

This chapter discusses the attributes of image compressor components and the functional interfaces these components must support. An **image compressor component** is a code resource that provides compression or decompression services for image data. Throughout this chapter, the term *image compressor component* is used to describe both compressor and decompressor components.

Note

The information in this chapter is intended for developers of image compressor components. Application developers normally do not need to be familiar with this material to use the Image Compression Manager. ♦

This chapter has been divided into the following sections:

- “About Image Compressor Components” presents general information about image compressor components.
- “Using Image Compressor Components” discusses how the Image Compression Manager uses image compressor components to compress and decompress images.
- “Image Compressor Components Reference” describes the data structures used by the Image Compression Manager to communicate with image compressor components. It also provides a comprehensive reference to the functions that your image compressor component must support.
- “Summary of Image Compressor Components” presents a summary of image compressor components in C and in Pascal.

If you are developing an image compressor component, you should read all the material in this chapter. In addition, you should read the appropriate sections of the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

About Image Compressor Components

Image compressor components are registered by the Component Manager, and they present a standard interface to the Image Compression Manager (see “Functions” beginning on page 4-53 for a detailed description of the functions that image compressor components must provide). An image compressor component can be a systemwide resource, or it can be local to a particular application.

Applications never communicate directly with these components. Applications request compression and decompression services by issuing the appropriate Image Compression Manager functions. The Image Compression Manager then performs its necessary processing before invoking the component. Of course, an application could install its own image compressor component. However, any interaction between the application and the component is still managed by the Image Compression Manager.

Image Compressor Components

The Image Compression Manager knows about two types of image compressor components. Components that can compress image data carry a component type of 'imco' and are called *image compressors*. Components that can decompress images have a component type of 'imdc' and are called *image decompressors*.

```
#define compressorComponentType 'imco'      /* compressor component
                                              type */

#define decompressorComponentType 'imdc'    /* decompressor
                                              component type */
```

The value of the component subtype indicates the compression algorithm supported by the component. For example, the graphics compressor has the component subtype 'cvid'. (A **component subtype** is an element in the classification hierarchy used by the Component Manager to define the services provided by a component.) All compressor components with the same subtype must be able to handle the same format of compressed data. During decompression, a component should handle all variations of the data specified for a subtype. While compressing an image, a compressor must not produce data that decompressors of the same subtype cannot handle during decompression.

The Image Compression Manager provides a set of utility functions for compressor components. These functions allow compressors and decompressors to create custom color lookup tables, among other things. For a complete description of these utility functions, along with the functions that must be supported by compressor components, see “Image Compression Manager Utility Functions,” which begins on page 4-65.

The Image Compression Manager defines four callback functions that may be provided to compressors and decompressors by applications. These callback functions are data-loading functions, data-unloading functions, completion functions, and progress functions. Data-loading functions and data-unloading functions support spooling of compressed data. Completion functions allow components to report that asynchronous operations have completed. Progress functions provide a mechanism for components to report their progress toward completing an operation. For more information about these callback functions, see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime*.

Banding and Extending Images

QuickTime handles images in **bands**, which are horizontal strips of an image. Bands allow large images to be accommodated even if the entire image cannot fit into memory. The Image Compression Manager calls the image compressor component once for each band as the image is compressed or decompressed.

The Image Compression Manager determines the height of a band based on the amount of available memory and the `bandMin` and `bandInc` parameters provided by the compressor component in the compressor capability structure (described in “The Compressor Capability Structure” beginning on page 4-35). The `bandMin` field specifies the minimum band height supported by a decompressor component. By providing a minimum height, decompressor components that operate on blocks of pixels can operate more efficiently since the minimum height ensures that a band has at least one row of pixel blocks. The `bandInc` field specifies the increment in pixels by which the height of a band is increased above the minimum when sufficient memory is available. This specification allows easier processing by ensuring that a band is an integral number of rows of blocks. The larger these two parameters, the more memory is required for the band buffer, which may limit the size of images used with a given amount of memory. By specifying a minimum height that is the size of the image, the compressor component can indicate that it cannot handle banded images. However, the specification of a full size is not recommended unless required by the compression format, since it requires large amounts of memory for large images.

For decompressing sequences of images with temporal compression, the Image Compression Manager always allocates the band to include the full image. The entire image must be available whenever the screen needs updating and the current frame does not have information for all pixels. The entire image is needed to make the comparison with the previous frame.

The depth of the band is determined by the Image Compression Manager and the `wantedPixelSize` field of the compressor capability structure (described on page 4-35). That field is filled in by the image compressor component’s `CDPreCompress` or `CDPreDecompress` function (described on page 4-62 and page 4-63, respectively). The Image Compression Manager requests the depth that it decides is best for the image, and the compressor component can return the `wantedPixelSize` field set to that depth or another appropriate depth if the compressor cannot handle the one requested.

The width of the band is usually the width of the image, but the compressor can extend the measurement if it cannot easily handle partial blocks of pixels at the edge of the image. For compression operations, the Image Compression Manager sets the extra pixels added to the right edge of the band to the same value as the last pixel in each scan line. For decompression operations, the Image Compression Manager ignores the pixels that were added to the right edge for the extension.

Image compressor components can also use extension for the height of the last (or the only) band in the image (the other bands should always be an integral multiple of the `bandInc` field set by the decompressor component). The extended pixels are added to the bottom of the band. For compression operations, the added pixels have the same value as the pixel at the same location in the last scan line of the image. For decompression operations, the added pixels are ignored. If an image compressor component does not want to deal with partial blocks of pixels, either horizontally or vertically, it can use this extension technique. However, it would be more efficient for the compressor to handle those blocks itself.

Spooling of Compressed Data

If available memory is insufficient to hold the entire image that is being compressed or decompressed, the image compressor component must call data-loading or data-unloading functions to spool—that is, read or write the data from storage in stages. The calling application indicates this in the data-loading or data-unloading structure, as described in the following sections.

Data Loading

Decompressor components use data loading. The data buffer still exists when the calling application supplies a data-loading function; however, the data buffer holds only part of the data and you must use the data-loading function to load the remaining data into this buffer. The `bufferSize` parameter of the decompression parameters structure (described on page 4-46) indicates the size of the data buffer.

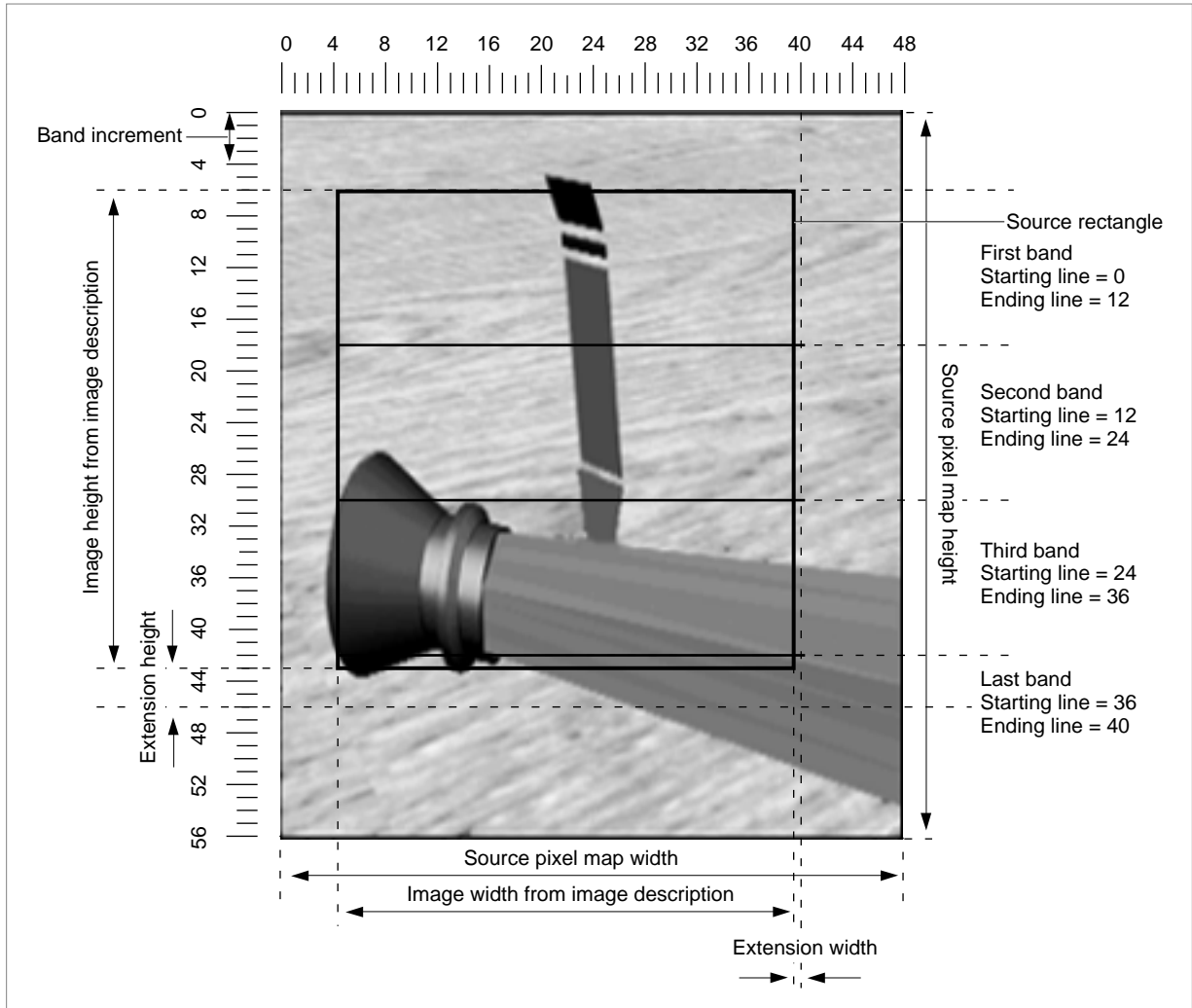
To use the data-loading function, the decompressor component calls it with the pointer to the current position in the data buffer as a parameter. The decompressor specifies the number of bytes it needs (this number must be less than or equal to the size of the data buffer). The data-loading function fills in the data buffer with the number of bytes requested and may adjust the pointer as necessary to remove some of the used data and make room for new data.

If the decompressor component needs to skip data in the compressed stream or go back to data earlier in the stream, the decompressor should call the data-loading function with a `nil` pointer (instead of the pointer to the data buffer of the data-loading function) and with the `size` parameter set to the number of bytes that the decompressor wants to skip relative to the current position in the stream. A positive number seeks forward and a negative one seeks backward. To ensure that the position in the stream is known by the data-loading function, the decompressor should call the function before specifying a seek operation with an actual pointer to the current position in the data buffer and a 0 byte count. After the seek operation, the decompressor component should call the data-loading function again with the number of bytes needed from the new position to make sure the needed bytes are read into the buffer.

A decompressor component should not depend on the ability to skip backward in the data stream since not all applications are able to take advantage of this feature. The decompressor should check the error from the data-loading function during a seek operation and should not use the seek feature if an error code is returned. Seeking forward works in most situations; however, it may entail reading the data and throwing it out. Hence, seeking forward may not always be faster than reading the data.

Figure 4-1 shows several image bands and their measurements.

Figure 4-1 Image bands and their measurements



Data Unloading

Data-unloading functions are used by compressor components when there is insufficient memory to hold the buffer for the compressed data produced by the compressor component. The compressor component needs to use a data-unloading function if the `flushProcRecord` field in the compression parameters structure is not `nil`. (For details on the compression parameters structure, see page 4-40). A data buffer is provided even if the data-unloading function is present, and it should be used to hold the data to be unloaded by the data-unloading function. The size of the data buffer is indicated by the `bufferSize` field in the parameters.

Image Compressor Components

To use the data-unloading function, the compressor fills the data buffer with as much data as possible (within the size limitations of the data buffer). The compressor component then calls the data-unloading function with a pointer to the start of the data buffer and the number of bytes written. The data-unloading function then unloads the data from the buffer. The compressor should then use the entire buffer for the next piece of data—and continue in this manner until all the data is unloaded.

If the compressor component needs to skip forward or backward in the data stream, it should call the data-unloading function with a `nil` data pointer, and the compressor should specify the number of bytes to seek relative to the current position in the `size` parameter. A positive number seeks forward and a negative one seeks backward. The compressor component should make sure that all data is unloaded from the buffer before commencing the seek operation. After the seek operation, the next data unloaded from the buffer with the data-unloading function is written starting at the new location. The new data overwrites any data previously written at that location in the data stream.

Not all applications support the ability to seek forward or backward with a data-unloading function. The compressor component should check the error result when performing such an operation.

Compressing or Decompressing Images Asynchronously

With the appropriate hardware, image compressor components can handle asynchronous compression and decompression of images using the `CDBandCompress` and `CDBandDecompress` functions, which are described on page 4-63 and page 4-64, respectively. *Asynchronous* refers to the fact that the compression or decompression hardware performs its operations while the Macintosh computer simultaneously continues its activities. For example, the Macintosh can read a movie for the next frame while the current frame is decompressed. The Image Compression Manager ensures that any asynchronous operation in progress is completed before starting the next operation.

If the Image Compression Manager wants the image compressor component to perform an operation asynchronously, then the `completionProcRecord` field in the compression or decompression parameters structure that the Image Compressor Manager sends to the image compressor component should be set to a nonzero value. If the value is `-1`, then the component should perform the operation asynchronously, but it does not need to call a completion function. If the value is not `nil` and not `-1`, then the component should perform the operation asynchronously, and it should call the completion function when the operation is done. For details on the compression parameters structure, see page 4-40. For more on the decompression parameters structure, see page 4-46.

To provide synchronization for the Image Compression Manager, an image compressor component provides the `CDCodecBusy` function (described on page 4-61). `CDCodecBusy` should always return 1 if an asynchronous operation is in progress; it should return 0 if there is no asynchronous operation in progress or if the image compressor component does not perform asynchronous operations. If the Image Compression Manager provided a completion function, the image compressor component must call the completion function as well.

IMPORTANT

If the Image Compression Manager provided a completion function, then the compressor component must call it; otherwise, the memory for that operation may become increasingly stranded in the system and difficult to deallocate. ▲

There are two distinct steps to an asynchronous compression or decompression operation. The first step depends on the source data, and the second step depends on the destination data.

- For a compression operation, the first step indicates when the compressor is finished with the pixels of the source image, and the second step specifies that the compressed data is fully written to memory.
- For a decompression operation, the first step is complete when the compressed data is read into the hardware or the decompressor's local buffers, and the second step is complete when all the pixels of the image have been written to the destination.

Depending on the design of the hardware used by your image compressor component, the two steps in the asynchronous operations may be independent of each other or tied together. To indicate to the completion function which steps have been completed, you use the `codecCompletionSource` and `CodecCompletionDest` flags for the first and second steps, respectively. If both parts of the asynchronous operation are completed together, the image compressor component can call the completion function once with both flags set. The memory used for each part of the operation remains valid and locked while asynchronous operations are in progress. It is the responsibility of image compressor components to make sure that they remain resident in RAM if virtual memory is active (this is only an issue for hardware image compressor components that perform direct memory access).

Progress Functions

Progress functions provide the calling application an indication of how much of an operation is complete and a way for the user to cancel an operation. If the `progressProcRecord` field is set either in the compression parameters structure or the decompression parameters structure, then the image compressor component should call the progress function as it performs the operation. The progress function is typically called once for each scan line or row of pixel blocks processed, and it returns a completion value that is the percentage of the band that is complete, represented as a fixed-point number from 0 to 1.0.

If the result returned from a progress function is not 0, then the image compressor component should return as soon as possible (without completing the band that is being processed) with a return value of `codecAbortErr`.

Note

For efficiency, many image compressor components have a streamlined path used for cases that do not require data-loading, data-unloading, or progress functions, and a slower path that supports any or all these application-defined functions when required. ♦

Using Image Compressor Components

This section shows how to use compressors and decompressors in conjunction with the Image Compression Manager.

Performing Image Compression

This section describes what the Image Compression Manager does that affects compressors. It then provides sample code that shows how the compressor components prepare for image compression and how to compress an entire image or a horizontal band of an image.

When compressing an image, the Image Compression Manager performs three major tasks:

1. The Image Compression Manager first determines which compressor is best able to compress the image. To do so, the Image Compression Manager examines the source image as well as the parameters specified by the application. If the application requested a specific compressor, the Image Compression Manager uses that compressor (unless it is not installed, in which case the Image Compression Manager returns an error to the application). If the application did not request a compressor, the Image Compression Manager chooses the compressor that will do the best job. The Image Compression Manager collects the information it needs to choose a compressor by issuing the `CDPreCompress` request to each qualifying compressor (see page 4-62 for a detailed description of the `CDPreCompress` function).
2. If the chosen compressor can handle the image directly, the Image Compression Manager passes the request through to the compressor. The compressor then processes the image and returns the compressed data to the specified location.
3. If none of the compressors can handle it directly, the Image Compression Manager allocates an offscreen buffer and passes image bands to the compressor by issuing a `CDBandCompress` request. (For more on the `CDBandCompress` function, see page 4-63.) The compressor processes each band, accumulating the compressed data as it goes. When the image has been completely compressed, the Image Compression Manager returns control to the application.

