

## CompressPicture

The `CompressPicture` function compresses a single-frame image stored as a picture structure and places the result in another picture. If a picture with multiple pixel maps and other graphical objects is passed, the pixel maps will be compressed individually and the other graphic objects will not be affected.

```
pascal OSErr CompressPicture (PicHandle srcPicture,
                              PicHandle dstPicture,
                              CodecQ quality, CodecType cType);
```

`srcPicture`

Contains a handle to the source image, stored as a picture.

`dstPicture`

Contains a handle to the destination for the compressed image. The compressor resizes this handle for the result data.

`quality`

Specifies the desired compressed image quality. See “Compression Quality Constants” beginning on page 3-61 for valid values.

`cType`

Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-68 for a list of the available compressor types). If the value passed in is 0, or 'raw ', and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.

### DESCRIPTION

The `CompressPicture` function compresses only image data. Any other types of data in the picture, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture.

This function does not use the graphics port.

If your picture does not fit in memory, use the `CompressPictureFile` function, which is described on page 3-97.

This function supports parameters governing image quality and compressor type. The compressor infers the other compression parameters from the image data in the source picture.

### SPECIAL CONSIDERATIONS

The `CompressPicture` function doesn't compress pictures that contain compressed data. Do not alter data in pictures that are already compressed. Instead use `FCompressPicture`, described in the next section.

## Image Compression Manager

## 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	The Image Compression Manager could not find the specified compressor

## SEE ALSO

If you need more control over the compression operation than is provided by the `CompressPicture` function, use the `FCompressPicture` function.

**FCompressPicture**

---

The `FCompressPicture` function compresses a single-frame image stored as a picture structure and places the result in another picture. If a picture with multiple pixel maps and other graphical objects is passed, the pixel maps will be compressed individually and the other graphic objects will not be affected.

```
pascal OSErr FCompressPicture (PicHandle srcPicture,
                               PicHandle dstPicture,
                               short colorDepth,
                               CTabHandle clut,
                               CodecQ quality,
                               short doDither,
                               short compressAgain,
                               ProgressProcRecordPtr progressProc,
                               CodecType cType,
                               CompressorComponent codec);
```

`srcPicture`      Contains a handle to the source image, stored as a picture.

`dstPicture`      Contains a handle to the destination for the compressed image. The compressor resizes this handle for the result data.

`colorDepth`      Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure

## Image Compression Manager

	returned by the <code>GetCodecInfo</code> function (see “Getting Information About Compressor Components,” which begins on page 3-66, for more information).						
<code>clut</code>	Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the <code>colorDepth</code> parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the <code>ctSeed</code> field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the <code>colorDepth</code> parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the <code>clut</code> parameter when <code>colorDepth</code> is set to 33, 34, 36, or 40. If you set this parameter to <code>nil</code> , the compressor uses the color lookup table from the source pixel map.						
<code>quality</code>	Specifies the desired compressed image quality. See “Compression Quality Constants” beginning on page 3-61 for available values.						
<code>doDither</code>	Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available: <table> <tr> <td><code>defaultDither</code></td><td>Indicates that the dithering in the source file is to be respected.</td></tr> <tr> <td><code>forceDither</code></td><td>Indicates that the specified image should be dithered whether the source uses dithering or not.</td></tr> <tr> <td><code>suppressDither</code></td><td>Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into a 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.</td></tr> </table>	<code>defaultDither</code>	Indicates that the dithering in the source file is to be respected.	<code>forceDither</code>	Indicates that the specified image should be dithered whether the source uses dithering or not.	<code>suppressDither</code>	Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into a 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.
<code>defaultDither</code>	Indicates that the dithering in the source file is to be respected.						
<code>forceDither</code>	Indicates that the specified image should be dithered whether the source uses dithering or not.						
<code>suppressDither</code>	Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into a 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.						
<code>compressAgain</code>	Indicates whether to recompress compressed image data in the picture. Use this parameter to control whether any compressed image data that is in the source picture should be decompressed and then recompressed using the current parameters. Set the value of this parameter to <code>true</code> to recompress such data. Set the value of the parameter to <code>false</code> to leave the data as it is. Note that recompressing the data may have undesirable side effects, including image quality degradation.						
<code>progressProc</code>	Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on						

## Image Compression Manager

	page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function.								
<code>cType</code>	Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-68 for a list of the available compressor types). If the value passed in is <code>0</code> , or <code>'raw'</code> , the resulting picture is not compressed and does not require QuickTime to be displayed.								
<code>codec</code>	Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers: <table> <tr> <td><code>anyCodec</code></td><td>Choose the first compressor of the specified type</td></tr> <tr> <td><code>bestSpeedCodec</code></td><td>Choose the fastest compressor of the specified type</td></tr> <tr> <td><code>bestFidelityCodec</code></td><td>Choose the most accurate compressor of the specified type</td></tr> <tr> <td><code>bestCompressionCodec</code></td><td>Choose the compressor that produces the smallest resulting data</td></tr> </table> <p>You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a <code>codec</code> field and want to make sure that the specified instance is used for that operation.</p>	<code>anyCodec</code>	Choose the first compressor of the specified type	<code>bestSpeedCodec</code>	Choose the fastest compressor of the specified type	<code>bestFidelityCodec</code>	Choose the most accurate compressor of the specified type	<code>bestCompressionCodec</code>	Choose the compressor that produces the smallest resulting data
<code>anyCodec</code>	Choose the first compressor of the specified type								
<code>bestSpeedCodec</code>	Choose the fastest compressor of the specified type								
<code>bestFidelityCodec</code>	Choose the most accurate compressor of the specified type								
<code>bestCompressionCodec</code>	Choose the compressor that produces the smallest resulting data								

## DESCRIPTION

The `FCompressPicture` function compresses only image data. Any other types of data in the picture, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture.

This function supports parameters governing image quality, compressor type, image depth, custom color tables, and dithering.

## RESULT CODES

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

## SEE ALSO

If you do not need such a high degree of control over the compression operation, use the `CompressPicture` function, described on page 3-93.

## CompressPictureFile

---

The `CompressPictureFile` function compresses a single-frame image stored as a picture file (PICT file) and places the result in another picture file.

```
pascal OSErr CompressPictureFile (short srcRefNum,
                                   short dstRefNum,
                                   CodecQ quality,
                                   CodecType cType);
```

<code>srcRefNum</code>	Contains a file reference number for the source PICT file.
<code>dstRefNum</code>	Contains a file reference number for the destination PICT file. Note that the compressor overwrites the contents of the file referred to by <code>dstRefNum</code> . You must open this file with write permission. The destination file can be the same as the source file specified by the <code>srcRefNum</code> parameter.
<code>quality</code>	Specifies the desired compressed image quality. See “Compression Quality Constants” on page 3-61 for available values.
<code>cType</code>	Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-68 for a list of the available compressor types). If the value passed in is 0, or 'raw', and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.

### DESCRIPTION

The `CompressPictureFile` function compresses only image data. Any other types of data in the file, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture. This function does not use the graphics port.

This function supports parameters governing image quality and compressor type. The compressor infers the other compression parameters from the image data in the source picture file.

### SPECIAL CONSIDERATIONS

The `CompressPictureFile` function doesn't compress pictures that contain compressed data. Do not alter data in pictures that are already compressed. Instead use `FCompressPictureFile`, described in the next section.

## Image Compression Manager

## 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

File Manager errors

## SEE ALSO

If you need more control over the compression operation, use the `FCompressPictureFile` function.

## FCompressPictureFile

---

The `FCompressPictureFile` function compresses a single-frame image stored as a picture file (PICT file) and places the result in another picture file.

```
pascal OSErr FCompressPictureFile (short srcRefNum,
                                   short dstRefNum, short colorDepth,
                                   CTabHandle clut, CodecQ quality,
                                   short doDither,
                                   short compressAgain,
                                   ProgressProcRecordPtr progressProc,
                                   CodecType cType,
                                   CompressorComponent codec);
```

`srcRefNum` Specifies a file reference number for the source PICT file.

`dstRefNum` Specifies a file reference number for the destination PICT file. Note that the compressor overwrites the contents of the file referred to by `dstRefNum`. You must open this file with write permissions. The destination file may be the same as the source file specified by the `srcRefNum` parameter.

`colorDepth` Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor capability structure returned by the `GetCodecInfo` function (see “Getting Information About Compressor Components,” which begins on page 3-66, for more information).

