragma unused (device, lastPosition)
    UInt32          index;

    if (device == GetMainDevice())
    {
        index = (*storage)->animIndex;
        index++;
        if (index > kNumberOfAnimations - 1)
            index = 0;
        (*storage)->animIndex = index;
    }
    return noErr;
}
```

## Resources

The cursor component compiled as a code resource needs a `'thng'` resource, which allows the application to identify the specific component it is accessing. Below is a simple example of a beach ball cursor's `'thng'` resource. Note the `kCursorSubType`: this will be used later to identify this particular cursor code resource. Beyond this, you are free to use resources to store any data required to build the cursor.

```
#define kCursorResID            129
#define kCursorName             "Beachball Cursor"
#define kCursorType             'curs'
#define kCursorSubType          'bbcc'
#define kCursorManufacturer     'appl'


resource 'thng' (kCursorResID, kCursorName, purgeable)
{
    kCursorType,
    kCursorSubType,
    kCursorManufacturer,
    0x80000020,
    kAnyComponentFlagsMask,
    'crco',
    kCursorResID,
    'STR ',
    kCursorResID + 5000,
    'STR ',
    kCursorResID + 5050,
    'ICON',
    kCursorResID,
    0x0,
    8,
    0,
    {   /* array ComponentPlatformInfo: 1 elements */
        /* [1] */
        0x80000000, 'crco', kCursorResID, platformPowerPC
    }
};
```

Back to top

# Using Components

Once the cursor component is created, an application must be created to instantiate it. The application interface for the component is fairly simple. The basic requirements are that the code resource is loaded, the cursor is instantiated by opening it, and then it is set. Once this occurs the application can pass data directly to the cursor. Finally, when it is done with it, close and unload it.

## Registering Components

First, we look at handling the code resource. The code below registers the cursors code resource, allowing it to be opened.

```
OSErr
RegisterCursorComponents
(void)
{
    OSErr       err = -1;

    if (RegisterComponentResourceFile (CurResFile(), 0))
        err = noErr;

    return err;
}
```

## Opening the Cursor

Once the component is registered, we can open the cursor. This is achieved by setting up a `ComponentDescription` to look for the cursor component. The keys here are the subtype and the manufacturer: ensure that these match your cursor's `'thng'` resource. Once this it completed, call `FindNextComponent` to find the next component that fits the description. If this returns without error, the component found can be used in the new QuickDraw function `OpenCursorComponent` to actually open the cursor. `OpenCursorComponent` is defined as follows:

```
 OSErr OpenCursorComponent (Component c, ComponentInstance * ci);
```

The `ComponentInstance` parameter should be set on return to a specific instance the cursor that can used with the other cursor component functions. This functionality can be seen in the example below, which takes a component subtype and attempts to open it, in turn setting the global `ComponentInstance` as applicable.

```
void
OpenTheCursor
(OSType subType)
{
    OSErr                       err;
    ComponentDescription        theDesc;
    Component                   comp = nil;
    Point                       hotspot;

    // set up the descriptor
    theDesc.componentType           = 'curs';
    theDesc.componentSubType        = subType;
    theDesc.componentManufacturer   = 'appl';
    theDesc.componentFlags          = 0L;
    theDesc.componentFlagsMask      = 0L;

    comp = FindNextComponent(0L, &theDesc);
    if (comp != nil)
    {
        err = OpenCursorComponent(comp, &gComp);
        if (err == noErr)
            gCurrentSubType = subType;

        else
            gComp = nil;

        if (err != noErr)
            HandleError(err, false);
    }
}
```

## Setting the Cursor

After the cursor is opened it can be set. This should be done when the application wants to display the particular cursor. To achieve this `SetCursorComponent` is call. The function is defined in QuickDraw.h as:

```
 OSErr SetCursorComponent (ComponentInstance ci);
```

This can be accomplished in the following simple manner.

```
void
SetTheCursor
(void)
{
    SetCursorComponent(gComp);
}
```

## Closing the Cursor

Once the application is done with a particular cursor (and absolutely before exit), it should be close with `CloseCursorComponent` defined as:

```
OSErr CloseCursorComponent (ComponentInstance ci);
```

Again, this is simply implemented in the sample by just checking for `NIL` and passing the global `ComponentInstance` to the function as shown below.

```
void
CloseTheCursor
(void)
{
    if (gComp != nil)
    {
        CloseCursorComponent(gComp);
        gComp = nil;
    }
}
```

## Modifying the Cursor

Cursor components possess the `CursorComponentSetData` function, which allows modification of the cursor. This can be used to change frame data, set a state, or just about anything else you can imagine. The set data function is defined as:

```
OSErr CursorComponentSetData (ComponentInstance ci, long data);
```

This allows passing either direct data or a pointer to the cursor itself. An example of utilizing can also be found in the example application's `SetCursorPictureID` function that sets the bitmap for the current cursor.

```
OSErr
SetCursorPictureID
(short pictureID, Point *hotspot)
{
    OSErr                       err = noErr;
    PrivateCursorData           data;
    static  short               lastID = -99;

    // return if image is already set
    if (lastID == pictureID) return err;

    // only set image if cursor is bitmap cursor...
    if (gCurrentSubType == kBitMapSubType)
    {
        data.pictureID = (long)pictureID;
        data.hotSpot = *hotspot;

        err = CursorComponentSetData((ComponentInstance)gComp, (long)&data);
        if (err == noErr)
            lastID = pictureID;
    }
    return err;
}
```

Basically `CursorComponentSetData` provides an application-defined conduit to the cursor component, but does not alter any outside characteristics. It is worthwhile to note if the call to `CursorComponentSetData` modifies that appearance of the cursor, `CursorComponentSetData` should call `CursorComponentChanged`. This will notify QuickDraw that the appearance and/or other info about the cursor has changed. QuickDraw will re-interrogate the cursor for hotspot, bounds, etc., and redraw it.

## Miscellaneous

If you are changing cursors (not just changing images) you should ensure that you open the new cursor before closing the old one. This will prevent the cursor from reverting to the arrow during the change. Here's some example code to do this:

```
void
OpenNewCloseOld
(OSType subType)
{
    ComponentInstance   oldCursor;

    // save the current cursor
    oldCursor = gComp;

    // open the new one
    OpenTheCursor(subType);

    if (oldCursor != nil)
    {
        CloseCursorComponent(oldCursor);
    }
}
```

Back to top

# Fighting Flicker

Cursor flicker can occur when using color cursors under some circumstances. RAGE 128-based systems may not flush their internal cache on every cursor redraw, causing some or all of the cursor to flicker. This problem can be corrected by utilizing the included CursorFlush library. The CursorFlush header file defines two routines, `InitializeCursorFlush` and `CursorFlush`. The former should be called in the `CursorInitialize` function to set up the flush library. The later should be called when drawing is complete to ensure the video card correctly displays the cursor image without flicker.

Back to top

# Summary

Obviously, Cursor Components provide a useful and greatly expanded human interface element. At first, they may seem difficult to implement. By reading this Technote and studying the code provided in the SDK, you can see the implementation is straightforward, and the resources in the SDK provide a good start. Freeform color cursors are here today on the Mac OS; it is only your imagination that limits their implementation. Go forth and build some awesome cursors.

## Further References

- Cursor Component SDK
- Technote 1104: Interrupt-Safe Routines
- Inside Macintosh: Imaging With QuickDraw
- Technote web site

Back to top

## Downloadables

 Acrobat version of this Note (how many K?)

Back to top

---

**To contact us, please use the Contact Us page.**
**Updated: 17-April-2000**