Choosing a Compressor

Listing 4-1 on page 4-12 shows how the Image Compression Manager calls the `CDPreCompress` function before an image is compressed. The compressor component returns information about how it is able to compress the image to the Image Compression Manager, so that it can fit the destination data to the requirements of the compressor component. This information includes compressor capabilities for

- depth of input pixels
- minimum buffer band size

- band increment size
- extension width and height

When your compressor component is called with the `CDPreCompress` function (described on page 4-62), it can handle all aspects of the function itself, or only the most common ones. All image compressor components must handle at least one case.

Here is a list of some of the operations your compressor component can perform during compression. It describes parameters in the compression parameters structure (described on page 4-40) and indicates the operations that are required and which flags in the compressor capabilities flags field of the compressor capabilities structure (described on page 4-35) must be set to allow your compressor to handle them.

- **Depth conversion.** If your compressor component can compress from the pixel depth indicated by the `pixelSize` field (in the pixel map structure pointed to by the `srcPixmap` field of the compression parameters structure), it should set the `wantedPixelSize` field of the compressor capability structure to the same value. If it cannot handle that depth, it should specify the closest depth it can support in the `wantedPixelSize` field. The Image Compression Manager will convert the source image to that depth.
- **Extension.** If the format for the compressed data is block oriented, the compressor component can request that the Image Compression Manager allocate a buffer that is a multiple of the proper block size by setting the `extendWidth` and `extendHeight` parameters of the compressor capability structure. The new pixels are replicated from the left and bottom edges to fill the extended area. If your compressor can perform this extension itself, it should leave the `extendWidth` and `extendHeight` fields set to 0. In this case, the Image Compression Manager can avoid copying the source image to attain more efficient operation.
- **Pixel shifting.** For pixel sizes less than 8 bits per pixel, it may be necessary to shift the source pixels so that they are at an aligned address. If the `pixelSize` field of the source pixel map structure is less than 8, and your compressor component handles that depth directly, and the left address of the image (`srcRect.left - srcPixmap.bounds.left`) is not aligned and your compressor component can handle these pixels directly, then it should set the `codecCanShift` flag in the `flags` field of the compressor capabilities structure. If your compressor component does not set this flag, then the data will be copied to a buffer with the image shifted so the first pixel is in the most significant bit of an aligned long-word address.
- **Updating previous pixel maps.** Compressors that perform temporal compression may keep their own copy of the previous frame's pixel map, or they may update the previous frame's pixel map as they perform the compression. In these cases, the compressor component should set the `codecCanCopyPrev` flag if it updates the previous pixel map with the original data from the current frame, or it should set the `codecCanCopyPrevComp` flag if it updates the previous pixel map with a compressed copy of the current frame.

Listing 4-1 Preparing for simple compression operations

```

pascal long
CDPreCompress (Handle storage, register CodecCompressParams *p)
{
    CodecCapabilities *capabilities = p->capabilities;
/*
    First the compressor returns which depth input pixels it
    supports based on what the application has available. This
    compressor can only work with 32-bit input pixels.
*/
    switch ( (*p->imageDescription)->depth ) {
        case 16:
            capabilities->wantedPixelSize = 32;
            break;
        default:
            return(codecConditionErr);
            break;
    }

/*
    If the buffer gets banded, return the smallest one the
    compressor can handle.
*/
    capabilities->bandMin = 2;

/*
    If the buffer gets banded, return the increment
    by which it should increase.
*/
    capabilities->bandInc = 2;

    capabilities->extendWidth = (*p->imageDescription)->width & 1;
    capabilities->extendHeight = (*p->imageDescription)->height &
                                1;

/*
    For efficiency, if the compressor could perform extension,
    these flags would be set to 0.
*/

    return(noErr);
}

```

Compressing a Horizontal Band of an Image

Listing 4-2 shows how the Image Compression Manager calls the `CDBandCompress` function when it wants the compressor to compress a horizontal band of an image.

Note

This example does not perform compression on bands with a bit depth of more than 1 or an extension of width and height. If the example did do so, it would handle these cases faster. ♦

Listing 4-2 Performing simple compression on a horizontal band of an image

```
pascal long
CDBandCompress (Handle storage, register CodecCompressParams *p)
{
    short          width,height;
    Ptr            cDataPtr,dataStart;
    short          depth;
    Rect           sRect;
    long           offsetH,offsetV;
    Globals        **glob = (Globals **)storage;
    register char  *baseAddr;
    long           numLines,numStrips;
    short          rowBytes;
    long           stripBytes;
    char           mmuMode = 1;
    register short y;
    ImageDescription **desc = p->imageDescription;
    OSErr          result = noErr;

    /*
    If there is a progress function, give it an open call at
    the start of this band.
    */

    if (p->progressProcRecord.progressProc)
        p->progressProcRecord.progressProc (codecProgressOpen, 0,
        p->progressProcRecord.progressRefCon);

    width = (*desc)->width;
    height = (*desc)->height;
    depth = (*desc)->depth;
    dataStart = cDataPtr = p->data;
```

Image Compressor Components

```

/*
    Figure out offset to first pixel in baseAddr from the
    pixel size and bounds.
*/

rowBytes = p->srcPixmap.rowBytes;
sRect = p->srcPixmap.bounds;

numLines = p->stopLine - p->startLine; /* number of scan
                                       lines */
numStrips = (numLines+1)>>1;           /* number of strips
                                       in */
stripBytes = ((width+1)>>1) * 5;

/*
    Adjust the source baseAddress to be at the beginning
    of the desired rect.
*/

switch ( p->srcPixmap.pixelSize ) {
case 32:
    offsetH = sRect.left<<2;
    break;
case 16:
    offsetH = sRect.left<<1;
    break;
case 8:
    offsetH = sRect.left;
    break;

/*
    This compressor does not handle the other cases directly.
*/

default:
    result = codecErr;
    goto bail;
}

offsetV = sRect.top * rowBytes;
baseAddr = p->srcPixmap.baseAddr + offsetH + offsetV;

/*
    If there is not a data-unloading function,

```

Image Compressor Components

```

        adjust the pointer to the next band.
    */

    if ( p->flushProcRecord.flushProc == nil ) {
        cDataPtr += (p->startLine>>1) * stripBytes;
    }
    else { /*
        Make sure the compressor can deal with the
        data-unloading function in this case.
        */
        if ( p->bufferSize < stripBytes ) {
            result = codecSpoolErr;
            goto bail;
        }
    }
    /*
    Perform the slower data-loading or progress operation, as
    required.
    */

    if ( p->flushProcRecord.flushProc ||
        p->progressProcRecord.progressProc ) {

        SharedGlobals *sg = (*glob)->sharedGlob;

        for ( y=0; y < numStrips; y++) {
            SwapMMUMode(&mmuMode);
            CompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
            SwapMMUMode(&mmuMode);
            baseAddr += rowBytes<<1;
            if ( p->flushProcRecord.flushProc ) {
                if ( (result=
                    p->flushProcRecord.flushProc(cDataPtr,stripBytes,
                    p->flushProcRecord.flushRefCon)) != noErr) {
                    result = codecSpoolErr;
                    goto bail;
                }
            }
            else {
                cDataPtr += stripBytes;
            }
            if (p->progressProcRecord.progressProc) {
                if ( (result=
                    p->progressProcRecord.progressProc)

```

Image Compressor Components

```

        codecProgressUpdatePercent,
        FixDiv(y,numStrips),
        p->progressProcRecord.progressRefCon)
    ) != noErr ) {
        result = codecAbortErr;
        goto bail;
    }
}
}
} else {
    SharedGlobals *sg = (*glob)->sharedGlob;
    short tRowBytes = rowBytes<<1;

    SwapMMUMode(&mmuMode);
    for ( y=numStrips; y--; ) {
        CompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
        cDataPtr += stripBytes;
        baseAddr += tRowBytes;
    }
    SwapMMUMode(&mmuMode);
}
}

```

Decompressing an Image

When decompressing an image, the Image Compression Manager performs these three major tasks:

1. The Image Compression Manager first determines which decompressor is best able to decompress the image. To do so, the Image Compression Manager examines the source image as well as the parameters specified by the application. If the application requested a specific decompressor, the Image Compression Manager uses that decompressor (unless it is not installed, in which case the Image Compression Manager returns an error to the application). If the application did not request a decompressor, the Image Compression Manager chooses the decompressor that will do the best job. The Image Compression Manager collects the information it needs to choose a decompressor by issuing the `CDPreDecompress` request to each qualifying decompressor (see page 4-63 for a detailed description of the `CDPreDecompress` function).

2. If the chosen decompressor can handle the image directly, the Image Compression Manager passes the request through to the decompressor. The decompressor then processes the image and returns the image to the specified location.
3. If none of the decompressors can handle all of the conditions (matrix mapping, masking or matting, depth conversion, and so on) the Image Compression Manager allocates an offscreen buffer and passes image bands to the decompressor at a depth that the decompressor can handle by issuing a `CDBandDecompress` request. (For details on the `CDBandDecompress` function, see page 4-64). The decompressor processes each band, building the image as it goes. When the image has been completely decompressed, the Image Compression Manager returns control to the application.

Choosing a Decompressor

Listing 4-3 on page 4-20 provides an example of how a decompressor is chosen. The Image Compression Manager calls the `CDPreDecompress` function (described on page 4-63) before an image is decompressed. The decompressor returns information about how it can decompress an image. The Image Compression Manager can fit the destination pixel map to your decompressor's requirements if it is not able to support decompression to the destination directly. The capability information the decompressor returns includes

- depth of pixels for the destination pixel map
- minimum band size handled
- extension width and height required
- band increment size

When your decompressor component is called with the `CDPreDecompress` function, it can handle all aspects of the call itself, or only the most common ones. All decompressors must handle at least one case.

This section contains a bulleted list of some of the operations your decompressor component can perform during the decompression operation. The list describes which parameters in the decompression parameters structure (described on page 4-46) indicate the operations are required and which flags in the flags field of the compressor capabilities structure (described on page 4-35) must be set to allow your decompressor to handle them.

For sequences of images the `conditionFlags` field in the decompression parameters structure can be used to determine which parameters may have changed since the last decompression operation. These parameters are also indicated in the bulleted list.

Since your decompressor's capabilities depend on the full combination of parameters, it must inspect all the relevant parameters before indicating that it will perform one of the operations itself. For instance, if your decompressor has hardware that can perform scaling only if the destination pixel depth is 32 and there is no clipping, then the pre-decompression operation would have to check the following fields in the decompression parameters structure: the `matrix` field, the `pixelSize` field of the destination pixel map structure pointed to by the `destPixMap` field, and the `maskBits` fields. Only then could the decompressor decide whether to set the `codecCanScale` flag in the `capabilities` field of the decompression parameters structure.

- **Scaling.** The decompressor component can look at the `matrix` and selectively decide which scaling operations it wishes to handle. If the scaling factor specified by the `matrix` is not unity and your decompressor can perform the scaling operation, it must set the `codecCanScale` flag in the `capabilities` field. If it does not, then the decompressor is asked to decompress without scaling, and the Image Compression Manager performs the scaling operation afterward.
- **Depth conversion.** If your component can decompress to the pixel depth indicated by the `pixelSize` field (of the pixel map structure pointed to by the `dstPixmap` field of the decompression parameters structure), it should set the `wantedPixelSize` field of the compressor capability structure to the same value. If it cannot handle that depth, it should specify the closest depth it can handle in the `wantedPixelSize` field.
- **Dithering.** When determining whether depth conversion can be performed (for converting an image to a lower bit depth, or to a similar bit depth with a different color table), dithering may be required. This is specified by the `dither` bit in the `transferMode` field (0x40) of the decompression parameters structure being set. The `accuracy` field of the decompression parameters structure indicates whether fast dithering is acceptable (`accuracy <= codecNormalQuality`) or whether true error diffusion dithering should be used (`accuracy > codecNormalQuality`). Most decompressors do not perform true error diffusion dithering, although they can. When a decompressor cannot perform the dither operation, it should specify the higher bit depth in the `wantedPixelSize` field of the compressor capability structure and let the Image Compression Manager perform the depth conversion with dithering. Dithering to 16-bit destinations is normally done only if the `accuracy` field is set to the `codecNormalQuality` value. However, if your decompressor component can perform dithering fast enough, it could be performed at the lower accuracy settings as well. To indicate that your decompressor can perform dithering as specified, it should set the `codecCanTransferMode` flag in the `capabilities` field of the decompression parameters structure.
- **Color remapping.** If the compressed data has an associated color lookup table that is different from the color lookup table of the destination pixel map, then the decompressor can remap the color indices to the closest available ones in the destination itself, or it can let the Image Compression Manager do the remapping. If the decompressor can do the mapping itself, it should set the `codecCanRemap` flag in the `capabilities` flags field of the decompression parameters structure.

- **Extending.** If the format for the compressed data is block-oriented, the decompressor can ask that the Image Compression Manager to allocate a buffer which is a multiple of the proper block size by setting the `extendWidth` and `extendHeight` fields of the compressor capabilities structure. If the right and bottom edges of the destination image (as determined by the transformed `srcRect` and `dstPixmap.bounds` fields of the decompression parameters structure) are not a multiple of the block size that your decompressor handles, and your decompressor cannot handle partial blocks (writing only the pixels that are needed for blocks that cross the left or bottom edge of the destination), then your decompressor component must set the `extendWidth` and `extendHeight` fields in the compressor capabilities structure. In this case, the Image Compression Manager creates a buffer large enough so that no partial blocks are needed. Your component can decompress into that buffer. This is then copied to the destination by the Image Compression Manager. Your component can avoid this extra step if it can handle partial blocks. In this case, it should leave the `extendWidth` and `extendHeight` fields set to 0.
- **Clipping.** If clipping must be performed on the image to be decompressed, the `maskBits` field of the decompression parameters structure is nonzero. In the `CDPreDecompress` function, it will be a region handle to the actual clipping region. If your decompressor can handle the clipping operation as specified by this region, it should set the `codecCanMask` bit in the `capabilities` flags field of the decompression parameters structure. If it does this, then the parameter passed to the `CDBandDecompress` function in the `maskBits` field will be a bitmap instead of a region. If desired, your decompressor can save a copy of the actual region structure during the pre-decompression operation.
- **Matting.** If a matte must be applied to the decompressed image, the `transferMode` field of the decompression parameters structure is set to `blend` and the `mattePixmap` field is a handle to the pixel map to be used as the matte. If your decompressor can perform the matte operation, then it should set the `codecCanMatte` field in the compressor capabilities structure. If it does not, then the Image Compression Manager will perform the matte operation after the decompression is complete.
- **Pixel shifting.** For pixel sizes less than 8 bits per pixel, it may be necessary to shift the destination pixels so that they are at an aligned address. If the pixel size of the destination pixel map is less than 8 and your component handles that depth directly, and the left address of the image is not aligned and your component can handle these pixels directly, then it should set the `codecCanShift` flag in the `capabilities` field of the decompression parameters structure. If your component does not set this flag, the Image Compression Manager allocates a buffer for and performs the shifting after the decompression is completed.
- **Partial extraction.** If the source rectangle is not the entire image and the component can decompress only the part of the image specified by the source rectangle, it should set the `codecCanSrcExtract` flag in the `capabilities` field of the decompression parameters structure. If it does not, the Image Compression Manager asks the component to decompress the entire image and copy only the required part to the destination.

Listing 4-3 Preparing for simple decompression

```

pascal long
CDPreDecompress(Handle storage, register CodecDecompressParams *p)

{
    register CodecCapabilities*capabilities = p->capabilities;
    RectdRect = p->srcRect;

    /*
       Check if the matrix is OK for this decompressor.
       This decompressor doesn't do anything fancy.
    */

    if ( !TransformRect(p->matrix,&dRect,nil) )
        return(codecConditionErr);

    /*
       Decide which depth compressed data this decompressor can
       deal with.
    */

    switch ( (*p->imageDescription)->depth ) {
        case 16:
            break;
        default:
            return(codecConditionErr);
            break;
    }

    /*
       This decompressor can deal only with 32-bit pixels.
    */

    capabilities->wantedPixelSize = 32;

    /*
       The smallest possible band the decompressor can handle is
       2 scan lines.
    */

    capabilities->bandMin = 2;

    /* This decompressor can deal with 2 scan line high bands. */

```

Image Compressor Components

```

capabilities->bandInc = 2;

/*
   If this decompressor needed its pixels be aligned on
   some integer multiple, you would set extendWidth and
   extendHeight to the number of pixels by which you need the
   destination extended. If you don't have such requirements
   or if you take care of them yourself, you set extendWidth
   and extendHeight to 0.
*/

capabilities->extendWidth = p->srcRect.right & 1;
capabilities->extendHeight = p->srcRect.bottom & 1;

return(noErr);
}

```

Decompressing a Horizontal Band of an Image

Listing 4-4 shows how to decompress the horizontal band of an image. The Image Compression Manager calls the `CDBandDecompress` function when it wants a decompressor to decompress an image or a horizontal band of an image. The pixel data indicated by the `baseAddr` field is guaranteed to conform to the criteria your decompressor specified in the `CDPreDecompress` function.