## Image Compression Manager

<code>clut</code>	Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the <code>colorDepth</code> parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the <code>ctSeed</code> field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the <code>colorDepth</code> parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the <code>clut</code> parameter when <code>colorDepth</code> is set to 33, 34, 36, or 40. If you set this parameter to <code>nil</code> , the compressor uses the color lookup table from the source pixel map.
<code>quality</code>	Specifies the desired compressed image quality. See “Compression Quality Constants” on page 3-61 for available values.
<code>doDither</code>	Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:
<code>defaultDither</code>	Indicates that the dithering in the source file is to be respected.
<code>forceDither</code>	Indicates that the specified image should be dithered whether the source uses dithering or not.
<code>suppressDither</code>	Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit, color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.
<code>compressAgain</code>	Indicates whether to recompress compressed image data in the picture. Use this parameter to control whether any compressed image data that is in the source picture should be decompressed and then recompressed using the current parameters. Set the value of this parameter to <code>true</code> to recompress such data. Set the value of this parameter to <code>false</code> to leave the data as it is. Note that recompressing the data may have undesirable side effects, including image quality degradation.
<code>progressProc</code>	Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function.

## Image Compression Manager

<code>cType</code>	Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-68 for a list of the available compressor types). If the value passed in is 0, or 'raw' and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.								
<code>codec</code>	Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers: <table> <tr> <td><code>anyCodec</code></td><td>Choose the first compressor of the specified type</td></tr> <tr> <td><code>bestSpeedCodec</code></td><td>Choose the fastest compressor of the specified type</td></tr> <tr> <td><code>bestFidelityCodec</code></td><td>Choose the most accurate compressor of the specified type</td></tr> <tr> <td><code>bestCompressionCodec</code></td><td>Choose the compressor that produces the smallest resulting data</td></tr> </table> <p>You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a <code>codec</code> field and want to make sure that the specified instance is used for that operation.</p>	<code>anyCodec</code>	Choose the first compressor of the specified type	<code>bestSpeedCodec</code>	Choose the fastest compressor of the specified type	<code>bestFidelityCodec</code>	Choose the most accurate compressor of the specified type	<code>bestCompressionCodec</code>	Choose the compressor that produces the smallest resulting data
<code>anyCodec</code>	Choose the first compressor of the specified type								
<code>bestSpeedCodec</code>	Choose the fastest compressor of the specified type								
<code>bestFidelityCodec</code>	Choose the most accurate compressor of the specified type								
<code>bestCompressionCodec</code>	Choose the compressor that produces the smallest resulting data								

## DESCRIPTION

The `FCompressPicture` function compresses only image data. Any other types of data in the file, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture file.

This function supports parameters governing image quality, compressor type, image depth, custom color tables, and dithering.

## 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	The Image Compression Manager could not find the specified compressor
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function
File Manager errors		



## SEE ALSO

If you do not need such a high degree of control over the compression operation, use the `CompressPictureFile` function, described on page 3-97.

## DrawPictureFile

---

The `DrawPictureFile` function draws an image from a specified picture file (PICT file) in the current graphics port. Your program also specifies the destination rectangle for the image.

```
pascal OSErr DrawPictureFile (short refNum, const Rect *frame,
                             ProgressProcRecordPtr progressProc);
```

<code>refNum</code>	Contains a file reference number for the source PICT file.
<code>frame</code>	Contains a pointer to the rectangle into which the image is to be loaded. The compressor scales the source image to fit into this destination rectangle.
<code>progressProc</code>	Points to a progress function structure. During the operation, the draw function may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function.

## DESCRIPTION

The `DrawPictureFile` function is essentially the same as QuickDraw’s `DrawPicture` routine, except that `DrawPictureFile` reads the picture from disk. (For details on `DrawPicture`, see *Inside Macintosh: Imaging*.) The Image Compression Manager performs any spooling that may be necessary when reading the picture file.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function
File Manager errors		

## DrawTrimmedPicture

---

The `DrawTrimmedPicture` function draws an image that is stored as a picture into the current graphics port and trims that image to fit a region you specify.

```
pascal OSErr DrawTrimmedPicture (PicHandle srcPicture,
                                const Rect *frame, RgnHandle trimMask,
                                short doDither,
                                ProgressProcRecordPtr progressProc);
```

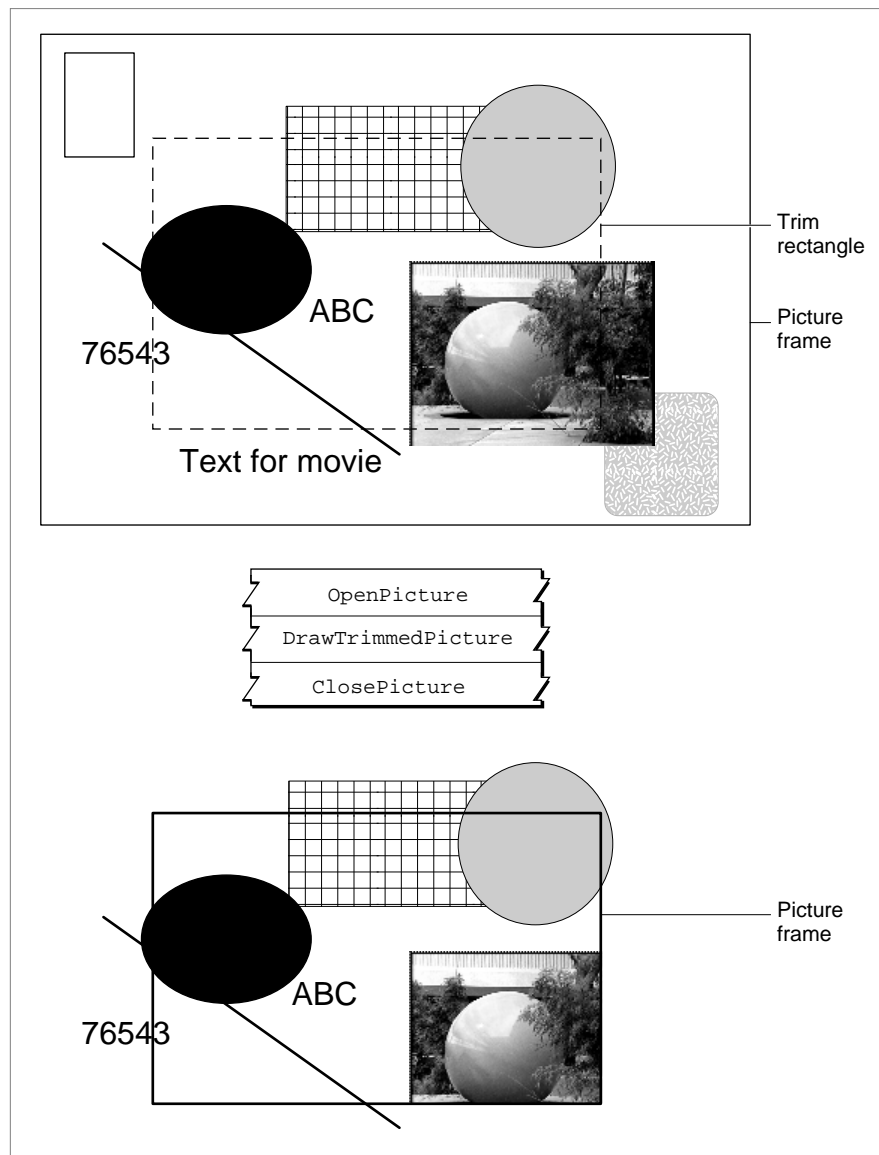
- `srcPicture` Contains a handle to the source image; stored as a picture.
- `frame` Contains a pointer to the rectangle into which the decompressed image is to be loaded.
- `trimMask` Contains a handle to a clipping region in the destination coordinate system. The decompressor applies this mask to the destination image and ignores any image data that fall outside the specified region. Set this parameter to `nil` if you do not want to clip the source image. In this case, this function acts like QuickDraw's `DrawPicture` routine, but it also allows you to control dithering and assign a progress function. (See *Inside Macintosh: Imaging* for more on `DrawPicture`.)
- `doDither` Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:
- `defaultDither`  
Indicates that the dithering in the source file is to be respected.
  - `forcedDither`  
Indicates that the specified image should be dithered whether the source uses dithering or not.
  - `suppressDither`  
Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit, color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.
- `progressProc` Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of `-1`, you obtain a standard progress function.

## Image Compression Manager

## DESCRIPTION

The `DrawTrimmedPicture` function works with compressed image data—the source data stays compressed. The function trims the image to fit the specified clipping region. Figure 3-10 shows how the `DrawTrimmedPicture` function works. It illustrates how you can use this function to save part of a picture (the clipped or uncompressed image data that is not within the trim region is ignored and is not included in the destination picture). All the remaining objects in the resulting image are clipped. You use QuickDraw's `OpenPicture` and `ClosePicture` routines to open and close the destination picture. (For more on `OpenPicture` and `ClosePicture`, see *Inside Macintosh: Imaging*.)

Note that if you just use a clip while making a picture, the data—though not visible—is still stored in the picture.

**Figure 3-10** The operation of the `DrawTrimmedPicture` function**RESULT CODES**

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

**SEE ALSO**

If your source image does not fit in memory, use the `DrawTrimmedPictureFile` function, which is described in the next section.