Note

This example does not perform decompression on bands with a bit depth of more than one or an extension of width and height. If the example did do so, it would handle these cases faster. ♦

Listing 4-4 Performing a decompression operation

```

pascal long
CDBandDecompress(Handle storage,register CodecDecompressParams *p)
{
    Rect          dRect;
    long          offsetH,offsetV;
    Globals       **glob  = (Globals **)(storage;
    long          numLines,numStrips;
    short         rowBytes;
    long          stripBytes;
    short         width;
    register short y;

```

Image Compressor Components

```

register char* baseAddr;
char          *cDataPtr;
char          mmuMode = 1;
OSErr         result = noErr;

/*
    Calculate the real base address based on the boundary
    rectangle. If it's not a linear transformation, this
    decompressor does not perform the operation.
*/

dRect = p->srcRect;
if ( !TransformRect(p->matrix,&dRect,nil) )
    return(paramErr);

/* If there is a progress function, give it an open call at
   the start of this band.
*/

if (p->progressProcRecord.progressProc)
    p->progressProcRecord.progressProc(codecProgressOpen,0,
        p->progressProcRecord.progressRefCon);

/*
    Initialize some local variables.
*/

width = (*p->imageDescription)->width;
rowBytes = p->dstPixMap.rowBytes;
numLines = p->stopLine - p->startLine; /* number of scan lines
                                       in this band */
numStrips = (numLines+1)>>1;           /* number of strips in
                                       this band */
stripBytes = ((width+1)>>1) * 5;       /* number of bytes in
                                       1 strip of blocks */

cDataPtr = p->data;

/*
    Adjust the destination base address to be at the beginning
    of the desired rectangle.
*/

offsetH = (dRect.left - p->dstPixMap.bounds.left);

```

Image Compressor Components

```

switch ( p->dstPixMap.pixelSize ) {
    case 32:
        offsetH <=<=2;  /* 1 pixel = 4 bytes */
        break;
    case 16:
        offsetH <=<=1;  /* 1 pixel = 2 bytes */
        break;
    case 8:
        break;          /* 1 pixel = 1 byte */
    default:
        result = codecErr;  /* This decompressor doesn't handle
                           these cases, although it
                           could. */
        goto bail;
}
offsetV = (dRect.top - p->dstPixMap.bounds.top) * rowBytes;
baseAddr = p->dstPixMap.baseAddr + offsetH + offsetV;

/*
   If your decompressor component is skipping some data,
   it just skips it here. You can tell because
   firstBandInFrame indicates this is the first band for a new
   frame, and if startLine is not 0, then that many lines were
   clipped out.
*/
if ( (p->conditionFlags & codecConditionFirstBand) &&
      p->startLine != 0 ) {
    if ( p->dataProcRecord.dataProc ) {
        for ( y=0; y < p->startLine>>1; y++ ) {
            if ( (result=p->dataProcRecord.dataProc
                  (&cDataPtr,stripBytes,
                   p->dataProcRecord.dataRefCon)) != noErr ) {
                result = codecSpoolErr;
                goto bail;
            }
            cDataPtr += stripBytes;
        }
    } else
        cDataPtr += (p->startLine>>1) * stripBytes;
}
/*
   If there is a data-loading function spooling the data to your
   decompressor, then you have to decompress the data in the

```

Image Compressor Components

```

    chunk size that is specified, or, if there is a progress
    function, you must make sure to call it as you go along.
    */

    if ( p->dataProcRecord.dataProc ||
        p->progressProcRecord.progressProc ) {

        SharedGlobals *sg = (*glob)->sharedGlob;

        for (y=0; y < numStrips; y++) {
            if (p->dataProcRecord.dataProc) {
                if ( (result=p->dataProcRecord.dataProc
                    (&cDataPtr,stripBytes,
                     p->dataProcRecord.dataRefCon)) != noErr ) {
                    result = codecSpoolErr;
                    goto bail;
                }
            }
            SwapMMUMode(&mmuMode);
            DecompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
            SwapMMUMode(&mmuMode);
            baseAddr += rowBytes<<1;
            cDataPtr += stripBytes;

            if (p->progressProcRecord.progressProc) {
                if ( (result=p->progressProcRecord.progressProc
                    (codecProgressUpdatePercent,
                     FixDiv(y, numStrips),
                     p->progressProcRecord.progressRefCon)) != noErr ) {
                    result = codecAbortErr;
                    goto bail;
                }
            }
        }
    }

    /*
    Otherwise, do the fast case.
    */
    } else {

```

Image Compressor Components

```

SharedGlobals *sg = (*glob)->sharedGlob;
shorttRowBytes = rowBytes<<1;

SwapMMUMode(&mmuMode);
for ( y=numStrips; y--; ) {
    DecompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
    baseAddr += tRowBytes;
    cDataPtr += stripBytes;
}
SwapMMUMode(&mmuMode);
}
/*
    IMPORTANT-- Update the pointer to data in the decompression
    parameters structure, so that when your decompressor gets the
    next band, you'll be at the right place in your data.
*/

p->data = cDataPtr;

if ( p->conditionFlags & codecConditionLastBand ) {
    /*
        Tie up any loose ends on the last band of the frame.
    */
}

bail:
/*
    If there is a progress function, give it a close call
    at the end of this band.
*/

if (p->progressProcRecord.progressProc)
    p->progressProcRecord.progressProc(codecProgressClose,0,
        p->progressProcRecord.progressRefCon);
return(result);
}

```

Image Compressor Components Reference

This section describes the constants, data structures, and functions that are specific to image compression components.

Constants

This section provides details on the image compressor component capability and format flags.

Image Compressor Component Capabilities

Apple has defined several component flags for image compressor components. These flags specify information about the capabilities of the component. You set these flags in the `componentFlags` field of your component's component description structure. The Image Compression Manager uses these same flags in the compressor information structure to describe the capabilities of image compressors and decompressors. For a complete description of this structure, see the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime*.

The `compressFlags` and `decompressFlags` fields of the compressor information structure contain a number of flags that define the capabilities of your component.

Note

If the compressor information structure is shared, the compressor component uses the component flags that are the same as the compression flags for the component description structure, and the decompressor component uses the component flags that are the same as the decompression flags for the component description structure. ♦

The flag bits for those fields are defined as follows (each flag is valid for both fields unless the description states otherwise):

```
#define codecInfoDoes1      (1L<<0)  /* works with 1-bit pixel
                                   maps */
#define codecInfoDoes2      (1L<<1)  /* works with 2-bit pixel
                                   maps */
#define codecInfoDoes4      (1L<<2)  /* works with 4-bit pixel
                                   maps */
#define codecInfoDoes8      (1L<<3)  /* works with 8-bit pixel
                                   maps */
#define codecInfoDoes16     (1L<<4)  /* works with 16-bit pixel
                                   maps */
```


Image Compressor Components

```

#define codecInfoDoes32          (1L<<5)  /* works with 32-bit pixel
                                           maps */
#define codecInfoDoesDither      (1L<<6)  /* supports fast dithering */
#define codecInfoDoesStretch     (1L<<7)  /* stretches to arbitrary
                                           sizes */
#define codecInfoDoesShrink      (1L<<8)  /* shrinks to arbitrary sizes */
#define codecInfoDoesMask        (1L<<9)  /* handles clipping regions */
#define codecInfoDoesTemporal    (1L<<10) /* sequential temporal
                                           compression */
#define codecInfoDoesDouble      (1L<<11) /* stretches to double size
                                           exactly */
#define codecInfoDoesQuad        (1L<<12) /* stretches to quadruple
                                           size */
#define codecInfoDoesHalf        (1L<<13) /* shrinks to half size */
#define codecInfoDoesQuarter     (1L<<14) /* shrinks to one-quarter
                                           size */
#define codecInfoDoesRotate      (1L<<15) /* rotates during
                                           decompression */
#define codecInfoDoesHorizFlip   (1L<<16) /* flips horizontally during
                                           decompression */
#define codecInfoDoesVertFlip    (1L<<17) /* flips vertically during
                                           decompression */
#define codecInfoDoesSkew        (1L<<18) /* skews image during
                                           decompression */
#define codecInfoDoesBlend       (1L<<19) /* blends image with matte
                                           during decompression */
#define codecInfoDoesWarp        (1L<<20) /* warps image arbitrarily
                                           during decompression */
#define codecInfoDoesRecompress  (1L<<21) /* recompresses images without
                                           accumulating errors */
#define codecInfoDoesSpool       (1L<<22) /* uses data-loading or
                                           data-unloading function */
#define codecInfoDoesRateConstrain (1L<<23) /* constrains amount of
                                           generated data to
                                           caller-defined limit */

```

Image Compressor Components

Flag descriptions`codecInfoDoes1`

Indicates whether the component can work with pixel maps that contain 1-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 1-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoes2`

Indicates whether the component can work with pixel maps that contain 2-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 2-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoes4`

Indicates whether the component can work with pixel maps that contain 4-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 4-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoes8`

Indicates whether the component can work with pixel maps that contain 8-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 8-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoes16`

Indicates whether the component can work with pixel maps that contain 16-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 16-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoes32`

Indicates whether the component can work with pixel maps that contain 32-bit pixels. If this flag is set to 1, then the component can compress or decompress images that contain 32-bit pixels. If this flag is set to 0, then the component cannot handle such images.

`codecInfoDoesDither`

Indicates whether the component supports dithering. If this flag is set to 1, the component supports dithering of colors. If this flag is set to 0, the component does not support dithering. This flag is only available for decompressor components.

`codecInfoDoesStretch`

Indicates whether the component can stretch images to arbitrary sizes. If this flag is set to 1, the component can stretch images. If this flag is set to 0, the component does not support stretching. This flag is only available for decompressor components.

`codecInfoDoesShrink`

Indicates whether the component can shrink images to arbitrary sizes. If this flag is set to 1, the component can shrink images. If this flag is set to 0, the component does not support shrinking. This flag is only available for decompressor components.

Image Compressor Components

`codecInfoDoesMask`

Indicates whether the component can handle clipping regions. If this flag is set to 1, the component can mask to an arbitrary clipping region. If this flag is set to 0, the component does not support clipping regions. This flag is only available for decompressor components.

`codecInfoDoesTemporal`

Indicates whether the component supports temporal compression in sequences. If this flag is set to 1, the component supports time compression. If this flag is set to 0, the component does not support time compression.

`codecInfoDoesDouble`

Indicates whether the component supports stretching to double size during decompression. Since images are in two dimensions (height and width), this means a total of four times as many pixels. The parameters for the stretch operation are specified in the matrix structure for the request—the component modifies the scaling attributes of the matrix (see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for information about transformation matrices). If this flag is set to 1, the component can stretch an image to exactly four times its original size, up to the maximum size supported by the decompressor. If this flag is set to 0, the component does not support stretching to double size. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesQuad`

Indicates whether the component supports stretching an image to four times its original size during decompression. Since images are in two dimensions (height and width), this means a total of sixteen times as many pixels. The parameters for the stretch operation are specified in the matrix structure (defined by the `MatrixRecord` data type) for the request—the component modifies the scaling attributes of the matrix (see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for information about transformation matrices). If this flag is set to 1, the component can stretch an image to exactly sixteen times its original size, up to the maximum size supported by the decompressor. If this flag is set to 0, the component does not support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesHalf`

Indicates whether the component supports shrinking an image to half of its original size during decompression. Since images are in two dimensions (height and width), this means a total of one-fourth the number of pixels. The parameters for the shrink operation are specified in the matrix structure for the request—the component modifies the scaling attributes of the matrix (see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for information about transformation matrices). If this flag is set to 1, the component can shrink an image to half size, down to the minimum size specified by the `minimumHeight` and `minimumWidth` fields in the compressor information structure. If this flag is set to 0, the component does not

Image Compressor Components

support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesQuarter`

Indicates whether the component can shrink an image to one-quarter of its original size during decompression. Since images are in two dimensions (height and width), this means a total of one-sixteenth the number of pixels. The parameters for the shrink operation are specified in the matrix structure for the request—the component modifies the scaling attributes of the matrix (see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for information about transformation matrices). If this flag is set to 1, the component can shrink an image to exactly one-quarter of its original size, down to the minimum size specified by the `minimumHeight` and `minimumWidth` fields in the compressor information structure. If this flag is set to 0, the component does not support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesRotate`

Indicates whether the component can rotate an image during decompression. The parameters for the rotation are specified in the matrix structure for a decompression operation. If this flag is set to 1, the component can rotate the image at decompression time. If this flag is set to 0, the component cannot rotate the resulting image. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesHorizFlip`

Indicates whether the component can flip an image horizontally during decompression. The parameters for the horizontal flip are specified in the matrix structure for a decompression operation. If this flag is set to 1, the component can flip the image at decompression time. If this flag is set to 0, the component cannot flip the resulting image. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesVertFlip`

Indicates whether the component can flip an image vertically during decompression. The parameters for the vertical flip are specified in the matrix structure for a decompression operation. If this flag is set to 1, the component can flip the image at decompression time. If this flag is set to 0, the component cannot flip the resulting image. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesSkew`

Indicates whether the component can skew an image during decompression. Skewing an image distorts it linearly along only a single axis—for example, drawing a rectangular image into a parallelogram-shaped region. The parameters for the skew operation are specified in the matrix structure for the

decompression request. If this flag is set to 1, the component can skew an image at decompression time. If this flag is set to 0, the component does not support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesBlend`

Indicates whether the component can blend the resulting image with a matte during decompression. The matte is provided by the application in the decompression request. If this flag is set to 1, the component can blend during decompression. If this flag is set to 0, the component does not support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesWarp`

Indicates whether the component can warp an image during decompression. Warping an image distorts it along one or more axes, perhaps in a nonlinear fashion, in effect “bending” the resulting region. The parameters for the warp operation are specified in the matrix structure for the decompression request. If this flag is set to 1, the component can warp an image at decompression time. If this flag is set to 0, the component does not support this capability. This flag is valid only for the `decompressFlags` field.

`codecInfoDoesRecompress`

Indicates whether the component can recompress images it has previously compressed without losing image quality. Many compression algorithms cause image degradation when you apply them repeatedly to the same image. If this flag is set to 1, the component uses an algorithm that does not compromise image quality after repeated compressions. If this flag is set to 0, you should not use the component for repeated compressions of the same image. This flag is only available for compressor components.

`codecInfoDoesSpool`

Indicates whether the component uses data-loading or data-unloading functions. Your application can define data-loading and data-unloading functions to help the component work with images that are too large to be stored in memory (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about data-loading and data-unloading functions). If this flag is set to 1, the component uses these functions if needed for a given operation. If this flag is set to 0, the component does not use these functions under any circumstances.

`codecInfoDoesRateConstrain`

Indicates the compressor is able to constrain the amount of data it generates when compressing sequences of images to a limit defined by the caller. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for details on data rate constraint functions. This flag is only available for compressor components.

Format of Compressed Data and Files

The `formatFlags` field of the compressor information structure contains a number of flags that define the possible format of compressed data produced by the component and the format of compressed files that the component can handle during decompression. The defined flags are as follows:

```
#define codecInfoDepth1      (1L<<0) /* compressed images with 1-bit color
                                     depth available */
#define codecInfoDepth2      (1L<<1) /* compressed images with 2-bit color
                                     depth available */
#define codecInfoDepth4      (1L<<2) /* compressed images with 4-bit color
                                     depth available */
#define codecInfoDepth8      (1L<<3) /* compressed images with 8-bit color
                                     depth available */
#define codecInfoDepth16     (1L<<4) /* compressed images with 16-bit color
                                     depth available */
#define codecInfoDepth32     (1L<<5) /* compressed images with 32-bit color
                                     depth available */
#define codecInfoDepth24     (1L<<6) /* compressed images with 24-bit color
                                     depth available */
#define codecInfoDepth33     (1L<<7) /* compressed data with monochrome images
                                     of 1-bit color depth */
#define codecInfoDepth34     (1L<<8) /* compressed images with 2-bit grayscale
                                     depth available */
#define codecInfoDepth36     (1L<<9) /* compressed images with 4-bit grayscale
                                     depth available */
#define codecInfoDepth40     (1L<<10) /* compressed images with 8-bit grayscale
                                     depth available */
#define codecInfoStoresClut (1L<<11) /* compressed data with custom color
                                     tables */
#define codecInfoDoesLossless
                                     (1L<<12) /* compressed data stored lossless
                                     format */
#define codecInfoSequenceSensitive
                                     (1L<<13) /* compressed data requires non-key
                                     frames to be decompressed in same
                                     order as compressed */
```

Flag descriptions`codecInfoDepth1`

Indicates whether the component can work with files containing color images with a color depth of 1 bit. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth2`

Indicates whether the component can work with files containing color images with a color depth of 2 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth4`

Indicates whether the component can work with files containing color images with a color depth of 4 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth8`

Indicates whether the component can work with files containing color images with a color depth of 8 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth16`

Indicates whether the component can work with files containing color images with a color depth of 16 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth32`

Indicates whether the component can work with files containing color images with a color depth of 32 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files. This flag is the same as the `codecInfoDepth24` flag except it contains one extra byte used as an alpha channel.

`codecInfoDepth24`

Indicates whether the component can work with files containing color images with a color depth of 24 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth33`

Indicates whether the component can work with files containing monochrome images, which have a grayscale depth of 1 bit. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

Image Compressor Components

`codecInfoDepth34`

Indicates whether the component can work with files containing grayscale images with a grayscale depth of 2 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth36`

Indicates whether the component can work with files containing grayscale images with a grayscale depth of 4 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoDepth40`

Indicates whether the component can work with files containing grayscale images with a grayscale depth of 8 bits. If this flag is set to 1, the component can compress into and decompress from files at this depth. If this flag is set to 0, the component cannot handle such files.

`codecInfoStoresClut`

Indicates whether the component can accommodate compressed data with custom color tables. If this flag is set to 1, the component can create compressed files with custom color tables and can decompress files that contain custom color tables. If this flag is set to 0, the component cannot handle such files.

`codecInfoDoesLossless`

Indicates whether the component can perform lossless compression or decompression operations. Lossless compression results in a decompressed image that is exactly the same as the original, uncompressed image. If this flag is set to 1, the component can perform lossless compression or decompression. If this flag is set to 0, the component cannot perform lossless operations. The application specifies a lossless operation by setting the desired quality level to `codecLosslessQuality` (see *Inside Macintosh: QuickTime* for more information about quality levels).

`codecInfoSequenceSensitive`

Indicates that the compressed data generated by this image compressor component has the requirement that non-key frames in a sequence be decompressed in the same order that they were compressed.

Data Types

This section discusses the data structures that the Image Compression Manager uses to communicate with image compressor and decompressor components.

The Compressor Capability Structure

Image compressor components use the compressor capability structure to report their capabilities to the Image Compression Manager. Before compressing or decompressing an image, the Image Compression Manager requests this capability information from the component that will be handling the operation by calling the `CDPreCompress` or `CDPreDecompress` function provided by that component. The compressor component examines the compression or decompression parameters and indicates any restrictions on its ability to satisfy the request in a formatted compressor capability structure. The Image Compression Manager then manages the operation according to the capabilities of the component.

The `CodecCapabilities` data type defines the compressor capability structure.

```
typedef struct {
    long          flags;           /* control information */
    short         wantedPixelSize; /* pixel depth for component
                                   to use with image */
    short         extendWidth;     /* extension width of image
                                   in pixels */
    short         extendHeight;    /* extension height of image
                                   in pixels */
    short         bandMin;         /* supported minimum
                                   image band height */
    short         bandInc;         /* common factor of
                                   supported band heights */
    short         pad;             /* reserved */
    unsigned long time;            /* milliseconds operation
                                   takes to complete */
} CodecCapabilities;

typedef CodecCapabilities *CodecCapabilitiesPtr;
```

Image Compressor Components

Field descriptions

<code>flags</code>	Contains flags that contain control information that is used by both the Image Compression Manager and the compressor component. The defined bit positions for this field are discussed later in this section.
<code>wantedPixelSize</code>	Indicates the pixel depth the component can use with the specified image. The component determines the pixel depth of the image for the operation by examining the appropriate pixel map.
<code>extendWidth</code>	Specifies the number of pixels the image must be extended in width. If the component cannot accommodate the image at its given width, the component may request that the Image Compression Manager extend the width of the image by adding pixels to the right edge of the image. This is sometimes necessary to accommodate the component's block size.
<code>extendHeight</code>	Specifies the number of pixels the image must be extended in height. If the component cannot accommodate the image at its given height the component may request that the Image Compression Manager extend the height of the image by adding pixels to the bottom of the image. This is sometimes necessary to accommodate the component's block size.
<code>bandMin</code>	Contains the minimum image band height supported by the component. Components that can tolerate small values operate under a wider set of memory conditions.
<code>bandInc</code>	Specifies a common factor of supported image band heights. A component may support only image bands that are an even multiple of some number of pixels high. These components report this common factor in the <code>bandInc</code> field. Set this field to 1 if your component supports bands of any size.
<code>pad</code>	Reserved for use by Apple.
<code>time</code>	Indicates the number of milliseconds the operation will take to complete. If the compressor cannot determine this value, it sets this field to 0.

The `flags` field of the compressor capability structure contains flags that exchange control information between the Image Compression Manager and the compressor component. Components use flags in the low-order 16 bits to indicate their capabilities to the manager. The Image Compression Manager may use flags in the high-order 16 bits to pass control information to the component.

The following flags are defined:

```
#define codecCanScale          (1L<<0)  /* decompressor scales
                                         information */
#define codecCanMask          (1L<<1)  /* decompressor applies mask to
                                         image */
#define codecCanMatte         (1L<<2)  /* decompressor blends image using
                                         matte */
#define codecCanTransform     (1L<<3)  /* decompressor works with complex
                                         placement matrices */
#define codecCanTransferMode  (1L<<4)  /* decompressor accepts transfer
                                         mode */
#define codecCanCopyPrev      (1L<<5)  /* compressor updates previous
                                         image buffer */
#define codecCanSpool         (1L<<6)  /* component can use functions to
                                         spool data */
#define codecCanClipVertical   (1L<<7)  /* decompressor clips image
                                         vertically */
#define codecCanClipRectangular (1L<<8) /* decompressor clips image
                                         vertically & horizontally */
#define codecCanRemapColor     (1L<<9)  /* compressor remaps color */
#define codecCanFastDither    (1L<<10) /* compressor supports fast
                                         dithering */
#define codecCanSrcExtract     (1L<<11) /* compressor extracts portion
                                         of source image */
#define codecCanCopyPrevComp  (1L<<12) /* compressor updates previous
                                         image buffer */
#define codecCanAsync          (1L<<13) /* component can work
                                         asynchronously */
#define codecCanMakeMask      (1L<<14) /* decompressor makes
                                         modification masks */
#define codecCanShift          (1L<<15) /* component works with pixels
                                         that are not byte-aligned */
```

IMPORTANT

The following flags are currently unused by the Image Compression Manager: `codecCanClipVertical`, `codecCanClipRectangular`, and `codecCanFastDither`. ▲

Image Compressor Components

Flag descriptions

<code>codecCanScale</code>	Indicates whether the decompressor can scale the image during decompression. The decompressor sets this flag to 1 to indicate that it can scale the image during decompression. The decompressor sets this flag to 0 if it cannot scale the decompressed image.
<code>codecCanMask</code>	Indicates whether the decompressor can apply a mask to the decompressed image. The decompressor sets this flag to 1 to indicate that it can use a mask to control the image that results from a decompression operation. The decompressor sets this flag to 0 if it cannot work with masks.
<code>codecCanMatte</code>	Indicates whether the decompressor can blend the decompressed image using a matte. The decompressor sets this flag to 1 to indicate that it can use a blend matte during decompression. The decompressor sets this flag to 0 if it cannot use a blend matte.
<code>codecCanTransform</code>	Indicates whether the decompressor can work with complex placement matrixes. The decompressor sets this flag to 1 to indicate that it can work with transformation matrixes during decompression. The decompressor sets this flag to 0 if it cannot work with matrixes.
<code>codecCanTransferMode</code>	Indicates whether the decompressor can accept a transfer mode other than source copy or dither copy when displaying a decompressed image. The decompressor sets this flag to 1 to indicate that it can accept transfer modes; otherwise, the decompressor sets this flag to 0.
<code>codecCanCopyPrev</code>	Indicates whether the compressor can update the previous image buffer during sequence compression. The compressor sets this flag to 1 to indicate that it can update the previous image buffer. The compressor sets this flag to 0 if it cannot update the buffer.
<code>codecCanSpool</code>	Indicates whether the component can use data-loading and data-unloading functions to spool data during decompression and compression operations, respectively. Applications may define data-loading and data-unloading functions to handle images that cannot fit into memory (see the chapter “Image Compression Manager” in <i>Inside Macintosh: QuickTime</i> for more information on data-loading and data-unloading functions). The component sets this flag to 1 to indicate that it can use these functions. The component sets this flag to 0 to indicate that it cannot use these functions.
<code>codecCanClipVertical</code>	Indicates whether the decompressor can clip an image vertically during decompression. The decompressor sets this flag to 1 to indicate that it can clip an image vertically. The decompressor sets this flag to 0 to indicate that it cannot clip an image vertically.

Image Compressor Components

`codecCanClipRectangular`

Indicates whether the decompressor can clip both vertically and horizontally during decompression. The decompressor sets this flag to 1 to indicate that it can clip along both axes. The decompressor sets this flag to 0 to indicate that it cannot clip an image both vertically and horizontally.

`codecCanRemapColor`

Indicates whether the compressor can remap the colors for an image using color tables. The compressor sets this flag to 1 if it can remap colors. The compressor sets this flag to 0 if it cannot remap colors.

`codecCanFastDither`

Indicates whether the compressor supports fast dithering. The compressor sets this flag to 1 if it supports fast dithering. The compressor sets this flag to 0 if it does not support fast dithering. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about fast dithering.

`codecCanSrcExtract`

Indicates whether the compressor can extract a portion of the source image. The compressor sets this flag to 1 if it can extract a portion of the source image. The compressor sets the flag to 0 if it cannot.

`codecCanCopyPrevComp`

Indicates whether the compressor can update the previous image buffer during sequence compression using compressed data. The compressor sets this flag to 1 to indicate that it can update the previous image buffer. The compressor sets this flag to 0 if it cannot update the buffer.

`codecCanAsync`

Indicates whether the component can work asynchronously. The compressor sets this flag to 1 if it can compress and decompress asynchronously; otherwise, it sets this flag to 0.

`codecCanMakeMask`

Indicates whether the decompressor creates modification masks during decompression. These masks indicate which pixels in the decompressed image differ from the previous image and must therefore be displayed. Such masks are useful only when processing sequences. The decompressor sets this flag to 1 to indicate that it creates modification masks. The decompressor sets this flag to 0 if it does not create such masks.

`codecCanShift`

Indicates whether the component can work with pixels that are not byte-aligned. This flag is valid only when the source or destination uses fewer than 8 bits per pixel. Components set this flag to 1 if they can read or write pixels that are not byte-aligned. Components set this flag to 0 if pixels must be byte-aligned.

The Compression Parameters Structure

Compressor components accept the parameters that govern a compression operation in the form of a data structure. This data structure is called a *compression parameters structure*. This structure is used by the `CDBandCompress` and `CDPreCompress` functions (described on page 4-63 and page 4-62, respectively).

The compression parameters structure is defined by the `CodecCompressParams` data type as follows:

```
typedef struct {
    ImageSequence      sequenceID;          /* sequence identifier ID
                                           (precompress or
                                           bandcompress) */
    ImageDescriptionHandle imageDescription; /* handle to image
                                           description structure
                                           (precompress or
                                           bandcompress) */
    Ptr                data;               /* location for receipt of
                                           compressed image data */
    long               bufferSize;         /* size of buffer for data */
    long               frameNumber;        /* frame identifier */
    long               startLine;          /* starting line for band */
    long               stopLine;           /* ending line for band */
    long               conditionFlags;     /* condition flags */
    CodecFlags         callerFlags;        /* control information
                                           flags */
    CodecCapabilitiesPtr *capabilities;     /* pointer to compressor
                                           capability structure */
    ProgressProcRecord progressProcRecord; /* progress function
                                           structure */
    CompletionProcRecord completionProcRecord; /* completion function
                                           structure */
    FlushProcRecord    flushProcRecord;    /* data-unloading function
                                           structure */
    PixMap             srcPixMap;           /* pointer to image
                                           (precompress or
                                           bandcompress) */
    PixMap             prevPixMap;          /* pointer to pixel map
                                           for previous image */
    CodecQ             spatialQuality;      /* compressed image
                                           quality */
    CodecQ             temporalQuality;     /* sequence temporal
                                           quality */
}
```

Image Compressor Components

```

Fixed                similarity;          /* similarity between
                                           adjacent frames */
DataRateParamsPtr    dataRateParams;      /* data constraint
                                           parameters */
long                 reserved;            /* reserved */
} CodecCompressParams;

typedef CodecCompressParams *CodecCompressParamsPtr;

```

Field descriptions**sequenceID**

Contains a unique sequence identifier. If the image to be compressed is part of a sequence, this field contains the sequence identifier that was assigned by the `CompressSequenceBegin` function. If the image is not part of a sequence, this field is set to 0.

imageDescription

Contains a handle to the image description structure that describes the image to be compressed.

data

Points to a location to receive the compressed image data. This is a 32-bit clean address—do not call `StripAddress`. If there is not sufficient memory to store the compressed image, the application may choose to write the compressed data to mass storage during the compression operation. The `flushProc` field identifies the data-unloading function that the application provides for this purpose.

This field is used only by the `CDBandCompress` function.

bufferSize

Contains the size of the buffer specified by the `data` field. Your component sets the value of the `bufferSize` field to the number of bytes of compressed data written into the buffer. Your component should not return more data than the buffer can hold—it should return a nonzero result code instead.

This field is used only by the `CDBandCompress` function.

frameNumber

Contains a frame identifier. Indicates the relative frame number within the sequence. The Image Compression Manager increments this value for each frame in the sequence.

This field is used only by the `CDBandCompress` function.

startLine

Contains the starting line for the band. This field indicates the starting line number for the band to be compressed. The line number refers to the pixel row in the image, starting from the top of the image. The first row is row number 0.

This field is used only by the `CDBandCompress` function.

stopLine

Contains the ending line for the band. This field indicates the ending line number for the band to be compressed. The line number refers to the pixel row in the image, starting from the top of the image. The first row in the image is row number 0.

Image Compressor Components

The image band includes the row specified by this field. So, to define a band that contains one row of pixels at the top of an image, you set the `startLine` field to 0 and the `stopLine` field to 1.

`conditionFlags`

Contains flags that identify the condition under which your component has been called. This field is used only by the `CDBandCompress` function. In addition, these fields contain information about actions taken by your component. Condition flags fields contain the following flags:

```
#define codecConditionFirstBand    (1L<<0)
#define codecConditionLastBand    (1L<<1)
```

The `codecConditionFirstBand` constant is an input flag that indicates if this is the first band in the frame. If this flag is set to 1, then your component is being called for the first time for the current frame.

The `codecConditionLastBand` constant is an input flag that indicates if this is the last band in the frame. If this flag is set to 1, then your component is being called for the last time for the current frame. If the `codecConditionFirstBand` flag is also set to 1, this is the only time the Image Compression Manager is calling your component for the current frame.

The `codecConditionCodecChangedMask` constant is an output flag that indicates that the component has changed the mask bits. If your image decompressor component can mask decompressed images and if some of the image pixels should not be written to the screen, set to 0 the corresponding bits in the mask defined by the `maskBits` field in the decompression parameter structure. In addition, set this flag to 1. Otherwise, set this flag to 0.

`callerFlags`

The `callerFlags` constant is an output flag that contains flags providing further control information. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information about the Image Compression Manager function control flags. The following flags are available:

`codecFlagUpdatePrevious`

Controls whether your compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If this flag is set to 1, your compressor should copy the current frame into the previous frame buffer at the end of the frame-compression sequence. Use the source image.

`codecFlagWasCompressed`

Indicates to your compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. This flag is set to 1 to indicate that the image was previously compressed. This flag is set to 0 if the image was not previously compressed.

`codecFlagUpdatePreviousComp`

Controls whether your compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression. If this flag is set to 1, your compressor should update the previous frame buffer at the end of the frame-compression sequence, allowing your compressor to perform frame differencing against the compression results. Use the image that results from the compression operation. If this flag is set to 0, your compressor should not modify the previous frame buffer during compression.