## DrawTrimmedPictureFile

---

The `DrawTrimmedPictureFile` function draws an image that is stored as a picture file (PICT file) into the current graphics port and trims that image to fit a region you specify.

```
pascal OSErr DrawTrimmedPictureFile (short srcRefnum,
                                     const Rect *frame,
                                     RgnHandle trimMask,
                                     short doDither,
                                     ProgressProcRecordPtr progressProc);
```

<code>srcRefNum</code>	Contains a file reference number for the source PICT file.
<code>frame</code>	Contains a pointer to the rectangle into which the decompressed image is to be loaded.
<code>trimMask</code>	Contains a handle to a clipping region in the destination coordinate system. The decompressor applies this mask to the destination image and ignores any image data that fall outside the specified region. Set this parameter to <code>nil</code> if you do not want to clip the source image. In this case, this function acts like the <code>DrawPictureFile</code> function, which is described on page 3-101.
<code>doDither</code>	Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available: <div> <div><code>defaultDither</code></div> <div>Indicates that the dithering in the source picture file is to be respected.</div> <div><code>forceDither</code></div> <div>Indicates that the specified image should be dithered whether the source uses dithering or not.</div> <div><code>suppressDither</code></div> <div>Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.</div> </div>
<code>progressProc</code>	Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function.

## Image Compression Manager

## DESCRIPTION

The `DrawTrimmedPictureFile` function works with compressed image data—the source data stays compressed. The function trims the image to fit the specified clipping region. The Image Compression Manager handles any spooling issues that may arise when reading the picture file.

You can use this function to save part of a picture, since the image data that is not within the trim region is ignored and is not included in the destination picture file. All the remaining objects in the resulting object are clipped.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

File Manager errors

## GetPictureFileHeader

---

The `GetPictureFileHeader` function extracts the picture frame (the `picFrame` rectangle in the picture structure) and file header from a specified picture file (PICT file). Your program can use this information to determine how to draw an image without having to read the picture file.

```
pascal OSErr GetPictureFileHeader (short refNum, Rect *frame,
                                   OpenCPicParams *header);
```

<code>refNum</code>	Contains a file reference number for the source PICT file.
<code>frame</code>	Contains a pointer to a rectangle that is to receive the picture frame rectangle of the picture file. This function places the <code>picFrame</code> rectangle from the picture structure into the rectangle referred to by the <code>frame</code> parameter. If you are not interested in this information, pass <code>nil</code> in this parameter.
<code>header</code>	Contains a pointer to an <code>OpenCPicture</code> parameters structure. The <code>GetPictureFileHeader</code> function places the header from the specified picture file into the structure referred to by the <code>header</code> parameter. Note that this function always returns a version 2 header. If the source file is a version 1 PICT file, the <code>GetPictureFileHeader</code> function converts the header into version 2 format before returning it to your application. See <i>Inside Macintosh: Imaging</i> for more information about picture headers and the <code>OpenCPicture</code> function. If you are not interested in this information, pass <code>nil</code> in this parameter.

**DESCRIPTION**

The `GetPictureFileHeader` function always returns a version 2 PICT file header. If the specified picture file is a version 1 file, the `GetPictureFileHeader` function synthesizes a version 2 header from the information available in the file header.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
File Manager errors		

## Making Thumbnail Pictures

---

This section describes the functions that allow your application to create thumbnail pictures from existing images that are stored as pixel maps, pictures, or picture files. Thumbnail pictures are useful for creating small, representative images of a source image. You can use thumbnails when you create previews for files that contain image data (for more information about file previews, see the chapter “Movie Toolbox” in this book).

You can create thumbnails from pictures, picture files, or pixel maps—use the `MakeThumbnailFromPicture`, `MakeThumbnailFromPictureFile`, or `MakeThumbnailFromPixMap` function, as appropriate.

### MakeThumbnailFromPicture

---

The `MakeThumbnailFromPicture` function creates an 80-by-80 pixel thumbnail picture from a specified picture structure.

```
pascal OSErr MakeThumbnailFromPicture (PicHandle picture,
                                       short colorDepth,
                                       PicHandle thumbnail,
                                       ProgressProcRecordPtr progressProc);
```

<code>picture</code>	Contains a handle to the image from which the thumbnail is to be extracted. The image must be stored in a picture structure.
<code>colorDepth</code>	Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
<code>thumbnail</code>	Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.

## Image Compression Manager

`progressProc`

Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of `-1`, you obtain a standard progress function.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

**MakeThumbnailFromPictureFile**

---

The `MakeThumbnailFromPictureFile` function creates an 80-by-80 pixel thumbnail picture from a specified picture file (PICT file).

```
pascal OSErr MakeThumbnailFromPictureFile (short refNum,
                                           short colorDepth,
                                           PicHandle thumbnail,
                                           ProgressProcRecordPtr progressProc);
```

`refNum`      Contains a file reference number for the PICT file from which the thumbnail is to be extracted.

`colorDepth`      Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`thumbnail`      Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.

`progressProc`

Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of `-1`, you obtain a standard progress function.



RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memFullErr	-108	Not enough memory available
codecAbortErr	-8967	Operation aborted by the progress function
File Manager errors		

MakeThumbnailFromPixMap

The MakeThumbnailFromPixMap function creates an 80-by-80 pixel thumbnail picture from a specified pixel map structure.

```
pascal OSErr MakeThumbnailFromPixMap (PixMapHandle src,
                                     const Rect *srcRect,
                                     short colorDepth,
                                     PicHandle thumbnail,
                                     ProgressProcRecordPtr progressProc);
```

src	Contains a handle to the image from which the thumbnail is to be extracted. The image must be stored in a pixel map structure.
srcRect	Contains a pointer to a rectangle defining the portion of the image to use for the thumbnail.
colorDepth	Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
thumbnail	Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.
progressProc	Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156, for more information about progress functions). This parameter contains a pointer to a structure that identifies that progress function. If you have not provided a progress function, set this parameter to nil. If you pass a value of -1, you obtain a standard progress function.

## Image Compression Manager

## DESCRIPTION

The thumbnail returned is an 80-by-80 pixel picture, but the aspect ratio is maintained.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>codecAbortErr</code>	-8967	Operation aborted by the progress function

## Working With Sequences

This section describes the functions that enable your application to compress and decompress sequences of images. Each image in the sequence is referred to as a *frame*. Note that the sequence carries no time information. The Movie Toolbox manages all temporal aspects of displaying the sequence. Consequently, your application can focus on the order of images in the sequence.

To process a sequence of frames, your program first begins the sequence (by issuing either the `CompressSequenceBegin` or `DecompressSequenceBegin` functions). You then process each frame in the sequence (use `CompressSequenceFrame` to compress a frame; use `DecompressSequenceFrame` to decompress a frame). When you are done, close the sequence by issuing the `CDSequenceEnd` function. You can check on the status of the current operation by calling the `CDSequenceBusy` function.

Note that the Image Compression Manager provides a rich set of functions that allow your application to control many of the parameters that govern sequence processing. You set default values for most of these parameters when you start the sequence. These additional functions allow you to modify those parameters while you are processing a sequence. See “Changing Sequence-Compression Parameters,” which begins on page 3-124, for information on functions that affect sequence compression. See “Changing Sequence-Decompression Parameters” beginning on page 3-133 for information on functions that affect sequence decompression.

## CompressSequenceBegin

Your application calls the `CompressSequenceBegin` function to signal the beginning of the process of compressing a sequence of frames. The Image Compression Manager prepares for the sequence-compression operation by reserving appropriate system resources. You must call this function before calling the `CompressSequenceFrame` function, which is described in the next section.

## Image Compression Manager

```

pascal OSErr CompressSequenceBegin (ImageSequence *seqID,
                                     PixMapHandle src,
                                     PixMapHandle prev,
                                     const Rect *srcRect,
                                     const Rect *prevRect,
                                     short colorDepth,
                                     CodecType cType,
                                     CompressorComponent codec,
                                     CodecQ spatialQuality,
                                     CodecQ temporalQuality,
                                     long keyFrameRate,
                                     CTabHandle clut,
                                     CodecFlags flags,
                                     ImageDescriptionHandle desc);

```

seqID	Contains a pointer to a field to receive the unique identifier for this sequence. You must supply this identifier to all subsequent Image Compression Manager functions that relate to this sequence.
src	Contains a handle to a pixel map that will contain the image to be compressed. The image must be stored in a pixel map structure.
prev	Contains a handle to a pixel map that will contain a previous image. The compressor uses this buffer to store a previous image against which the current image (stored in the pixel map referred to by the src parameter) is compared when performing temporal compression. This pixel map must be created at the same depth and with the same color table as the source image. The compressor manages the contents of this pixel map based upon several considerations, such as the key frame rate and the degree of difference between compared images. If you want the compressor to allocate this pixel map or if you do not want to perform temporal compression (that is, you have set the value of the temporalQuality parameter to 0), set this parameter to nil.  You can set or change the previous image buffer for an active sequence by calling the SetCSequencePrev function. You can obtain a pointer to a pixel map that was allocated by the compressor by calling the GetCSequencePrevBuffer function. See “Changing Sequence-Compression Parameters,” which begins on page 3-124, for information about these functions.
srcRect	Contains a pointer to a rectangle defining the portion of the image to compress. The compressor applies this rectangle to the image stored in the buffer referred to by the src parameter.