`codecFlagLiveGrab`

Indicates whether the current sequence results from grabbing live video. When working with live video, your compressor should operate as quickly as possible and disable any additional processing, such as compensation for previously compressed data. This flag is set to 1 when you are compressing from a live video source.

This field is used only by the `CDBandCompress` function (described on page 4-63).

`capabilities`

Points to a compressor capability structure. The Image Compression Manager uses this field to determine the capabilities of your compressor component.

This field is used only by the `CDPreCompress` function (described on page 4-62).

`progressProcRecord`

Contains a progress function structure. During the compression operation, your compressor may occasionally call a function that the application provides in order to report your progress (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about progress functions). This field contains a structure that identifies the progress function. If the `progressProc` field in this structure is set to `nil`, the application has not supplied a progress function.

This structure is used only by the `CDBandCompress` function (described on page 4-63).

Image Compressor Components

`completionProcRecord`

Contains a completion function structure. This structure governs whether you perform the compression asynchronously. If the `completionProc` field in this structure is set to `nil`, perform the compression synchronously. If this field is not `nil`, it specifies an application completion function. Perform the compression asynchronously and call that completion function when your component is finished. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information on calling completion functions. If the `completionProc` field in this structure has a value of `-1`, perform the operation asynchronously but do not call the application’s completion function.

This structure is used only by the `CDBandCompress` function.

`flushProcRecord`

Contains a data-unloading function structure. If there is not enough memory to store the compressed image, the application may provide a function that unloads some of the compressed data (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about data-unloading functions). This field contains a structure that identifies that data-unloading function.

If the application did not provide a data-unloading function, the `flushProc` field in this structure is set to `nil`. In this case, your component writes the entire compressed image into the memory location specified by the `data` field.

The data-unloading function structure is defined by the `flushProcRecord` data type as follows:

```
struct FlushProcRecord {
    FlushProcPtr flushProc; /* pointer to
                           data-unloading
                           function */
    long flushRefCon;      /* data-unloading
                           function reference
                           constant */
};
typedef struct FlushProcRecord FlushProcRecord;
typedef FlushProcRecord *FlushProcRecordPtr;
```

The data-unloading function structure is used only by the `CDBandCompress` function (described on page 4-63).

`srcPixMap`

Points to the image to be compressed. The image must be stored in a pixel map structure. The contents of this pixel map differ from a standard pixel map in two ways. First, the `rowBytes` field is a full 16-bit value—the high-order bit is not necessarily set to 1. Second, the `baseAddr` field must contain a 32-bit clean address.

This field is used only by the `CDBandCompress` function.

Image Compressor Components

`prevPixMap`

Points to a pixel map containing the previous image. If the image to be compressed is part of a sequence that is being temporally compressed, this field defines the previous image for temporal compression. Your component should then use this previous image as the basis of comparison for the image to be compressed.

If the `temporalQuality` field is set to 0, do not perform temporal compression. If the `codecFlagUpdatePrevious` flag or the `codecFlagUpdatePreviousComp` flag in the `flags` field is set to 1, update the previous image at the end of the compression operation.

The contents of this pixel map differ from a standard pixel map in two ways. First, the `rowBytes` field is a full 16-bit value—the high-order bit is not necessarily set to 1. Second, the `baseAddr` field must contain a 32-bit clean address.

This field is used only by the `CDBandCompress` function.

`spatialQuality`

Specifies the desired compressed image quality. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for valid values.

This field is used only by the `CDBandCompress` function.

Check to see if the value of this parameter is `nil` and, if so, do not write to location 0.

`temporalQuality`

Specifies the desired sequence temporal quality. This field governs the level of compression the application desires with respect to information in successive frames in the sequence. If this field is set to 0, do not perform temporal compression on this frame. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for other available values.

This field is used only by the `CDBandCompress` function (described on page 4-63).

Check to see if the value of this parameter is `nil` and, if so, do not write to location 0.

`similarity`

Indicates the similarity between adjacent frames when performing temporal compression. Your component returns a fixed-point number in this field. That value indicates the relative similarity between the frame just compressed and the previous frame. Valid values range from 0 (key frame) to 1 (identical).

This field is used only by the `CDBandCompress` function.

Check to see if the value of this parameter is `nil` and, if so, do not write to location 0.

`dataRateParams`

Points to the parameters used when performing data rate constraint.

`reserved`

Reserved for use by Apple.

The Decompression Parameters Structure

Decompressors accept the parameters that govern a decompression operation in the form of a data structure. This data structure is called a *decompression parameters structure*. It is used by the `CDBandDecompress` and `CDPreDecompress` functions, which are described on page 4-64 and page 4-63, respectively.

The decompression parameters structure is defined by the `CodecDecompressParams` data type as follows:

```
typedef struct {
    ImageSequence          sequenceID;          /* unique sequence ID
                                                (predecompress,
                                                band decompress) */
    ImageDescriptionHandle imageDescription; /* handle to image description
                                                structure (predecompress,
                                                band decompress) */
    Ptr                    data;                /* compressed image data */
    long                   bufferSize;           /* size of data buffer */
    long                   frameNumber;          /* frame identifier */
    long                   startLine;            /* starting line for band */
    long                   stopLine;             /* ending line for band */
    long                   conditionFlags;        /* condition flags */
    CodecFlags             callerFlags;          /* control flags */
    CodecCapabilitiesPtr   *capabilities;        /* pointer to compressor
                                                capability structure
                                                (predecompress,
                                                band decompress) */
    ProgressProcRecord     progressProcRecord;   /* progress function
                                                structure */
    CompletionProcRecord   completionProcRecord; /* completion function
                                                structure */
    DataProcRecord         dataProcRecord;       /* data-loading function
                                                structure */
    CGrafPtr               port;                 /* pointer to color
                                                graphics port for image
                                                (predecompress,
                                                band decompress) */
    PixMap                 dstPixMap;            /* destination pixel map
                                                (predecompress,
                                                band decompress) */
    BitMapPtr              maskBits;             /* update mask */
    PixMapPtr              mattePixMap;          /* blend matte pixel map */
}
```

Image Compressor Components

```

Rect                srcRect;                /* source rectangle
                                           (predecompress,
                                           band decompress) */
MatrixRecordPtr     *matrix;                /* pointer to matrix structure
                                           (predecompress,
                                           band decompress) */
CodecQ              accuracy;               /* desired accuracy
                                           (predecompress,
                                           band decompress) */
short               transferMode;           /* transfer mode(predecompress,
                                           band decompress) */
long                reserved[2];           /* reserved */
} CodecDecompressParams;

typedef CodecDecompressParams *CodecDecompressParamsPtr;

```

Field descriptions

sequenceID	Contains the unique sequence identifier. If the image to be decompressed is part of a sequence, this field contains the sequence identifier that was assigned by the Image Compression Manager's DecompressSequenceBegin function. If the image is not part of a sequence, this field is set to 0.
imageDescription	Contains a handle to the image description structure that describes the image to be decompressed.
data	Points to the compressed image data. This must be a 32-bit clean address. The bufferSize field indicates the size of this data buffer. If the entire compressed image does not fit in memory, the application should provide a data-loading function, identified by the dataProc field of the data-loading function structure stored in the dataProcRecord field. This field is used only by the CDBandDecompress function (described on page 4-64).
bufferSize	Specifies the size of the image data buffer. This field is used only by the CDBandDecompress function.
frameNumber	Contains a frame identifier. Indicates the relative frame number within the sequence. The Image Compression Manager increments this value for each frame in the sequence. This field is used only by the CDBandDecompress function (described on page 4-64).
startLine	Specifies the starting line for the band. This field indicates the starting line number for the band to be decompressed. The line number refers to the pixel row in the image, starting from the top of the image. The first row in the image is row number 0. This field is used only by the CDBandDecompress function.

Image Compressor Components

stopLine Specifies the ending line for the band. This field indicates the ending line number for the band to be decompressed. The line number refers to the pixel row in the image, starting from the top of the image. The first row is row number 0.

The image band includes the row specified by this field. So, to define a band that contains one row of pixels at the top of an image, you set the `startLine` field to 0 and the `stopLine` field to 1.

This field is used only by the `CDBandDecompress` function.

conditionFlags Contains flags that identify the condition under which your component has been called (in order to save the component some work). The flags in this field are passed to the component in the `CDBandCompress` and `CDPreDecompress` functions when conditions change to save it some work. In addition, these fields contain information about actions taken by your component. Condition flags fields contain the following flags:

```
#define codecConditionFirstFrame      (1L<<2)
#define codecConditionNewDepth        (1L<<3)
#define codecConditionNewTransform    (1L<<4)
#define codecConditionNewSrcRect      (1L<<5)
#define codecConditionNewMatte        (1L<<7)
#define codecConditionNewTransferMode (1L<<8)
#define codecConditionNewClut         (1L<<9)
#define codecConditionNewAccuracy     (1L<<10)
#define codecConditionNewDestination  (1L<<11)
#define codecConditionCodecChangedMask (1L<<31)
```

The `codecConditionFirstBand` constant is an input flag that indicates if this is the first band in the frame. If this flag is set to 1, then your component is being called for the first time for the current frame.

The `codecConditionLastBand` constant is an input flag that indicates if this is the last band in the frame. If this flag is set to 1, then your component is being called for the last time for the current frame. If the `codecConditionFirstBand` flag is also set to 1, this is the only time the Image Compression Manager is calling your component for the current frame.

The `codecConditionFirstFrame` constant is an input flag that indicates that this is the first frame to be decompressed for this image sequence.

callerFlags

The `codecConditionNewDepth` constant is an input flag that indicates that the depth of the destination has changed for this image sequence.

The `codecConditionNewTransform` constant is an input flag that indicates that the transformation matrix has changed for this sequence.

The `codecConditionNewSrcRect` constant is an input flag that indicates that the source rectangle has changed for this sequence.

The `codecConditionNewMatte` is an input flag that indicates that the matte pixel map has changed for this sequence.

The `codecConditionNewTransferMode` constant is an input flag that indicates that the transfer mode has changed for this sequence.

The `codecConditionNewClut` constant is an input flag that indicates that the color lookup table has changed for this sequence.

The `codecConditionNewAccuracy` constant is an input flag that indicates to the component that the accuracy parameter has changed for this sequence.

The `codecConditionNewDestination` constant is an input flag that indicates to the component that the destination pixel map has changed for this sequence.

The `codecConditionCodecChangedMask` constant is an output flag that indicates that the component has changed the mask bits. If your image decompressor component can mask decompressed images and if some of the image pixels should not be written to the screen, set the corresponding bits in the mask (defined by the `maskBits` field in the decompression parameter structure) to 0. In addition, set this flag to 1. Otherwise, set this flag to 0.

Contains flags providing further control information. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information about the Image Compression Manager function control flags. Four flags are available:

The `codecFlagUpdatePrevious` flag controls whether your compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If this flag is set to 1, your compressor should copy the current frame into the previous frame buffer at the end of the frame-compression sequence. Use the source image.

The `codecFlagWasCompressed` flag indicates to your compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. This flag is set to 1 to indicate that the image was previously compressed. This flag is set to 0 if the image was not previously compressed.

Image Compressor Components

The `codecFlagUpdatePreviousComp` flag controls whether your compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression. If this flag is set to 1, your compressor should update the previous frame buffer at the end of the frame compression sequence, allowing your compressor to perform frame differencing against the compression results. Use the image that results from the compression operation. If this flag is set to 0, your compressor should not modify the previous frame buffer during compression.

The `codecFlagLiveGrab` flag indicates whether the current sequence results from grabbing live video. When working with live video, your compressor should operate as quickly as possible and disable any additional processing, such as compensation for previously compressed data. This flag is set to 1 when you are compressing from a live video source. This field is used only by the `CDBandCompress` function (described on page 4-63).

capabilities Points to a compressor capability structure (described on page 4-35). The Image Compression Manager uses this parameter to determine the capabilities of your decompressor component.

This field is used only by the `CDPreDecompress` function (described on page 4-63).

progressProcRecord Contains a progress function structure. During the decompression operation, your decompressor may occasionally call a function that the application provides in order to report your progress (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about progress functions). This field contains a structure that identifies the progress function. If the `progressProc` field of this structure is set to `nil`, the application did not provide a progress function.

The progress function structure is defined by the `progressProcRecord` data type as follows:

```
struct ProgressProcRecord {
    ProgressProcPtr progressProc; /* pointer to
                                   progress
                                   function */
    long progressRefCon; /* reference
                           constant */
};
typedef struct ProgressProcRecord ProgressProcRecord;
typedef ProgressProcRecord *ProgressProcRecordPtr;
```

This field is used only by the `CDBandDecompress` function (described on page 4-64).

completionProcRecord Contains a completion function structure. This field governs whether you perform the decompression asynchronously. If the

`completionProc` field in this structure is set to `nil`, perform the decompression synchronously. If this field is not `nil`, it specifies an application completion function. Perform the decompression asynchronously and call that completion function when your component is finished. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information on calling completion functions. If this field has a value of `-1`, perform the operation asynchronously but do not call the application’s completion function.

The completion function structure is defined by the `CompletionProcRecord` data type as follows:

```
struct CompletionProcRecord {
    CompletionProcPtr completionProc; /* pointer to
                                        completion
                                        function */
    long completionRefCon; /* reference
                            constant */
};
typedef struct CompletionProcRecord CompletionProcRecord;
typedef CompletionProcRecord *CompletionProcRecordPtr;
```

This field is used only by the `CDBandDecompress` function (described on page 4-64).

`dataProcRecord`

Contains a data-loading function structure. If the data stream is not all in memory, your component may call an application function that loads more compressed data (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about data-loading functions). This field contains a structure that identifies that data-loading function. If the application did not provide a data-loading function, the `dataProc` field in this structure is set to `nil`. In this case, the entire image must be in memory at the location specified by the `data` field.

The data-loading function structure is defined by the `dataProcRecord` data type as follows:

```
struct DataProcRecord {
    DataProcPtr dataProc; /* pointer to data-loading
                           function */
    long dataRefCon; /* reference constant */
};
typedef struct DataProcRecord DataProcRecord;
typedef DataProcRecord *DataProcRecordPtr;
```

This field is used only by the `CDBandDecompress` function.

Image Compressor Components

<code>port</code>	Points to the color graphics port that receives the decompressed image.
<code>dstPixMap</code>	<p>Points to the pixel map where the decompressed image is to be displayed. The <code>GDevice</code> global variable is set to the destination graphics device.</p> <p>The contents of this pixel map differ from a standard pixel map in two ways. First, the <code>rowBytes</code> field is a full 16-bit value—the high-order bit is not necessarily set to 1. Second, the <code>baseAddr</code> field must contain a 32-bit clean address.</p>
<code>maskBits</code>	<p>Contains an update mask. If your component can mask result data, use this mask to indicate which pixels in the destination pixel map to update. Your component indicates whether it can mask with the <code>codecCanMask</code> flag in the <code>flags</code> field of the compressor capability structure referred to by the <code>capabilities</code> field. This field is updated in response to the <code>CDPreDecompress</code> request (described on page 4-63). See “The Compressor Capability Structure” beginning on page 4-35 for a description of the compressor capability structure.</p> <p>If the mask has not changed since the last <code>CDBandDecompress</code> request, the <code>codecConditionCodecChangedMask</code> flag in the <code>conditionFlags</code> field is set to 0.</p> <p>This field is used only by the <code>CDBandDecompress</code> function.</p>
<code>mattePixMap</code>	<p>Points to a pixel map that contains a blend matte. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. The matte must be in the coordinate system of the source image. If the application does not want to apply a blend matte, this field is set to <code>nil</code>.</p> <p>The contents of this pixel map differ from a standard pixel map in two ways. First, the <code>rowBytes</code> field is a full 16-bit value—the high-order bit is not necessarily set to 1. Second, the <code>baseAddr</code> field must contain a 32-bit clean address.</p> <p>This field is used only by the <code>CDBandDecompress</code> function (described on page 4-64).</p>
<code>srcRect</code>	Points to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the compressed image, which is defined by the <code>width</code> and <code>height</code> fields of the image description structure referred to by the <code>imageDescription</code> field.
<code>matrix</code>	Points to a matrix structure that specifies how to transform the image during decompression.
<code>accuracy</code>	Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See the chapter “Image Compression Manager” in <i>Inside Macintosh: QuickTime</i> for valid values.

Image Compressor Components

transferMode	Specifies the QuickDraw transfer mode for the operation. For details on QuickDraw's transfer modes, see the chapter "Basic QuickDraw" in <i>Inside Macintosh: Imaging</i> .
reserved	Reserved for use by Apple.

Functions

This section describes the external interface that image compressor components must support. It also discusses the utility functions that the Image Compression Manager provides for use by compressors and decompressors.

This discussion has been divided into two parts. They discuss the image compressor component functions that are called by the Image Compression Manager. "Direct Functions" deals with image compressor component functions that are called by the Image Compression Manager in response to application requests. "Indirect Functions" discusses image compressor component functions that may be called by the Image Compression Manager at any time. The next section, "Image Compression Manager Utility Functions," defines a number of Image Compression Manager utility functions that are available to image compressor components.

Apple has defined a functional interface for image compressor components. For information about the functions your component must support, see the individual function descriptions that follow.

You can use the following constants to refer to the request codes for each of the functions that your component must support.

```
#define codecGetCodecInfo          0x00 /* CDGetCodecInf */
#define codecGetCompressionTime    0x01 /* CDGetCompressionTime */
#define codecGetMaxCompressionSize 0x02 /* CDGetMaxCompressionSize */
#define codecPreCompress           0x03 /* CDPreCompress */
#define codecBandCompress          0x04 /* CDBandCompress */
#define codecPreDecompress         0x05 /* CDPreDecompress */
#define codecBandDecompress        0x06 /* CDBandDecompress */
#define codecCDSequenceBusy       0x07 /* CDSequenceBusy */
#define codecGetCompressedImageSize 0x08 /* CDGetCompressedImageSize */
#define codecGetSimilarity         0x09 /* CDGetSimilarity */
#define codecTrimImage            0x0A /* CDTrimImage */
```

Note

Code selectors 0 through 127 are reserved for use by Apple. Code selectors 128 through 191 are subtype specific. Code selectors 192 through 255 are vendor specific. Code selectors 256 through 32767 are available for general use. Negative selectors are reserved by the Component Manager. ♦

Direct Functions

These functions are invoked by the Image Compression Manager in direct response to application functions. Refer to the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for descriptions of the functions that applications call.

CDGetCodecInfo

Your component receives the `CDGetCodecInfo` request whenever an application calls the Image Compression Manager’s `GetCodecInfo` function.

```
pascal ComponentResult CDGetCodecInfo (CodecInfo *info);
```

`info` Contains a pointer to the compressor information structure (defined by the `CodecInfo` data type) to update. Your component should report its capabilities by formatting a compressor information structure in the location specified by this parameter.

DESCRIPTION

Your component returns a formatted compressor information structure defining its capabilities.

Both compressors and decompressors may receive this request.

RESULT CODES

<code>noErr</code>	0	No error
<code>codecUnimpError</code>	-8962	Feature not implemented by this compressor

SEE ALSO

See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for a description of the compressor information structure.

CDGetMaxCompressionSize

Your component receives the `CDGetMaxCompressionSize` request whenever an application calls the Image Compression Manager's `GetMaxCompressionSize` function. The caller uses this function to determine the maximum size the data will become for a given parameter.

```
pascal ComponentResult CDGetMaxCompressionSize (PixMapHandle src,
                                                const Rect *srcRect,
                                                short depth,
                                                CodecQ quality,
                                                long *size);
```

src	Contains a handle to the source image. The source image is stored in a pixel map structure. Applications use the size information you return to allocate buffers that may be used for more than one image. Consequently, your compressor should not consider the contents of the image when determining the maximum compressed size. Rather, you should consider only the quality level, pixel depth, and image size. This parameter may be set to <code>nil</code> . In this case the application has not supplied a source image—your component should use the other parameters to determine the characteristics of the image to be compressed.
srcRect	Contains a pointer to a rectangle defining the portion of the source image to compress.
depth	Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
quality	Specifies the desired compressed image quality. See the chapter “Image Compression Manager” in <i>Inside Macintosh: QuickTime</i> for valid values.
size	Contains a pointer to a field to receive the maximum size, in bytes, of the compressed image.

DESCRIPTION

Your component returns a long integer indicating the maximum number of bytes of compressed data that results from compressing the specified image.

Only compressors receive this request.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

CDGetCompressionTime

Your component receives the `CDGetCompressionTime` request whenever an application calls the Image Compression Manager's `GetCompressionTime` function.

```
pascal ComponentResult CDGetCompressionTime (PixMapHandle src,
                                             const Rect *srcRect,
                                             short depth, CodecQ
                                             *spatialQuality,
                                             CodecQ *temporalQuality,
                                             unsigned long *time);
```

<code>src</code>	<p>Contains a handle to the source image. The source image is stored in a pixel map. Applications may use the time information you return for more than one image. Consequently, your compressor should not consider the contents of the image when determining the maximum compression time. Rather, you should consider only the quality level, pixel depth, and image size.</p> <p>This parameter may be set to <code>nil</code>. In this case the application has not supplied a source image—your component should use the other parameters to determine the characteristics of the image to be compressed.</p>
<code>srcRect</code>	Contains a pointer to a rectangle defining the portion of the source image to compress.
<code>depth</code>	Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
<code>spatialQuality</code>	Contains a pointer to a field containing the desired compressed image quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter “Image Compression Manager” in <i>Inside Macintosh: QuickTime</i> for valid values. Check to see if the value of this field is <code>nil</code> and, if so, do not write to location 0.
<code>temporalQuality</code>	Contains a pointer to a field containing the desired sequence temporal quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter “Image Compression Manager” in <i>Inside Macintosh: QuickTime</i> for valid values. Check to see if the value of this field is <code>nil</code> and, if so, do not write to location 0.
<code>time</code>	Contains a pointer to a field to receive the compression time, in milliseconds. If your component cannot determine the amount of time required to compress the image, set this field to 0. Check to see if the value of this field is <code>nil</code> and, if so, do not write to location 0.

DESCRIPTION

Your component returns a long integer indicating the maximum number of milliseconds it would require to compress the specified image.

Only compressors receive this request.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
codecUnimpError	-8962	Feature not implemented by this compressor

CDGetSimilarity

Your component receives the CDGetSimilarity request whenever an application calls the Image Compression Manager's GetSimilarity function. Your component compares the specified compressed image to a picture stored in a pixel map and returns a value indicating the relative similarity of the two images.

Note

The CDGetSimilarity function is optional. If your component doesn't support it, it should return the codecUnimpError result code. ♦

```
pascal ComponentResult CDGetSimilarity (PixMapHandle src,
                                         const Rect *srcRect,
                                         ImageDescriptionHandle desc,
                                         Ptr data,
                                         Fixed *similarity);
```

src	Contains a handle to the noncompressed image. The image is stored in a pixel map structure.
srcRect	Contains a pointer to a rectangle defining the portion of the image to compare to the compressed image.
desc	Contains a handle to the image description structure that defines the compressed image for the operation.
data	Contains a pointer to the compressed image data.
similarity	Contains a pointer to a field that is to receive the similarity value. Your component sets this field to reflect the relative similarity of the two images. Valid values range from 0 (key frame) to 1 (identical).

DESCRIPTION

If the source image has been temporally compressed and is not a key frame (that is, the image relies on other frames that are not available to your component at this time), your component should return a result value of `paramErr`.

Only decompressors receive this request.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecUnimpError</code>	-8962	Feature not implemented by this compressor

CDGetCompressedImageSize

Your component receives the `CDGetCompressedImageSize` request whenever an application calls the Image Compression Manager's `GetCompressedImageSize` function.

You can use the `CDGetCompressedImageSize` function when you are extracting a single image from a sequence; therefore, you don't have an image description structure and don't know the exact size of one frame. In this case, the Image Compression Manager calls the component to determine the size of the data.

```
pascal ComponentResult CDGetCompressedImageSize
                                (ImageDescriptionHandle desc,
                                 Ptr data, long bufferSize,
                                 DataProcRecordPtr dataProc,
                                 long *dataSize);
```

<code>desc</code>	Contains a handle to the image description structure that defines the compressed image for the operation.
<code>data</code>	Points to the compressed image data.
<code>bufferSize</code>	Specifies the size of the buffer to be used by the data-loading function specified by the <code>dataProc</code> parameter. If the application did not specify a data-loading function this parameter is <code>nil</code> .
<code>dataProc</code>	Points to a data-loading function structure. If the data stream is not all in memory when the application calls <code>GetCompressedImageSize</code> , your component may call an application function that loads more compressed data (see the chapter "Image Compression Manager" in <i>Inside Macintosh: QuickTime</i> for more information about data-loading functions). This parameter contains a pointer to a structure that identifies the data-loading

function. If the application did not provide a data-loading function, this parameter is `nil`. In this case, the entire image must be in memory at the location specified by the `data` parameter.

`dataSize` Contains a pointer to a field that is to receive the size, in bytes, of the compressed image.

DESCRIPTION

Your component returns a long integer indicating the number of bytes of data in the compressed image. You may want to store the image size somewhere in the image description structure, so that you can respond to this request quickly. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about image description structures.

Only decompressors receive this request.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>codecSpoolErr</code>	-8966	Error loading or unloading data

CDTrimImage

Your component receives the `CDTrimImage` request whenever an application calls the `TrimImage` function. Your component adjusts a compressed image to the boundaries defined by a rectangle specified by your application. The resulting image data is still compressed and is in the same compression format as the source image.

Note
The `CDTrimImage` function is optional. If your component doesn’t support it, it should return the `codecUnimpError` result code. ♦

pascal ComponentResult CDTrimImage
 (ImageDescriptionHandle desc, Ptr inData,
 long inBufferSize, DataProcRecordPtr dataProc,
 Ptr outData, long outBufferSize,
 FlushProcRecordPtr flushProc, Rect *trimRect,
 ProgressProcRecordPtr progressProc);

`desc` Contains a handle to the image description structure that describes the compressed image. Your component updates this image description to refer to the resized image.

Image Compressor Components

<code>inData</code>	Points to the compressed image data. If the entire compressed image cannot be stored at this location, the application may provide a data-loading function (see the description of the <code>dataProc</code> parameter to this function for details). This is a 32-bit clean address.
<code>inBufferSize</code>	Specifies the size of the buffer to be used by the data-loading function specified by the <code>dataProc</code> parameter. If the application did not specify a data-loading function, this parameter is <code>nil</code> .
<code>dataProc</code>	Points to a data-loading function structure. If the data stream is not all in memory when the application calls the Image Compression Manager's <code>GetCompressedImageSize</code> function, your component may call an application function that loads more compressed data (see the chapter "Image Compression Manager" in <i>Inside Macintosh: QuickTime</i> for more information about data-loading functions). This parameter contains a pointer to a structure that identifies the data-loading function. If the application did not provide a data-loading function, this parameter is <code>nil</code> . In this case, the entire image must be in memory at the location specified by the <code>inData</code> parameter.
<code>outData</code>	Points to a buffer to receive the trimmed image. If there is not sufficient memory to store the compressed image, the application may choose to write the compressed data to mass storage during the compression operation. The <code>flushProc</code> parameter identifies the data-unloading function. This is a 32-bit clean address. Your component should place the actual size of the resulting image into the <code>dataSize</code> field of the image description referred to by the <code>desc</code> parameter.
<code>outBufferSize</code>	Specifies the size of the buffer to be used by the data-unloading function specified by the <code>flushProc</code> parameter. If the application did not specify a data-unloading function, this parameter is <code>nil</code> .
<code>flushProc</code>	Points to a data-unloading function structure. If there is not enough memory to store the compressed image, your component may call an application function that unloads some of the compressed data (see the chapter "Image Compression Manager" in <i>Inside Macintosh: QuickTime</i> for more information about data-unloading functions). This parameter contains a pointer to a structure that identifies that data-unloading function. If the application did not provide a data-unloading function, this parameter is <code>nil</code> . In this case, your component writes the entire compressed image into the memory location specified by the <code>outData</code> parameter.
<code>trimRect</code>	Contains a pointer to a rectangle that defines the desired image dimensions. Your component adjusts the rectangle values so that they refer to the same rectangle in the resulting image (this is necessary whenever data is removed from the beginning of the image).

`progressProc`
Points to a progress function structure. During the operation, your component should occasionally call an application function to report its progress (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about progress functions). This parameter contains a pointer to a structure that identifies that progress function. If the application did not provide a progress function, this parameter is `nil`.

DESCRIPTION

Only decompressors receive this request. If the `TrimImage` function has been called by an application, the resulting picture should be modified.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>noCodecErr</code>	-8961	Image Compression Manager could not find the specified compressor
<code>codecUnimpErr</code>	-8962	Feature not implemented by this compressor
<code>codecSpoolErr</code>	-8966	Error loading or unloading data
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

CDCodecBusy

Your component receives the `CDCodecBusy` request whenever an application calls the `CDSequenceBusy` function. Your component must report whether it is performing an asynchronous operation.

`pascal ComponentResult CDCodecBusy (ImageSequence seq);`

`seq` Contains the unique sequence identifier assigned by the Image Compression Manager’s `CompressSequenceBegin` or `DecompressSequenceBegin` function.

DESCRIPTION

Your component should return a result code value of 1 if an asynchronous operation is in progress; it should return a result code value of 0 if the component is not performing an asynchronous operation. You can indicate an error by returning a negative result code. Both compressors and decompressors may receive this request.

Image Compressor Components

RESULT CODES

<code>noErr</code>	0	No error
<code>codecUnimpError</code>	-8962	Feature not implemented by this compressor

Indirect Functions

This section describes functions that are invoked by the Image Compression Manager but do not correspond to functions called by applications. The Image Compression Manager may call these functions at any time.

CDPreCompress

Your component receives the `CDPreCompress` request before compressing an image or a band of an image. The Image Compression Manager also calls this function when processing a sequence. In that case, the Image Compression Manager calls this function whenever the parameters governing the sequence operation have changed substantially. Your component indicates whether it can perform the requested compression operation.

```
pascal ComponentResult CDPreCompress
                                (CodecCompressParams *params);
```

`params` Contains a pointer to a compression parameters structure. The Image Compression Manager places the appropriate parameter information in that structure. See “The Compression Parameters Structure” beginning on page 4-40 for details.

DESCRIPTION

Your component should return a 0 result code to indicate that it can handle the request. In addition, your component indicates any limitations on its capabilities in a compressor capability structure (see “The Compressor Capability Structure” beginning on page 4-35 for details). Your component should return a result code of `codecConditionError` if it cannot field the compression request.

Only compressors receive this request.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>codecConditionErr</code>	-8972	Component cannot perform requested operation

CDBandCompress

Your component receives the CDBandCompress request to compress an image or a band of an image. The image may be part of a sequence.

```
pascal ComponentResult CDBandCompress
                                (CodecCompressParams *params);
```

params Contains a pointer to a compression parameters structure. The Image Compression Manager places the appropriate parameter information in that structure. See “The Compression Parameters Structure” beginning on page 4-40 for a complete description of the compression parameters structure.

DESCRIPTION

Only compressors receive this request.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
codecSpoolErr	-8966	Error loading or unloading data
codecAbortErr	-8967	Operation aborted by the progress function

CDPreDecompress

Your component receives the CDPreDecompress request before decompressing an image or a band of an image. The Image Compression Manager also calls this function when processing a sequence. In that case, the Image Compression Manager calls this function whenever the parameters governing the sequence operation have changed substantially. Your component indicates whether it can perform the requested decompression operation.

```
pascal ComponentResult CDPreDecompress
                                (CodecDecompressParams *params);
```

params Contains a pointer to a decompression parameters structure. The Image Compression Manager places the appropriate parameter information in that structure. See “The Decompression Parameters Structure” beginning on page 4-46 for a complete description of the decompression parameters structure.

Image Compressor Components

DESCRIPTION

Your component should return a 0 result code to indicate that it can handle the request. In addition, your component indicates any limitations on its capabilities in a compressor capability structure (see page 4-35 for a description of that structure). Return a result code of `codecConditionError` if your component cannot field the decompression request.

Only decompressors receive this request.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>codecConditionErr</code>	-8972	Component cannot perform requested operation

CDBandDecompress

Your component receives the `CDBandDecompress` request to decompress an image or a band of an image. The image may be part of a sequence.

```
pascal ComponentResult CDBandDecompress
                        (CodecDecompressParams *params);
```

`params` Contains a pointer to a decompression parameters structure. The Image Compression Manager places the appropriate parameter information in that structure. See “The Decompression Parameters Structure” beginning on page 4-46 for a complete description of the decompression parameters structure.

DESCRIPTION

Only decompressors receive these requests.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>codecSpoolErr</code>	-8966	Error loading or unloading data
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

Image Compression Manager Utility Functions

The Image Compression Manager provides a number of utility functions for use by your compressor component. These utility functions allow compressor components to manipulate the Image Compression Manager’s image description structures.