## Image Compression Manager

<code>prevRect</code>	<p>Contains a pointer to a rectangle defining the portion of the previous image to use for temporal compression. The compressor uses this portion of the previous image as the basis of comparison with the current image. The compressor ignores this parameter if you have not provided a buffer for previous images. This rectangle must be the same size as the source rectangle, which is specified with the <code>srcRect</code> parameter.</p> <p>You can set or change the rectangle used with the previous image buffer for an active sequence by calling the <code>SetCSequencePrev</code> function. See “Changing Sequence-Compression Parameters,” which begins on page 3-124, for information about this function.</p>
<code>colorDepth</code>	<p>Specifies the depth at which the sequence is likely to be viewed. Compressors may use this as an indication of the color or grayscale resolution of the compressed images. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the <code>GetCodecInfo</code> function (described on page 3-69).</p>
<code>cType</code>	<p>Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-68 for a list of the available compressor types).</p>
<code>codec</code>	<p>Specifies a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:</p> <ul style="list-style-type: none"> <li><code>anyCodec</code>     Choose the first compressor of the specified type</li> <li><code>bestSpeedCodec</code>     Choose the fastest compressor of the specified type</li> <li><code>bestFidelityCodec</code>     Choose the most accurate compressor of the specified type</li> <li><code>bestCompressionCodec</code>     Choose the compressor that produces the smallest resulting data</li> </ul> <p>You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a <code>codec</code> field and want to make sure that the specified instance is used for that operation.</p> <p>If you set the <code>codec</code> parameter to <code>anyCodec</code>, the Image Compression Manager chooses the first compressor it finds of the specified type.</p>

## Image Compression Manager

`spatialQuality`

Specifies the desired compressed image quality. See “Compression Quality Constants” beginning on page 3-61 for available values. You can change the value of this parameter for an active sequence by calling the `SetCSequenceQuality` function (described on page 3-124).

`temporalQuality`

Specifies the desired sequence temporal quality. This parameter governs the level of compression you desire with respect to information between successive frames in the sequence. Set this parameter to 0 to prevent the compressor from applying temporal compression to the sequence. See “Compression Quality Constants” beginning on page 3-61 for other available values.

You can change the value of this parameter for an active sequence by calling the `SetCSequenceQuality` function (described on page 3-124).

`keyFrameRate`

Specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. Use this parameter to control the frequency at which the compressor places key frames into the compressed sequence. The compressor determines the optimum placement for key frames based upon the amount of redundancy between adjacent images in the sequence. Consequently, the compressor may insert key frames more frequently than you have requested. However, the compressor never places fewer key frames than is indicated by the setting of the `keyFrameRate` parameter. The compressor ignores this parameter if you have not requested temporal compression (that is, you have set the `temporalQuality` parameter to 0). If you pass in 0 in this parameter, this indicates that there are no key frames in the sequence. If you pass in any other number, it specifies the number of non-key frames between key frames. Set this parameter to 1 to specify all key frames, to 2 to specify every other frame as a key frame, to 3 to specify every third frame as a key frame, and so forth.

Your application may change the key frame rate for an active sequence by calling the `SetCSequenceKeyFrameRate` function (described beginning on page 3-125). See “Defining Key Frame Rates” on page 3-51 for more information about key frames.

`clut`

Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source pixel map.

## Image Compression Manager

flags	<p>Contains flags providing further control information. See “Image Compression Manager Function Control Flags,” which begins on page 3-62, for information about <code>CodecFlags</code> fields. You must set either the value of the <code>codecFlagUpdatePrevious</code> flag or the <code>codecFlagUpdatePreviousComp</code> flag to 1 (be sure to set unused flags to 0). The following flags are available for this function:</p> <p><code>codecFlagUpdatePrevious</code> Controls whether the compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If you set this flag to 1, the compressor copies the current frame into the previous frame buffer at the end of frame compression.</p> <p><code>codecFlagUpdatePreviousComp</code> Controls whether the compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression and is similar to the <code>codecFlagUpdatePrevious</code> flag. As with the <code>codecFlagUpdatePrevious</code> flag, if you set this flag to 1, the compressor updates the previous frame buffer at the end of frame compression. However, this flag causes the Image Compression Manager to update the frame buffer using an image obtained by decompressing the results of the most recent compression operation, rather than the source image, which may give better results at the expense of taking more time.</p> <p><code>codecFlagWasCompressed</code> Indicates to the 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. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.</p>
desc	<p>Contains a handle that is to receive a formatted image description structure. The Image Compression Manager resizes this handle for the returned image description structure. Your application should store this image description with the compressed sequence. During the compression operation, the Image Compression Manager and the compressor component update the contents of this image description. Consequently, you should not store the image description until the sequence has been completely processed.</p>

## Image Compression Manager

## 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
codecConditionErr	-8972	Component cannot perform requested operation

**CompressSequenceFrame**

Your application calls the `CompressSequenceFrame` function to compress one of a sequence of frames.

```
pascal OSErr CompressSequenceFrame (ImageSequence seqID,
                                     PixMapHandle src, const Rect *srcRect,
                                     CodecFlags flags, Ptr data, long *dataSize,
                                     unsigned char *similarity,
                                     CompletionProcRecordPtr asyncCompletionProc);
```

seqID	Unique sequence identifier that was returned by the <code>CompressSequenceBegin</code> function (described in the previous section).
src	Contains a handle to a pixel map that contains the image to be compressed. The image must be stored in a pixel map structure.
srcRect	Contains a pointer to a rectangle defining the portion of the image to compress. The compressor applies this rectangle to the image stored in the buffer referred to by the <code>src</code> parameter.
flags	Specifies flags providing further control information. See “Image Compression Manager Function Control Flags,” which begins on page 3-62, for information about <code>CodecFlags</code> fields. You must set the value of either the <code>codecFlagUpdatePrevious</code> flag or the <code>codecFlagUpdatePreviousComp</code> flag to 1 (be sure to set unused flags to 0). The following flags are available for this function:

`codecFlagUpdatePrevious`

Controls whether the compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If you set this flag to 1, the compressor copies the current frame into the previous frame buffer at the end of frame compression.

Indicates to the 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. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.

## Image Compression Manager

`codecFlagUpdatePreviousComp`

Controls whether the compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression and is similar to the `codecFlagUpdatePrevious` flag. As with the `codecFlagUpdatePrevious` flag, if you set this flag to 1, the compressor updates the previous frame buffer at the end of frame compression. However, this flag causes the Image Compression Manager to update the frame buffer using an image obtained by decompressing the results of the most recent compression operation, rather than the source image.

`codecFlagForceKeyFrame`

Controls whether the compressor creates a key frame from the current image. This flag is only used with temporal compression. If you set this flag to 1, the compressor makes the current image a key frame. If you set this flag to 0, the compressor decides based on other criteria, such as the key frame rate, whether to create a key frame from the current image. If you don't want any key frames other than the ones that are forced, set the key frame rate for the sequence to 0.

`codecFlagLiveGrab`

Indicates to the compressor that speed is of the utmost importance, and that size and quality are of lesser importance. This flag is useful when you are grabbing sequences from a live source where each frame must be compressed quickly.

<code>data</code>	Points to a location to receive the compressed image data. It is your program's responsibility to make sure that this location can receive at least as much data as indicated by the <code>GetMaxCompressionSize</code> function (described on page 3-72). The Image Compression Manager places the actual size of the compressed image into the field referred to by the <code>dataSize</code> parameter. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's <code>StripAddress</code> routine before you use that pointer with this parameter. For details on <code>StripAddress</code> , see <i>Inside Macintosh: Memory</i> .
<code>dataSize</code>	Contains a pointer to a field that is to receive the size, in bytes, of the compressed image.
<code>similarity</code>	Contains a pointer to a field that is to receive a similarity value. The <code>CompressSequenceFrame</code> function returns a value that indicates the similarity of the current frame to the previous frame. A value of 0 indicates that the current frame is a key frame in the sequence. A value of



255 indicates that the current frame is identical to the previous frame. Values from 1 through 254 indicate relative similarity, ranging from very different (1) to very similar (254).

`asyncCompletionProc`

Points to a completion function structure. The compressor calls your completion function when an asynchronous compression operation is complete. You can cause the compression to be performed asynchronously by specifying a completion function if the compressor supports asynchronous compression. For more information about completion function structures, see “Completion Functions” on page 3-158.

If you specify asynchronous operation, you must not read the compressed data until the compressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous compression. If you set `asyncCompletionProc` to `-1`, the operation is performed asynchronously but the compressor does not call your completion function.

If the `asyncCompletionProc` parameter is not `nil`, the following conditions are in effect: the pixels in the source image must stay valid until the completion function is called with its `codecCompletionSource` flag, and the resulting compressed data is not valid until it is called with its `codecCompletionDest` flag set.

SPECIAL CONSIDERATIONS

You must call the `CompressSequenceBegin` function (described in the previous section) shortly before you use the `CompressSequenceFrame` function. `CompressSequenceFrame` uses the current graphics device and port set from your prior call to `CompressSequenceBegin`.

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	The Image Compression Manager could not find the specified compressor
<code>codecSpoolErr</code>	-8966	Error loading or unloading data

DecompressSequenceBegin

The Movie Toolbox handles the details of decompressing image sequences in QuickTime movies. If you need to decompress other sequences, your application calls this function to signal the beginning of the process of decompressing a sequence of frames. You must

## Image Compression Manager

call this function before calling the `DecompressSequenceFrame` function (described in the next section).

```
pascal OSErr DecompressSequenceBegin (ImageSequence *seqID,
                                      ImageDescriptionHandle desc,
                                      CGrafPtr port, GDHandle gdh,
                                      const Rect *srcRect,
                                      MatrixRecordPtr matrix, short mode,
                                      RgnHandle mask, CodecFlags flags,
                                      CodecQ accuracy,
                                      DecompressorComponent codec);
```

<code>seqID</code>	Contains a pointer to a field to receive the unique identifier for this sequence returned by the <code>CompressSequenceBegin</code> function (described on page 3-110). You must supply this identifier to all subsequent Image Compression Manager functions that relate to this sequence.
<code>desc</code>	Contains a handle to the image description structure that describes the compressed image.
<code>port</code>	Points to the graphics port for the destination image. If this parameter specifies a graphics world or points to the screen, set the <code>gdh</code> parameter to <code>nil</code> . If you set this parameter to <code>nil</code> , the Image Compression Manager uses the current port (in this case, you should also set the <code>gdh</code> parameter to <code>nil</code> ).
<code>gdh</code>	Contains a handle to the graphics device record for the destination image. If the port parameter specifies a graphics world or the screen, or if you set the port parameter to <code>nil</code> , set this parameter to <code>nil</code> .
<code>srcRect</code>	Contains a pointer 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 (0,0) and <code>((**desc).width, (**desc).height)</code> . If you want to decompress the entire source image, set this parameter to <code>nil</code> . If the <code>srcRect</code> parameter is <code>nil</code> , the rectangle is set to the rectangle structure of the image description structure. Your application can change the source rectangle for an active sequence by calling the <code>SetDSequenceSrcRect</code> function (described on page 3-135).
<code>matrix</code>	Points to a matrix structure that specifies how to transform the image during decompression. You can use the matrix structure to translate or scale the image during decompression. If you do not want to apply such effects, set the <code>matrix</code> parameter to <code>nil</code> . For more information about matrix operations, see the chapter “Movie Toolbox” in this book.  Your application can change the matrix for an active sequence by calling the <code>SetDSequenceMatrix</code> function (described on page 3-135).
<code>mode</code>	Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw’s <code>CopyBits</code> routine (described in <i>Inside Macintosh: Imaging</i> ).

## Image Compression Manager

	<p>Your application can change the transfer mode for an active sequence by calling the <code>SetDSequenceTransferMode</code> function (described on page 3-134).</p>
mask	<p>Contains a handle to a clipping region in the destination coordinate system. If specified, the decompressor applies this mask to the destination image. If you do not want to mask pixels in the destination, set this parameter to <code>nil</code>.</p> <p>Your application can change the clipping mask for an active sequence by calling the <code>SetDSequenceMask</code> function (described on page 3-136).</p>
flags	<p>Contains flags providing further control information. See “Image Compression Manager Function Control Flags,” which begins on page 3-62, for information about <code>CodecFlags</code> fields. The following flags are available for this function:</p> <p><code>codecFlagUseScreenBuffer</code></p> <p>Controls whether the decompressor allocates an offscreen buffer. The decompressor places the decompressed image into that buffer and then copies the image to the destination pixel map after completing the decompression operation. Using an offscreen buffer reduces the tearing effect that can result from writing directly to the screen during decompression. Set this flag to 1 to cause the decompressor to allocate and use an offscreen buffer. Set this flag to 0 to cause the decompressor to write to the destination pixel map.</p> <p>Your application can determine the screen buffer for an active sequence by calling the <code>GetDSequenceScreenBuffer</code> function (described on page 3-140).</p> <p><code>codecFlagUseImageBuffer</code></p> <p>Controls whether the decompressor allocates an offscreen buffer for the current image. The decompressor uses this buffer to store the compressed data from the current image so that subsequent images that are temporally compressed can be processed correctly. Set this flag to 1 to cause the decompressor to use an image buffer. Set this flag to 0 if your sequence is not temporally compressed and therefore does not require the use of an image buffer.</p> <p>Your application can determine the image buffer for an active sequence by calling the <code>GetDSequenceImageBuffer</code> function (described on page 3-140).</p>
accuracy	<p>Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See “Compression Quality Constants” beginning on page 3-61 for available values.</p> <p>Your application can change the accuracy parameter for an active sequence by calling the <code>SetDSequenceAccuracy</code> function (described on page 3-138).</p>

## Image Compression Manager

**codec** Contains a compressor identifier. Specify a particular decompressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

**anyCodec** Choose the first decompressor of the specified type

**bestSpeedCodec** Choose the fastest decompressor of the specified type

**bestFidelityCodec** Choose the most accurate decompressor of the specified type

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

If you set the `codec` parameter to `anycodec`, the Image Compression Manager chooses the first decompressor it finds of the specified type.

## DESCRIPTION

Use the `SetDSequenceDataProc` function (described on page 3-139) to assign a data-loading function to the sequence. Use the `SetDSequenceMatte` function (described on page 3-137) to assign a blend matte to the sequence.

## 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	The Image Compression Manager could not find the specified compressor
<code>codecScreenBufErr</code>	-8964	Could not allocate the screen buffer
<code>codecImageBufErr</code>	-8965	Could not allocate the image buffer
<code>codecConditionErr</code>	-8972	Component cannot perform requested operation

## DecompressSequenceFrame

Your application calls the `DecompressSequenceFrame` function to decompress one of a sequence of frames. You must have called the `DecompressSequenceBegin` function before calling this function. You specify the destination with the `port` parameter to the `DecompressSequenceBegin` function, described in the previous section.

```
pascal OSErr DecompressSequenceFrame (ImageSequence seqID,
                                       Ptr data, CodecFlags inFlags,
                                       CodecFlags *outFlags,
                                       CompletionProcRecordPtr asyncCompletionProc);
```

## Image Compression Manager

seqID	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
data	Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's <code>StripAddress</code> routine before you use that pointer with this parameter.
inFlags	<p>Contains flags providing further control information. See "Image Compression Manager Function Control Flags," which begins on page 3-62, for information about <code>CodecFlags</code> fields. The following flags are available for this function:</p> <p><code>codecFlagNoScreenUpdate</code> Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.</p> <p><code>codecFlagDontOffscreen</code> Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.</p> <p><code>codecFlagOnlyScreenUpdate</code> Controls whether the decompressor decompresses the current frame. If you set this flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does not decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.</p>
outFlags	<p>Contains a pointer to status flags. The decompressor updates these flags at the end of the decompression operation. See "Image Compression Manager Function Control Flags," which begins on page 3-62, for information about <code>CodecFlags</code> constants. The following flags may be set by this function:</p> <p><code>codecFlagUsedNewImageBuffer</code> Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence.</p>

## Image Compression Manager

`codecFlagUsedImageBuffer`

Indicates whether the decompressor used the offscreen image buffer. If the decompressor used the image buffer during the decompress operation, it sets this flag to 1. Otherwise, it sets this flag to 0.

`codecFlagDontUseNewImageBuffer`

Forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`

Updates the screen by **interlacing** even and odd scan lines to reduce **tearing** artifacts (if the decompressor supports this mode).

`asyncCompletionProc`

Points to a completion function structure. The compressor calls your completion function when an asynchronous decompression operation is complete. You can cause the decompression to be performed asynchronously by specifying a completion function. See “Completion Functions,” which begins on page 3-158, for more information about completion functions.

If you specify asynchronous operation, you must not read the decompressed image until the decompressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous decompression. If you set `asyncCompletionProc` to `-1`, the operation is performed asynchronously but the decompressor does not call your completion function.