SetImageDescriptionExtension

Your component may use the `SetImageDescriptionExtension` function to create or update the extended data for an image.

```
pascal OSErr SetImageDescriptionExtension
                (ImageDescriptionHandle desc,
                 Handle extension,
                 long idType);
```

desc	Contains a handle to the appropriate image description structure. The <code>SetImageDescriptionExtension</code> function updates the size of the image description to accommodate the new extended data.
extension	Contains a handle to the new extended data. The <code>SetImageDescriptionExtension</code> function uses this data to update the extended data for the image described by the image description referred to by the <code>desc</code> parameter.
idType	Specifies the extension’s type value. Use this parameter to assign a data type to the extension. Use a four-character code, similar to an <code>OSType</code> field value.

DESCRIPTION

The Image Compression Manager appends the extended data for an image to the appropriate image description structure (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information about image description structures). Note that each compressor type may have its own format for the extended data that is stored with an image. The extended data is similar in concept to the user data that applications can associate with QuickTime movies—see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for more information about user data in QuickTime movies. Once you have added extended data to an image, you cannot delete it.

Image Compressor Components

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memFullErr	-108	Not enough memory available
noCodecErr	-8961	Image Compression Manager could not find the specified compressor
codecExtensionNotFoundErr	-8971	Requested extension is not in the image description

GetImageDescriptionExtension

Your component may use the `GetImageDescriptionExtension` function to obtain the extended data for an image.

```
pascal OSErr GetImageDescriptionExtension
                                (ImageDescriptionHandle desc,
                                 Handle *extension,
                                 long idType, long index);
```

desc	Contains a handle to the appropriate image description structure.
extension	Contains a pointer to a field to receive a handle to the returned data. The <code>GetImageDescriptionExtension</code> function returns the extended data for the image described by the image description referred to by the <code>desc</code> parameter. The function correctly sizes the handle for the data it returns.
idType	Specifies the extension's type value. Use this parameter to determine the data type of the extension. This parameter contains a four-character code, similar to an <code>OSType</code> field value.
index	Specifies the extension's index value.

DESCRIPTION

The Image Compression Manager appends the extended data for an image to the appropriate image description structure (see the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for information about image description structures). Note that each compressor type may have its own format for the extended data that is stored with an image. The extended data is similar in concept to the user data that applications can associate with QuickTime movies—see the chapter “Movie Toolbox” in *Inside Macintosh: QuickTime* for more information about user data in QuickTime movies. Once you have added extended data to an image, you cannot delete it.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memFullErr	-108	Not enough memory available
noCodecErr	-8961	The Image Compression Manager could not find the specified compressor
codecExtensionNotFoundErr	-8971	Requested extension is not in the image description

RemoveImageDescriptionExtension

The RemoveImageDescriptionExtension function allows you to remove an extension based on its type or index.

```
pascal OSErr RemoveImageDescriptionExtension
                                (ImageDescription **desc,
                                long type, long index);
```

desc	Contains a handle to the appropriate image description structure.
type	Specifies the extension's type, starting at 1. Use this parameter to specify the data type of the extension to be removed. This parameter contains a four-character code, similar to an OSType field value. Set the value of this parameter to 0 to indicate that any extension should be matched, with the index parameter becoming an index into all of the extensions.
index	Specifies the extension's index value.

RESULT CODE

codecExtensionNotFoundErr	-8971	Requested extension is not in the image description
---------------------------	-------	-----------------------------------------------------

CountImageDescriptionExtensionType

The CountImageDescriptionExtensionType function counts the number of image description extensions in a specified image description structure.

```
pascal OSErr CountImageDescriptionExtensionType
                                (ImageDescription **desc,
                                long type, long *count);
```

desc	Contains a handle to the image description structure with the extensions to be counted.
------	-----------------------------------------------------------------------------------------

Image Compressor Components

type	Indicates the type of extension to be counted in the specified image description structure. Set the value of this parameter to 0 to match any extension, and return a count of all of the extensions.
count	Contains a pointer to an integer that indicates how many extensions of the given type are in the given image description structure.

GetNextImageDescriptionExtensionType

The `GetNextImageDescriptionExtensionType` function retrieves the next extension type encountered after the one you specify.

```
pascal OSErr GetNextImageDescriptionExtensionType
                (ImageDescription **desc, long *type);
```

desc	Contains a handle to the image description structure with the extension under scrutiny.
type	Contains a pointer to an integer that indicates the type of the extension after which this function is to return the next extension type. Set the value of this parameter to 0 to return the first type found. Point to a value of 0 to return the first type found.

DESCRIPTION

If `GetNextImageDescriptionExtensionType` returns a value of 0 in the type parameter, no more types could be found.

Summary of Image Compressor Components

C Summary

Constants

```
#define compressorComponentType      'imco' /* compressor component type */
#define decompressorComponentType    'imdc' /* decompressor component type */

/* selector values */
#define codecGetCodecInfo             0x00 /* CDGetCodecInfo */
#define codecGetCompressionTime       0x01 /* CDGetCompressionTime */
#define codecGetMaxCompressionSize    0x02 /* CDGetMaxCompressionSize */
#define codecPreCompress              0x03 /* CDPreCompress */
#define codecBandCompress             0x04 /* CDBandCompress */
#define codecPreDecompress            0x05 /* CDPreDecompress */
#define codecBandDecompress           0x06 /* CDBandDecompress */
#define codecCDSequenceBusy           0x07 /* CDSequenceBusy */
#define codecGetCompressedImageSize   0x08 /* CDGetCompressedImageSize */
#define codecGetSimilarity            0x09 /* CDGetSimilarity */
#define codecTrimImage                0x0A /* CDTrimImage */

/* image compressor component capabilities flags */
#define codecCanScale                  (1L<<0) /* decompressor scales
                                                information */
#define codecCanMask                   (1L<<1) /* decompressor applies mask to
                                                image */
#define codecCanMatte                  (1L<<2) /* decompressor blends image using
                                                matte */
#define codecCanTransform              (1L<<3) /* decompressor works with complex
                                                placement matrices */
#define codecCanTransferMode           (1L<<4) /* decompressor accepts transfer
                                                mode */
#define codecCanCopyPrev               (1L<<5) /* compressor updates previous
                                                image buffer */
#define codecCanSpool                  (1L<<6) /* component can use functions to
                                                spool data */
#define codecCanClipVertical           (1L<<7) /* decompressor clips image
                                                vertically */
```

Image Compressor Components

```

#define codecCanClipRectangular (1L<<8) /* decompressor clips image
                                         vertically & horizontally */
#define codecCanRemapColor      (1L<<9) /* compressor remaps color */
#define codecCanFastDither      (1L<<10) /* compressor supports fast
                                         dithering */
#define codecCanSrcExtract      (1L<<11) /* compressor extracts portion
                                         of source image */
#define codecCanCopyPrevComp    (1L<<12) /* compressor updates previous
                                         image buffer */
#define codecCanAsync           (1L<<13) /* component can work
                                         asynchronously */
#define codecCanMakeMask        (1L<<14) /* decompressor makes
                                         modification masks */
#define codecCanShift           (1L<<15) /* component works with pixels
                                         that are not byte-aligned */

/* compressor component condition flags passed to component in
   CDBandDecompress and CDPDeCompress functions indicate changes */
#define codecConditionFirstBand (1L<<0) /* (input) first band
                                         in frame */
#define codecConditionLastBand  (1L<<1) /* (input) last band
                                         in frame */
#define codecConditionFirstFrame (1L<<2) /* (input) first frame to be
                                         decompressed in this
                                         sequence */
#define codecConditionNewDepth   (1L<<3) /* (input) depth of
                                         destination */
#define codecConditionNewTransform (1L<<4) /* (input) transformation
                                         matrix has changed */
#define codecConditionNewSrcRect (1L<<5) /* (input) source rectangle */
#define codecConditionNewMask    (1L<<6) /* (input) mask bitmap has
                                         changed */
#define codecConditionNewMatte   (1L<<7) /* (input) matte pixel map */
#define codecConditionNewTransferMode (1L<<8) /* (input) transfer mode */
#define codecConditionNewClut    (1L<<9) /* (input) color lookup
                                         table */
#define codecConditionNewAccuracy (1L<<10) /* accuracy parameter has
                                         changed */
#define codecConditionNewDestination (1L<<11) /* (input) destination pixel
                                         map */
#define codecConditionCodecChangedMask (1L<<31) /* (output) component has
                                         changed mask bits */

```

Image Compressor Components

```

/* compressor and decompressor flag bits */
#define codecInfoDoes1          (1L<<0) /* works with 1-bit pixel maps */
#define codecInfoDoes2          (1L<<1) /* works with 2-bit pixel maps */
#define codecInfoDoes4          (1L<<2) /* works with 4-bit pixel maps */
#define codecInfoDoes8          (1L<<3) /* works with 8-bit pixel maps */
#define codecInfoDoes16         (1L<<4) /* works with 16-bit pixel maps */
#define codecInfoDoes32         (1L<<5) /* works with 32-bit pixel maps */
#define codecInfoDoesDither     (1L<<6) /* supports fast dithering */
#define codecInfoDoesStretch    (1L<<7) /* stretches to arbitrary sizes */
#define codecInfoDoesShrink     (1L<<8) /* shrinks to arbitrary sizes */
#define codecInfoDoesMask       (1L<<9) /* handles clipping regions */
#define codecInfoDoesTemporal   (1L<<10) /* sequential temporal
                                         compression */
#define codecInfoDoesDouble      (1L<<11) /* stretches to double size
                                         exactly */
#define codecInfoDoesQuad       (1L<<12) /* stretches to quadruple size */
#define codecInfoDoesHalf       (1L<<13) /* shrinks to half size */
#define codecInfoDoesQuarter    (1L<<14) /* shrinks to one quarter size */
#define codecInfoDoesRotate     (1L<<15) /* rotates during decompression */
#define codecInfoDoesHorizFlip  (1L<<16) /* flips horizontally during
                                         decompression */
#define codecInfoDoesVertFlip   (1L<<17) /* flips vertically during
                                         decompression */
#define codecInfoDoesSkew       (1L<<18) /* skews image during
                                         decompression */
#define codecInfoDoesBlend      (1L<<19) /* blends image with matte during
                                         decompression */
#define codecInfoDoesWarp       (1L<<20) /* warps image arbitrarily during
                                         decompression */
#define codecInfoDoesRecompress (1L<<21) /* recompresses images without
                                         accumulating errors */
#define codecInfoDoesSpool      (1L<<22) /* uses data-loading or
                                         data-unloading function */
#define codecInfoDoesRateConstrain
                                         (1L<<23) /* constrains amount of generated
                                         data to caller-defined limit */

/* compressor and decompressor format flag bits */
#define codecInfoDepth1 (1L<<0) /* compressed images with 1-bit
                                color depth available */
#define codecInfoDepth2 (1L<<1) /* compressed images with 2-bit
                                color depth available */
#define codecInfoDepth4 (1L<<2) /* compressed images with 4-bit
                                color depth available */

```

Image Compressor Components

```

#define codecInfoDepth8 (1L<<3) /* compressed images with 8-bit
                                color depth available */
#define codecInfoDepth16(1L<<4) /* compressed images with 16-bit
                                color depth available */
#define codecInfoDepth32(1L<<5) /* compressed images with 32-bit
                                color depth available */
#define codecInfoDepth24(1L<<6) /* compressed images with 24-bit
                                color depth available */
#define codecInfoDepth33(1L<<7) /* compressed data with monochrome images of
                                1-bit color depth */
#define codecInfoDepth34(1L<<8) /* compressed images with
                                2-bit grayscale depth available */
#define codecInfoDepth36(1L<<9) /* compressed images with 4-bit grayscale
                                depth available */
#define codecInfoDepth40(1L<<10) /* compressed images with 8-bit grayscale
                                depth available */
#define codecInfoStoresClut
                                (1L<<11) /* compressed data with custom color
                                tables */
#define codecInfoDoesLossless
                                (1L<<12) /* compressed data stored lossless format */
#define codecInfoSequenceSensitive
                                (1L<<13) /* compressed data requires non-key frames
                                to be compressed in same order as
                                compressed */

```

Data Types

```

typedef struct {
    long        flags;           /* control information */
    short       wantedPixelSize; /* pixel depth for component to use
                                with image */
    short       extendWidth;     /* extension width of image in pixels */
    short       extendHeight;    /* extension height of image in pixels */
    short       bandMin;         /* supported minimum image band height */
    short       bandInc;         /* common factor of supported band
                                heights */
    short       pad;             /* reserved */
    unsigned long time;          /* milliseconds operation takes to
                                complete */
} CodecCapabilities;
typedef CodecCapabilities *CodecCapabilitiesPtr;

```

Image Compressor Components

```

typedef struct {
    ImageSequence      sequenceID;      /* sequence identifier ID
                                         (precompress or
                                         bandcompress) */

    ImageDescriptionHandle  imageDescription; /* handle to image
                                         description structure
                                         (precompress or
                                         bandcompress) */

    Ptr                data;            /* location for receipt of
                                         compressed image data */

    long               bufferSize;      /* size of buffer for data */
    long               frameNumber;     /* frame identifier */
    long               startLine;       /* starting line for band */
    long               stopLine;        /* ending line for band */
    long               conditionFlags;  /* condition flags */
    CodecFlags         callerFlags;     /* control info flags */
    CodecCapabilities   *capabilities;  /* pointer to compressor
                                         capability structure */

    ProgressProcRecord  progressProcRecord; /* progress function
                                         structure */

    CompletionProcRecord completionProcRecord; /* completion function
                                         structure */

    FlushProcRecord     flushProcRecord; /* data-unloading function
                                         structure */

    PixMap              srcPixMap;      /* pointer to image
                                         (precompress or
                                         bandcompress) */

    PixMap              prevPixMap;     /* pointer to pixel map
                                         for previous image */

    CodecQ              spatialQuality; /* compressed image
                                         quality */

    CodecQ              temporalQuality; /* sequence temporal
                                         quality */

    Fixed              similarity;      /* similarity between
                                         adjacent frames */

    DataRateParamsPtr   dataRateParams; /* pointer to the data rate
                                         parameters structure */

    long               reserved;        /* reserved */
} CodecCompressParams;
typedef CodecCompressParams *CodecCompressParamsPtr;

```

Image Compressor Components

```

typedef struct {
    ImageSequence          sequenceID;          /* unique sequence ID
                                                (predecompress,
                                                band decompress) */

    ImageDescriptionHandle imageDescription;     /* handle to image
                                                description structure
                                                (predecompress,
                                                band decompress) */

    Ptr                   data;                 /* compressed image data */
    long                  bufferSize;           /* size of data buffer */
    long                  frameNumber;          /* frame identifier */
    long                  startLine;            /* starting line for band */
    long                  stopLine;             /* ending line for band */
    long                  conditionFlags;       /* condition flags */
    CodecFlags            callerFlags;          /* control flags */
    CodecCapabilities     *capabilities;        /* pointer to compressor
                                                capability structure
                                                (predecompress,
                                                band decompress) */

    ProgressProcRecord    progressProcRecord;   /* progress function
                                                structure */

    CompletionProcRecord  completionProcRecord; /* completion function
                                                structure */

    DataProcRecord        dataProcRecord;       /* data-loading function
                                                structure */

    CGrafPtr              port;                 /* pointer to color
                                                graphics port for image
                                                (predecompress,
                                                band decompress) */

    PixMap                dstPixMap;            /* destination pixel map
                                                (predecompress,
                                                band decompress) */

    BitMapPtr             maskBits;             /* update mask */
    PixMapPtr             mattePixMap;          /* blend matte pixel map */
    Rect                  srcRect;              /* source rectangle
                                                (predecompress,
                                                band decompress) */

    MatrixRecord           *matrix;             /* pointer to matrix
                                                structure
                                                (predecompress,
                                                band decompress) */
}

```


Image Compressor Components

```

    CodecQ          accuracy;          /* desired accuracy
                                         (predecompress,
                                         band decompress) */

    short           transferMode;       /* transfer mode
                                         (predecompress,
                                         band decompress) */

    long            reserved[2];        /* reserved */
} CodecDecompressParams;

typedef CodecDecompressParams *CodecDecompressParamsPtr;

/* progress function structure */
typedef struct ProgressProcRecord ProgressProcRecord;
typedef ProgressProcRecord *ProgressProcRecordPtr;

struct ProgressProcRecord {
    ProgressProcPtr progressProc; /* pointer to your progress function */
    long progressRefCon;          /* reference constant for use by
                                   your progress function */
};

/* completion function structure */
typedef struct CompletionProcRecord CompletionProcRecord;
typedef CompletionProcRecord *CompletionProcRecordPtr;

struct CompletionProcRecord {
    CompletionProcPtr completionProc; /* pointer to completion function */
    long completionRefCon;            /* reference constant used by
                                   completion function */
};

/* data-loading structure */
typedef struct DataProcRecord DataProcRecord;
typedef DataProcRecord *DataProcRecordPtr;

struct DataProcRecord {
    DataProcPtr dataProc;             /* pointer to data-loading function */
    long dataRefCon;                  /* reference constant used by
                                   data-loading function */
};

/* data-unloading structure */
typedef struct FlushProcRecord FlushProcRecord;
typedef FlushProcRecord *FlushProcRecordPtr;