## SPECIAL CONSIDERATIONS

Only if the `asyncCompletionProc` parameter of `CompressSequenceFrame` is not `nil` are the following conditions in effect: the compressed data must remain valid until the completion function is called with its `codecCompletionSource` flag, and the pixels in the destination image will not be valid until the completion function is called with its `codecCompletionDest` flag set.

## 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	The Image Compression Manager could not find the specified compressor
<code>codecSpoolErr</code>	-8966	Error loading or unloading data

## CDSequenceBusy

---

Your application may call the CDSequenceBusy function to check the status of an asynchronous compression or decompression operation.

```
pascal OSErr CDSequenceBusy (ImageSequence seqID);
```

**seqID**            Contains the unique sequence identifier that was returned by the DecompressSequenceBegin or CompressSequenceBegin function (described on page 3-117 and page 3-110, respectively).

### DESCRIPTION

If there is no asynchronous operation in progress, the CDSequenceBusy function returns a 0 result code. If there is an asynchronous operation in progress, the result code is 1. Negative result codes indicate an error.

### SPECIAL CONSIDERATIONS

If you call the CDSequenceEnd function (described in the next section), you don't need to call CDSequenceBusy to make sure you have completed an operation.

### RESULT CODES

paramErr	-50	Invalid parameter specified
codecUnimpErr	-8962	Feature not implemented by this compressor
Component Manager errors		

## CDSequenceEnd

---

Your application calls the CDSequenceEnd function to indicate the end of processing for an image sequence.

```
pascal OSErr CDSequenceEnd (ImageSequence seqID);
```

**seqID**            Contains the unique sequence identifier that was returned by the DecompressSequenceBegin or CompressSequenceBegin function (described on page 3-117 and page 3-110, respectively).

### SPECIAL CONSIDERATIONS

You must make this call to CDSequenceEnd to make sure that all resources associated with the sequence are freed.

## Image Compression Manager

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>noCodecErr</code>	-8961	The Image Compression Manager could not find the specified compressor

## SEE ALSO

See “Compressing Sequences,” which begins on page 3-35, and “Decompressing Sequences,” which begins on page 3-37, for more on how to use `CDSequenceEnd`. Also see “A Sample Program for Compressing and Decompressing a Sequence of Images,” which begins on page 3-39, for details on how to use `CDSequenceEnd`.

## Changing Sequence-Compression Parameters

---

This section describes the functions that allow your application to manipulate the parameters that control sequence compression and to get information about memory that the compressor has allocated. You can use these functions during the sequence-compression process. Your application establishes the default value for most of these parameters with the `CompressSequenceBegin` function (described on page 3-110). Some of these functions deal with parameter values that cannot be set when starting a sequence.

You can determine the location of the previous image buffer used by the Image Compression Manager by calling the `GetCSequencePrevBuffer` function.

You can set a number of compression parameters. Use the `SetCSequenceFlushProc` function to assign a data-unloading function to the operation. You can set the rate at which the Image Compression Manager inserts key frames into the compressed sequence by calling the `SetCSequenceKeyFrameRate` function. You can set the frame against which the compressor compares a frame when performing temporal compression by calling the `SetCSequencePrev` function. Finally, you can control the quality of the compressed image by calling the `SetCSequenceQuality` function.

## SetCSequenceQuality

---

The `SetCSequenceQuality` function allows you to adjust the spatial or temporal quality for the current sequence.

```
pascal OSErr SetCSequenceQuality (ImageSequence seqID,
                                   CodecQ spatialQuality,
                                   CodecQ temporalQuality);
```

`seqID`            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function.



## Image Compression Manager

`spatialQuality`

Specifies the desired compressed image quality. See “Compression Quality Constants” on page 3-61 for available values.

`temporalQuality`

Specifies the desired sequence temporal quality. This parameter governs the level of compression you desire with respect to information between successive frames in the sequence. Set this parameter to 0 to prevent the compressor from applying temporal compression to the sequence. See “Compression Quality Constants” beginning on page 3-61 for other available values.

## DESCRIPTION

The spatial quality parameter indicates the image quality you desire for each frame in the sequence, which governs the level of spatial compression that the compressor may apply to each frame. The temporal quality parameter indicates the sequence quality you desire, which in turn governs the amount of temporal compression that the compressor may apply to the sequence. The new quality parameters take effect with the next frame in the sequence.

You set the default spatial and temporal quality values for a sequence with the `spatialQuality` and `temporalQuality` parameters to the `CompressSequenceBegin` function. For details on `CompressSequenceBegin`, see page 3-110.

If you change the quality settings while processing an image sequence, you affect the maximum image size that you may receive during sequence compression. Consequently, you should call the `GetMaxCompressionSize` function (described on page 3-72) after you change the quality settings. If the maximum size has increased, you should reallocate your image buffers to accommodate the larger image size.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

**SetCSequenceKeyFrameRate**

The `SetCSequenceKeyFrameRate` function adjusts the key frame rate for the current sequence.

```
pascal OSErr SetCSequenceKeyFrameRate (ImageSequence seqID,
                                         long keyframerate);
```

`seqID`            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-110).

## Image Compression Manager

`keyframerate`

Specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. Use this parameter to control the frequency at which the compressor places key frames into the compressed sequence.

The compressor determines the optimum placement for key frames based upon the amount of redundancy between adjacent images in the sequence. Consequently, the compressor may insert key frames more frequently than you have requested. However, the compressor will never place fewer key frames than is indicated by the setting of the `keyFrameRate` parameter. The compressor ignores this parameter if you have not requested temporal compression (that is, you have set the `temporalQuality` parameter to the `CompressSequenceBegin` function to 0).

If you set this parameter to 0, the Image Compression Manager only places key frames in the compressed sequence when you call the `CompressSequenceFrame` function (described on page 3-115) and set the value of the `codecFlagForceKeyFrame` flag to 1 in the `flags` parameter. If you pass in any number other than 0, it specifies the number of non-key frames between key frames. Set this parameter to 1 to specify all key frames, to 2 to specify every other frame as a key frame, to 3 to specify every third frame as a key frame, and so forth.

## DESCRIPTION

The key frame rate for a sequence specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. The new key frame rate takes effect with the next image in the sequence. See “Defining Key Frame Rates” on page 3-51 for more information about key frames.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## SEE ALSO

You set the default key frame rate for a sequence with the `keyFrameRate` parameter to the `CompressSequenceBegin` function (described on page 3-110).

## GetCSequenceKeyFrameRate

---

The `GetCSequenceKeyFrameRate` function lets you determine the current key frame rate of a sequence.

```
pascal OSErr GetCSequenceKeyFrameRate (ImageSequence seqID,
                                         long *keyframerate);
```

**seqID**            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-110).

**keyframerate**    Contains a pointer to a long integer that specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed.

### SEE ALSO

You can set the key frame rate of a sequence with the `SetCSequenceKeyFrameRate` function, described in the previous section.

### RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## SetCSequenceFrameNumber

---

The `SetCSequenceFrameNumber` function informs the compressor in use for the specified sequence that frames are being compressed out of order.

```
pascal OSErr SetCSequenceFrameNumber (ImageSequence seqID,
                                         long frameNumber);
```

**seqID**            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-110).

**frameNumber**    Specifies the frame number of the frame that is being compressed out of sequence.

### DESCRIPTION

This information is only necessary for compressors that are sequence-sensitive.

## Image Compression Manager

## RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

**GetCSequenceFrameNumber**

---

The `GetCSequenceFrameNumber` function returns the current frame number of the specified sequence.

```
pascal OSErr GetCSequenceFrameNumber (ImageSequence seqID,
                                       long *frameNumber);
```

**seqID**            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-110).

**frameNumber**    Contains a pointer to the current frame number of the sequence identified by the `seqID` parameter.

## RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

**SetCSequencePrev**

---

The `SetCSequencePrev` function allows the application to set the pixel map and boundary rectangle used by the previous frame in temporal compression. This is useful if the application that is compressing has multiple buffers and wants to update the previous frame by switching buffer pointers instead of copying the data. Usually, the Image Compression Manager allocates the previous buffer for temporal compression. Under normal circumstances, the compressor component or the Image Compression Manager updates the contents of the buffer by copying each frame into the buffer after it is compressed.

This is a very specialized function—your application should not need to call it under most circumstances.

```
pascal OSErr SetCSequencePrev (ImageSequence seqID,
                               PixMapHandle prev,
                               const Rect *prevRect);
```

**seqID**            Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-110).

Image Compression Manager

prev	Contains a handle to the new previous image buffer. The compressor uses this buffer to store a previous image against which the current image (stored in the buffer referred to by the src parameter to the CompressSequenceBegin function) is compared when performing temporal compression. You must allocate this buffer using the same pixel depth and color table as the source image buffer that you specify with the src parameter when you call the CompressSequenceBegin function (described on page 3-110). The compressor manages the contents of this buffer based upon several considerations, such as the key frame rate and the degree of difference between compared images.
prevRect	Contains a pointer to a rectangle defining the portion of the previous image to use for temporal compression. The compressor uses this portion of the previous image as the basis of comparison with the current image. This rectangle must be the same size as the source rectangle you specify with the srcRect parameter to the CompressSequenceBegin function. To get the boundary of a source pixel map, set this parameter to nil.

DESCRIPTION

When you start compressing a sequence, you may assign a previous frame buffer and rectangle with the prev and prevRect parameters to the CompressSequenceBegin function, respectively. If you specified a nil value for the prev parameter, the compressor allocates an offscreen buffer for the previous frame. In either case you may use this function to assign a new previous frame buffer.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
Memory Manager errors		

SetCSequenceFlushProc

The SetCSequenceFlushProc function lets you assign a data-unloading function to a sequence.

```
pascal OSErr SetCSequenceFlushProc (ImageSequence seqID,
                                     FlushProcRecordPtr flushProc,
                                     long bufferSize);
```

seqID	Contains the unique sequence identifier that was returned by the CompressSequenceBegin function (described on page 3-110).
flushProc	Points to a data-unloading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that unloads some of the compressed data (see “Data-Unloading Functions” beginning on page 3-154 for more

## Image Compression Manager

information on the data-unloading structure). If you have not provided a data-unloading function, set this parameter to `nil`. In this case, the compressor writes the entire compressed image into the memory location specified by the `data` parameter to the `CompressSequenceFrame` function (described on page 3-115).

`bufferSize` Specifies the size of the buffer to be used by the data-unloading function specified by the `flushProc` parameter. If you have not specified a data-unloading function, set this parameter to 0.

## DESCRIPTION

Data-unloading functions allow compressors to work with images that cannot fit in memory. During the compression operation, the compressor calls the data-unloading function whenever it has accumulated a specified amount of compressed data. Your data-unloading function then writes the compressed data to some other device, freeing buffer space for more compressed data. The compressor starts using the data-unloading function with the next image in the sequence. See “Spooling Compressed Data” on page 3-48 for more information.

There is no parameter to the `CompressSequenceBegin` function (described on page 3-110) that allows you to assign a data-unloading function to a sequence.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## GetCSequencePrevBuffer

---

The `GetCSequencePrevBuffer` function determines the location of the previous image buffer allocated by the compressor.

```
pascal OSErr GetCSequencePrevBuffer (ImageSequence seqID,
                                     GWorldPtr *gworld);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>CompressSequenceBegin</code> function (described on page 3-110).
<code>gworld</code>	Contains a pointer to a field to receive a pointer to the structure of type <code>GWorld</code> that describes the graphics world for the image buffer. If the compressor has allocated an offscreen image buffer, the compressor returns an appropriate pointer to the graphics world (of type <code>GWorldPtr</code> ) in the field referred to by this parameter. If the compressor has not allocated a buffer, the function returns an error result code.  You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using.

DESCRIPTION

If you do not specify a previous image buffer with the `prev` parameter to the `CompressSequenceBegin` function, the compressor allocates an offscreen graphics world for you. Your program can obtain access to the pixel map in that graphics world by calling this function.

Note that the `GetCSequencePrevBuffer` function only returns information about buffers that were allocated by the compressor. You cannot use this function to determine the location of a buffer you have provided.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

Constraining Compressed Data

The Image Compression Manager provides two functions and a data structure that allow your application to communicate information to compressors that can constrain compressed data to a specific data rate. Compressors indicate that they can constrain the data rate by setting the following flag in their compressor information structure:

```
#define codecInfoDoesRateConstrain(1L<<23)
```

(For details, see “The Compressor Information Structure” beginning on page 3-56.)

The `DataRateParams` data type defines the data rate parameters structure.

```
typedef struct {
    long    dataRate;                /* bytes per second */
    long    dataOverrun;            /* number of bytes outside
                                   rate */
    long    frameDuration;          /* in milliseconds */
    long    keyFrameRate;           /* frequency of key frames */
    CodecQ  minSpatialQuality;      /* minimum spatial quality */
    CodecQ  minTemporalQuality;     /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;
```

Field descriptions

<code>dataRate</code>	Specifies the bytes per second to which the data rate must be constrained.
<code>dataOverrun</code>	Indicates the current number of bytes above or below the desired data rate. A value of 0 means that the data rate is being met exactly. If your application doesn't know the data overrun, it should set this field to 0.
<code>frameDuration</code>	Specifies the duration of the current frame in milliseconds.

## Image Compression Manager

<code>keyFrameRate</code>	Indicates the frequency of key frames. This frequency is normally identical to the key frame rate passed to the <code>CompressSequenceBegin</code> function (described on page 3-110).
<code>minSpatialQuality</code>	Specifies the minimum spatial quality the compressor should use to meet the requested data rate. See “Compression Quality Constants” beginning on page 3-61 for available values.
<code>minTemporalQuality</code>	Indicates the minimum temporal quality the compressor should use to meet the requested data rate. See “Compression Quality Constants” beginning on page 3-61 for available values.

The `SetCSequenceDataRateParams` function allows you to specify the parameters in this structure and the `GetCSequenceDataRateParams` function allows you to retrieve the parameters.

## SetCSequenceDataRateParams

---

The `SetCSequenceDataRateParams` function allows your application to set parameters in the data rate parameters structure, which communicates information to compressors that can constrain compressed data in a particular sequence to a specific data rate.

```
pascal OSErr SetCSequenceDataRateParams
                (ImageSequence seqID,
                 DataRateParamsPtr params);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>CompressSequenceBegin</code> function (described on page 3-110).
<code>params</code>	Points to the data rate parameters structure to be associated with the sequence identifier specified in the <code>seqID</code> parameter.

### DESCRIPTION

If your application is keeping track of data overrun, you should call the `SetCSequenceDataRateParams` function before each use of the `CompressSequenceFrame` function (described on page 3-115). If not, you only need to call `SetCSequenceDataRateParams` before the first use of `CompressSequenceFrame`, with the `dataOverrun` parameter of the data rate parameters structure set to 0. In this case, it is assumed that the frame duration is valid for all frames. Setting the `dataRate` field in the data rate parameters structure to 0 is the same as not performing data rate constraint.



**RESULT CODES**

noErr	0	No error
paramErr	-50	Invalid parameter specified

**GetCSequenceDataRateParams**

---

The `GetCSequenceDataRateParams` function obtains the data rate parameters previously set with the `SetCSequenceDataRateParams` function, which is described in the previous section.

```
pascal OSErr GetCSequenceDataRateParams
                (ImageSequence seqID,
                 DataRateParamsPtr params);
```

seqID	Contains the unique sequence identifier that was returned by the <code>CompressSequenceBegin</code> function (described on page 3-110).
params	Points to the data rate parameters structure associated with the sequence identifier specified in the <code>seqID</code> parameter.

**RESULT CODES**

noErr	0	No error
paramErr	-50	Invalid parameter specified

**Changing Sequence-Decompression Parameters**

---

This section discusses the functions that enable your application to manipulate the parameters that control sequence decompression and to get information about memory that the decompressor has allocated. Your application establishes the default value for most of these parameters with the `DecompressSequenceBegin` function (described on page 3-117). Some of these functions deal with parameter values that cannot be set when starting a sequence.

You can determine the buffers used by a decompressor component when it decompresses a sequence. Use the `GetDSequenceImageBuffer` function to determine the location of the image buffer. Use the `GetDSequenceScreenBuffer` function to determine the location of the screen buffer.

You can control a number of the parameters that affect a decompression operation (note that changing these parameters may temporarily affect performance). Use the `SetDSequenceAccuracy` function to control the accuracy of the decompression. Use the `SetDSequenceDataProc` function to assign a data-loading function to the operation. Use the `SetDSequenceMask` function to set the clipping region for the resulting image. You can establish a blend matte for the operation by calling the `SetDSequenceMatte` function. You can alter the spatial characteristics of the resulting image by calling the `SetDSequenceMatrix` function. Your application can establish the

## Image Compression Manager

size and location of the operation's source rectangle by calling the `SetDSequenceSrcRect` function. Finally, you can set the transfer mode used by the decompressor when it draws to the screen by calling the `SetDSequenceTransferMode` function.

## SetDSequenceTransferMode

---

The `SetDSequenceTransferMode` function sets the mode used when drawing the decompressed image.

```
pascal OSErr SetDSequenceTransferMode (ImageSequence seqID,
                                       short mode,
                                       const RGBColor *opColor);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
<code>mode</code>	Specifies the transfer mode used when drawing the decompressed image. The Image Compression Manager supports the same transfer modes supported by QuickDraw's <code>CopyBits</code> routine (described in <i>Inside Macintosh: Imaging</i> ).
<code>opColor</code>	Contains a pointer to the color for use in <code>addPin</code> , <code>subPin</code> , <code>blend</code> , and transparent operations. The Image Compression Manager passes this color to QuickDraw as appropriate. If <code>nil</code> , the <code>opcolor</code> is left unchanged.

### DESCRIPTION

The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine. The new mode takes effect with the next frame in the sequence. For any given sequence, the default `opcolor` is 50 percent gray and the default mode is `ditherCopy`.

### RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

### SEE ALSO

You set the default transfer mode for a sequence with the `mode` parameter to the `DecompressSequenceBegin` function.

## SetDSequenceSrcRect

---

The `SetDSequenceSrcRect` function defines the portion of the image to decompress.