```

Image Compressor Components

```

struct FlushProcRecord {
    FlushProcPtr flushProc; /* pointer to data-unloading function */
    long flushRefCon;       /* reference constant used by data-unloading
                             function */
};

```

Functions

Direct Functions

```

pascal ComponentResult CDGetCodecInfo
    (CodecInfo *info);

pascal ComponentResult CDGetMaxCompressionSize
    (PixMapHandle src, const Rect *srcRect,
     short depth, CodecQ quality, long *size);

pascal ComponentResult CDGetCompressionTime
    (PixMapHandle src, const Rect *srcRect,
     short depth, CodecQ *spatialQuality,
     CodecQ *temporalQuality, unsigned long *time);

pascal ComponentResult CDGetSimilarity
    (PixMapHandle src, const Rect *srcRect,
     ImageDescriptionHandle desc, Ptr data,
     Fixed *similarity);

pascal ComponentResult CDGetCompressedImageSize
    (ImageDescriptionHandle desc, Ptr data,
     long bufferSize, DataProcRecordPtr dataProc,
     long *dataSize);

pascal ComponentResult CDTrimImage
    (ImageDescriptionHandle desc, Ptr inData,
     long inBufferSize, DataProcRecordPtr dataProc,
     Ptr outData, long outBufferSize,
     FlushProcRecordPtr flushProc, Rect *trimRect,
     ProgressProcRecordPtr progressProc);

pascal ComponentResult CDCodecBusy
    (ImageSequence seq);

```

Indirect Functions

```

pascal ComponentResult CDPreCompress
    (CodecCompressParams *params);

pascal ComponentResult CDBandCompress
    (CodecCompressParams *params);

pascal ComponentResult CDPreDecompress
    (CodecDecompressParams *params);

```

Image Compressor Components

```
pascal ComponentResult CDBandDecompress
    (CodecDecompressParams *params);
```

Image Compression Manager Utility Functions

```
pascal OSErr SetImageDescriptionExtension
    (ImageDescriptionHandle desc, Handle extension,
     long idType);

pascal OSErr GetImageDescriptionExtension
    (ImageDescriptionHandle desc,
     Handle *extension, long idType, long index);

pascal OSErr RemoveImageDescriptionExtension
    (ImageDescription **desc, long type,
     long index);

pascal OSErr CountImageDescriptionExtensionType
    (ImageDescription **desc, long type,
     long *count);

pascal OSErr GetNextImageDescriptionExtensionType
    (ImageDescription **desc, long *type);
```

Pascal Summary

Constants

```
CONST
compressorComponentType      = 'imco'; {compressor component type}
decompressorComponentType    = 'imdc'; {decompressor component type}

    {selector values}
codecGetCodecInfo             = $00;   {CDGetCodecInfo}
codecGetCompressionTime       = $01;   {CDGetCompressionTime}
codecGetMaxCompressionSize    = $02;   {CDGetMaxCompressionSize}
codecPreCompress              = $03;   {CDPreCompress}
codecBandCompress             = $04;   {CDBandCompress}
codecPreDecompress            = $05;   {CDPreDeCompress}
codecBandDecompress           = $06;   {CDBandDeCompress}
codecCDSequenceBusy           = $07;   {CDSequenceBusy}
codecGetCompressedImageSize    = $08;   {CDGetCompressedImageSize}
codecGetSimilarity            = $09;   {CDGetSimilarity}
codecTrimImage                = $0a;   {CDTrimImage}
```

Image Compressor Components

```

{image compressor component capabilities flags}
codecCanScale           = $1;      {decompressor scales information}
codecCanMask            = $2;      {decompressor applies mask to image}
codecCanMatte           = $4;      {decompressor blends using matte}
codecCanTransform       = $8;      {decompressor works with complex }
                                { placement matrices}

codecCanTransferMode    = $10;     {decompressor accepts transfer mode}
codecCanCopyPrev        = $20;     {compressor updates previous buffer}
codecCanSpool           = $40;     {component uses functions to spool }
                                { data}

codecCanClipVertical    = $80;     {decompressor clips vertically}
codecCanClipRectangular = $100;    {decompressor clips vertically }
                                { & horizontally}

codecCanRemapColor      = $200;    {compressor remaps color}
codecCanFastDither      = $400;    {compressor does fast dithering}
codecCanSrcExtract      = $800;    {compressor extracts portion of }
                                { source image}

codecCanCopyPrevComp    = $1000;   {compressor updates previous buffer}
codecCanAsync           = $2000;   {component works asynchronously}
codecCanMakeMask        = $4000;   {decompressor makes masks}
codecCanShift           = $8000;   {component works with pixels }
                                { that are not byte-aligned}

{condition flags}
codecConditionFirstBand  = $1;      {first band in frame}
codecConditionLastBand   = $2;      {last band in frame}
codecConditionFirstFrame = $4;      {(input) first frame to be }
                                { decompressed in this }
                                { sequence}

codecConditionNewDepth   = $8;      {(input) depth of }
                                { destination}

codecConditionNewTransform = $10;    {(input) transformation }
                                { matrix has changed}

codecConditionNewSrcRect = $20;      {(input) source rectangle}
codecConditionNewMask    = $40;      {(input) mask bitmap }
                                { has changed}

codecConditionNewMatte   = $80;      {(input) matte pixel map}
codecConditionNewTransferMode = $100; {(input) transfer mode}
codecConditionNewClut    = $200;     {(input) color lookup table}
codecConditionNewAccuracy = $400;    {accuracy parameter has }
                                { changed}

codecConditionNewDestination = $800; {(input) destination pixel }
                                { map}

codecConditionCodecChangedMask = $80000000; {changed mask bits}

```

Image Compressor Components

```

{CodecInfo compressFlags and deCompressFlags bits}
codecInfoDoes1          = $1;      {works with 1-bit pixel maps}
codecInfoDoes2          = $2;      {works with 2-bit pixel maps}
codecInfoDoes4          = $4;      {works with 4-bit pixel maps}
codecInfoDoes8          = $8;      {works with 8-bit pixel maps}
codecInfoDoes16         = $10;     {works with 16-bit pixel maps}
codecInfoDoes32         = $20;     {works with 32-bit pixel maps}
codecInfoDoesDither     = $40;     {supports fast dithering}
codecInfoDoesStretch    = $80;     {stretches to arbitrary sizes}
codecInfoDoesShrink     = $100;    {shrinks to arbitrary sizes}
codecInfoDoesMask       = $200;    {handles clipping regions}
codecInfoDoesTemporal   = $400;    {sequential temporal }
                                { compression}
codecInfoDoesDouble     = $800;    {stretches to double size}
codecInfoDoesQuad       = $1000;   {stretches to quadruple size}
codecInfoDoesHalf       = $2000;   {shrinks to half size}
codecInfoDoesQuarter    = $4000;   {shrinks to one-quarter size}
codecInfoDoesRotate     = $8000;   {rotates during decompression}
codecInfoDoesHorizFlip  = $10000;  {flips horizontally}
codecInfoDoesVertFlip   = $20000;  {flips vertically}
codecInfoDoesSkew       = $40000;  {skews image during }
                                { decompression}
codecInfoDoesBlend      = $80000;  {blends image with matte }
                                { during decompression}
codecInfoDoesWarp       = $100000; {warps image during }
                                { decompression}
codecInfoDoesRecompress = $200000; {recompresses images}
codecInfoDoesSpool      = $400000; {uses data-loading }
                                { or unloading functions}
codecInfoDoesRateConstrain = $800000; {constrains amount of generated }
                                { data to caller-defined limit}

{codecInfo formatFlags bits}
codecInfoDepth1         = $1;      {color images with 1-bit color depth}
codecInfoDepth2         = $2;      {color images with 2-bit color depth}
codecInfoDepth4         = $4;      {color images with 4-bit color depth}
codecInfoDepth8         = $8;      {color images with 8-bit color depth}
codecInfoDepth16        = $10;     {color images with 16-bit color depth}
codecInfoDepth32        = $20;     {color images with 32-bit color depth}
codecInfoDepth24        = $40;     {color images with 24-bit color depth}
codecInfoDepth33        = $80;     {monochrome images with 1-bit color }
                                { depth}
codecInfoDepth34        = $100;    {grayscale images with 2-bit }
                                { grayscale depth}

```

Image Compressor Components

```

codecInfoDepth36          = $200; { grayscale images with 4-bit }
                                { grayscale depth }
codecInfoDepth40          = $400; { grayscale images with 8-bit }
                                { grayscale depth }
codecInfoStoresClut       = $800; { custom color tables }
codecInfoDoesLossless     = $1000; { lossless compression or }
                                { decompression operations }
codecInfoSequenceSensitive = $2000; { compression data requires non-key }
                                { frames to be decompressed in same }
                                { order as compressed }

```

Data Types

```

TYPE  CodecCapabilities =
  RECORD
    flags:           LongInt;    { control information }
    wantedPixelSize: Integer;    { pixel depth for component to use }
                                { with image }
    extendWidth:     Integer;    { extension width of image }
    extendHeight:    Integer;    { extension height of image }
    bandMin:         Integer;    { supported minimum band height }
    bandInc:         Integer;    { common factor of band heights }
    pad:             Integer;    { reserved }
    time:            Integer;    { milliseconds to completion }
  END;

CodecCapabilitiesPtr      = ^CodecCapabilities;

CodecCompressParams =
  RECORD
    sequenceID:       ImageSequence; { sequence identifier ID }
    imageDescription: ImageDescriptionHandle;
                                { handle to image }
                                { description record }
    data:             Ptr;        { location for receipt of }
                                { compressed image data }
    bufferSize:       LongInt;    { size of buffer for data }
    frameNumber:      LongInt;    { frame identifier }
    startLine:        LongInt;    { starting line for band }
    stopLine:         LongInt;    { ending line for band }
    conditionFlags:   LongInt;    { condition flags }
    callerFlags:      CodecFlags; { control information flags }
  END;

```

Image Compressor Components

```

capabilities:          CodecCapabilitiesPtr;
                        {pointer to compressor }
                        { capability record }
progressProcRecord:   ProgressProcRecord;
                        {progress function record}
completionProcRecord:CompletionProcRecord;
                        {completion function }
                        { record}
flushProcRecord:      FlushProcRecord;
                        {data-unloading function }
                        { record}
srcPixMap:            PixMap;          {pointer to image}
prevPixMap:           PixMap;          {pointer to pixel map }
                        { for previous image}
spatialQuality:       CodecQ;          {compressed image quality}
temporalQuality:      CodecQ;          {sequence temporal quality}
similarity:           Fixed;           {similarity between }
                        { adjacent frames}
dataRateParams        dataRateParamsPtr;
                        {pointer to the data rate }
                        { parameters record}
reserved:             ARRAY[0..1] OF LongInt;
                        {reserved}

END;

CodecCompressParamsPtr = ^CodecCompressParams;

CodecDecompressParams =
RECORD
    sequenceID:        ImageSequence; {unique sequence ID}
    imageDescription:   ImageDescriptionHandle;
                        {handle to image }
                        { description record}
    data:              Ptr;            {compressed image data}
    bufferSize:        LongInt;        {size of data buffer}
    frameNumber:       LongInt;        {frame identifier}
    startLine:         LongInt;        {starting line for band}
    stopLine:          LongInt;        {ending line for band}
    conditionFlags:    LongInt;        {condition flags}
    callerFlags:       CodecFlags;     {control flags}
    capabilities:      CodecCapabilitiesPtr;
                        {pointer to compressor }
                        { capability record}

```

Image Compressor Components

```

progressProcRecord:    ProgressProcRecord;
                        {progress function record}
completionProcRecord:  CompletionProcRecord;
                        {completion function record}
dataProcRecord:        DataProcRecord; {data-loading function }
                        { record}
port:                  CGrafPtr;         {pointer to color }
                        { grafport for image}
dstPixMap:             PixMap;           {destination pixel map}
maskBits:              BitMapPtr;        {update mask}
mattePixMap:           PixMapPtr;        {blend matte pixel map}
srcRect:               Rect;             {source rectangle}
matrix:                MatrixRecordPtr;
                        {pointer to matrix }
                        { structure}
accuracy:              CodecQ;           {desired accuracy}
transferMode:          Integer;          {transfer mode}
reserved:              ARRAY[0..1] OF LongInt;
                        {reserved}

END;

CodecDecompressParamsPtr = ^CodecDecompressParams;

ProgressProcRecordPtr = ^ProgressProcRecord;
ProgressProcRecord =
RECORD
    progressProc:    ProgressProcPtr; {pointer to progress function}
    progressRefCon:  LongInt;          {progress function }
                                { reference constant}
END;

CompletionProcRecordPtr = ^CompletionProcRecord;
CompletionProcRecord =
RECORD
    completionProc:  CompletionProcPtr; {pointer to completion function}
    completionRefCon: LongInt;           {completion function reference }
                                { constant}
END;

DataProcRecordPtr = ^DataProcRecord;
DataProcRecord =
RECORD
    dataProc:    DataProcPtr; {pointer to data-loading function}

```


Image Compressor Components

```

dataRefCon:    LongInt;           {data-loading function }
                                   { reference constant}

END;

FlushProcRecordPtr  = ^FlushProcRecord;
FlushProcRecord =
RECORD
    flushProc:    FlushProcPtr;    {pointer to data-unloading function}
    flushRefCon:  LongInt;         {data-unloading function reference }
                                   { constant}

END;

```

Routines
Direct Functions

```

FUNCTION CDGetCodecInfo      (VAR info: CodecInfo): ComponentResult;
FUNCTION CDGetMaxCompressionSize
    (src: PixMapHandle; srcRect: Rect;
     depth: Integer; quality: CodecQ;
     VAR size: LongInt): ComponentResult;
FUNCTION CDGetCompressionTime
    (src: PixMapHandle; srcRect: Rect;
     depth: Integer; VAR spatialQuality: CodecQ;
     VAR temporalQuality: CodecQ;
     VAR time: LongInt): ComponentResult;
FUNCTION CDGetSimilarity    (src: PixMapHandle; srcRect: Rect;
     desc: ImageDescriptionHandle; data: Ptr;
     VAR similarity: Fixed): ComponentResult;
FUNCTION CDGetCompressedImageSize
    (desc: ImageDescriptionHandle; data: Ptr;
     bufferSize: LongInt;
     dataProc: DataProcRecordPtr;
     VAR dataSize: LongInt): ComponentResult;
FUNCTION CDTrimImage        (desc: ImageDescriptionHandle; inData: Ptr;
     inBufferSize: LongInt;
     dataProc: DataProcRecordPtr; outData: Ptr;
     outBufferSize: LongInt;
     flushProc: FlushProcRecordPtr;
     VAR trimRect: Rect;
     progressProc: ProgressProcRecordPtr):
     ComponentResult;
FUNCTION CDCodecBusy        (seq: ImageSequence): ComponentResult;

```

Image Compressor Components

Indirect Functions

```

FUNCTION CDPreCompress      (params: CodecCompressParamsPtr):
                             ComponentResult;

FUNCTION CDBandCompress     (params: CodecCompressParamsPtr):
                             ComponentResult;

FUNCTION CDPreDecompress    (params: CodecCompressParamsPtr):
                             ComponentResult;

FUNCTION CDBandDecompress   (params: CodecCompressParamsPtr):
                             ComponentResult;

```

Image Compression Manager Utility Functions

```

FUNCTION SetImageDescriptionExtension
                             (desc: ImageDescriptionHandle;
                              extension: Handle; idType: LongInt): OSErr;

FUNCTION GetImageDescriptionExtension
                             (desc: ImageDescriptionHandle;
                              VAR extension: Handle; idType: LongInt;
                              index: LongInt): OSErr;

FUNCTION RemoveImageDescriptionExtension
                             (desc: ImageDescriptionHandle; idType: LongInt;
                              index: LongInt): OSErr;

FUNCTION CountImageDescriptionExtensionType
                             (desc: ImageDescriptionHandle; idType: LongInt;
                              VAR count: LongInt): OSErr;

FUNCTION GetNextImageDescriptionExtensionType
                             (desc: ImageDescriptionHandle;
                              VAR idType: LongInt): OSErr;

```

Result Codes

codecErr	-8960	General error returned by compressor; can be returned by any function that gets handled by the compressor
noCodecErr	-8961	Image Compression Manager could not find specified error
codecUnimpErr	-8962	Feature not implemented by this compressor
codecSpoolErr	-8966	Error loading or unloading data
codecAbortErr	-8967	Operation aborted by progress function
codecExtensionNotFoundErr	-8971	Requested extension is not in the image description structure
codecOpenErr	-8973	Compressor component could not be opened by the Image Compression Manager