```
pascal OSErr SetDSequenceSrcRect (ImageSequence seqID,
                                   const Rect *srcRect);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
<code>srcRect</code>	Contains a pointer 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 (0,0) and <code>((**desc).width, (**desc).height)</code> , where <code>desc</code> refers to the image description structure you supply to the <code>DecompressSequenceBegin</code> function. If the <code>srcRect</code> parameter is <code>nil</code> , the rectangle is set to the rectangle structure of the image description structure.

### DESCRIPTION

The decompressor acts on that portion of the compressed image that lies within this rectangle. The new source rectangle takes effect with the next frame in the sequence.

### RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

### SEE ALSO

You set the default source rectangle for a sequence with the `srcRect` parameter to the `DecompressSequenceBegin` function.

## SetDSequenceMatrix

---

The `SetDSequenceMatrix` function assigns a mapping matrix to the sequence.

```
pascal OSErr SetDSequenceMatrix (ImageSequence seqID,
                                   MatrixRecordPtr matrix);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
--------------------	---

## Image Compression Manager

**matrix** Points to a matrix structure that specifies how to transform the image during decompression. You can use the matrix structure to translate or scale the image during decompression. To set the matrix to identity, pass `nil` in this parameter. See the chapter “Movie Toolbox” in this book for more information about matrix operations.

**DESCRIPTION**

The decompressor uses the matrix to create special effects with the decompressed image, such as translating or scaling the image. The new matrix takes effect with the next frame in the sequence.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

**SEE ALSO**

You set the default matrix for a sequence with the `matrix` parameter to the `DecompressSequenceBegin` function.

**SetDSequenceMask**

---

The `SetDSequenceMask` function assigns a clipping region to the sequence.

```
pascal OSErr SetDSequenceMask (ImageSequence seqID,
                                RgnHandle mask);
```

<b>seqID</b>	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
<b>mask</b>	Contains a handle to a clipping region in the destination coordinate system. If specified, the decompressor applies this mask to the destination image. If you want to stop masking, set this parameter to <code>nil</code> .

**DESCRIPTION**

The decompressor draws only that portion of the decompressed image that lies within the specified clipping region. The new region takes effect with the next frame in the sequence. You should not dispose of this region until the Image Compression Manager is finished with the sequence, or until you set the mask either to `nil` or to a different region by calling the `SetDSequenceMask` function again.

RESULT CODES

noErr           0       No error  
paramErr       -50     Invalid parameter specified  
Memory Manager errors

SEE ALSO

You set the default clipping region for a sequence with the mask parameter to the DecompressSequenceBegin function.

SetDSequenceMatte

The SetDSequenceMatte function assigns a blend matte to the sequence.

```
pascal OSErr SetDSequenceMatte (ImageSequence seqID,  
                                PixMapHandle matte,  
                                const Rect *matteRect);
```

seqID	Contains the unique sequence identifier that was returned by the DecompressSequenceBegin function (described on page 3-117).
matte	Contains a handle to a pixel map that contains a blend matte. You can use the blend matte to cause the decompressed image to be blended into the destination pixel map. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. However, the matte must be in the coordinate system of the source image. If you want to turn off the blend matte, set this parameter to nil.
matteRect	Contains a pointer to the boundary rectangle for the matte. The decompressor uses only that portion of the matte that lies within the specified rectangle. This rectangle must be the same size as the source rectangle you specify with the SetDSequenceSrcRect function (described on page 3-135) or with the srcRect parameter to the DecompressSequenceBegin function. To specify the matte pixel map bounds, pass nil in this parameter.

DESCRIPTION

The decompressor uses the matte to blend the decompressed image into the destination pixel map. The new matte and matte boundary rectangle take effect with the next frame in the sequence. You should not dispose of the matte until the Image Compression Manager is finished with the sequence.

## Image Compression Manager

## RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
Memory Manager errors		

**SetDSequenceAccuracy**

---

The `SetDSequenceAccuracy` function adjusts the decompression accuracy for the current sequence.

```
pascal OSErr SetDSequenceAccuracy (ImageSequence seqID,
                                   CodecQ accuracy);
```

seqID	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
accuracy	Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See “Compression Quality Constants” beginning on page 3-61, for available values.

## DESCRIPTION

The `accuracy` parameter governs how precisely the decompressor decompresses the image data. Some decompressors may choose to ignore some image data to improve decompression speed. A new `accuracy` parameter takes effect with the next frame in the sequence.

## RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

## SEE ALSO

You set the default accuracy value for a sequence with the `accuracy` parameter to the `DecompressSequenceBegin` function.

## SetDSequenceDataProc

---

The `SetDSequenceDataProc` function lets you assign a data-loading function to the sequence.

```
pascal OSErr SetDSequenceDataProc (ImageSequence seqID,
                                   DataProcRecordPtr dataProc,
                                   long bufferSize);
```

<code>seqID</code>	Contains the unique sequence identifier that was returned by the <code>DecompressSequenceBegin</code> function (described on page 3-117).
<code>dataProc</code>	Points to a data-loading function structure. If the data stream is not all in memory when your program calls <code>DecompressSequenceFrame</code> , the decompressor calls a function you provide that loads more compressed data (see “Data-Loading Functions” beginning on page 3-153 for more information about data-loading functions). If you have not provided a data-loading function, or if you want the decompressor to stop using your data-loading function, set this parameter to <code>nil</code> . In this case, the entire image must be in memory at the location specified by the <code>data</code> parameter to the <code>DecompressSequenceFrame</code> function (see page 3-120).
<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 you have not specified a data-loading function, set this parameter to 0.

### DESCRIPTION

Data-loading functions allow decompressors to work with images that cannot fit in memory. During the decompression operation the decompressor calls the data-loading function whenever it has exhausted its supply of compressed data. Your data-loading function then fills the available buffer space with more compressed data. The decompressor starts using the data-loading function with the next image in the sequence. See “Spooling Compressed Data,” which begins on page 3-48, for more information about data-loading functions.

There is no parameter to the `DecompressSequenceBegin` function that allows you to assign a data-loading function to a sequence.

### RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## GetDSequenceImageBuffer

---

The `GetDSequenceImageBuffer` function helps you determine the location of the offscreen image buffer allocated by the decompressor.

```
pascal OSErr GetDSequenceImageBuffer (ImageSequence seqID,
                                       GWorldPtr *gworld);
```

**seqID**            Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-117).

**gworld**           Contains a pointer to a field to receive a pointer to the structure of type `GWorld` describing the graphics world for the image buffer. If the decompressor has allocated an offscreen image buffer, the decompressor returns an appropriate `GWorldPtr` in the field referred to by this parameter. If the decompressor has not allocated a buffer, the function returns an error result code.

You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using. It is disposed of for you when the `CDSequenceEnd` function is called. For details on `CDSequenceEnd`, see page 3-123.

### DESCRIPTION

The decompressor uses this buffer when decompressing a sequence that was temporally compressed. You cause the decompressor to use an image buffer by setting the `codecFlagUseImageBuffer` flag to 1 in the `flags` parameter to the `DecompressSequenceBegin` function.

### RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## GetDSequenceScreenBuffer

---

The `GetDSequenceScreenBuffer` function enables you to determine the location of the offscreen buffer allocated by the decompressor.

```
pascal OSErr GetDSequenceScreenBuffer (ImageSequence seqID,
                                       GWorldPtr *gworld);
```

**seqID**            Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-117).



## Image Compression Manager

**gworld** Contains a pointer to a field to receive a pointer to the graphics world structure (defined by the `GWorld` data type) describing the graphics world for the screen buffer. If the decompressor has allocated an offscreen buffer, the decompressor returns an appropriate `GWorldPtr` in the field referred to by this parameter. If the decompressor has not allocated a buffer, the function returns an error result code.

You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using. It is disposed of for you when the `CDSequenceEnd` function is called. For details on `CDSequenceEnd`, see page 3-123.

## DESCRIPTION

The decompressor uses this buffer for decompressed images. During decompression the decompressor writes the decompressed image to an offscreen buffer and then copies the results to the screen. This reduces the tearing effect that can result from decompressing directly to the screen. You cause the decompressor to use a screen buffer by setting the `codecFlagUseScreenBuffer` flag to 1 in the `flags` parameter to the `DecompressSequenceBegin` function.

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

## Working With the StdPix Function

To allow applications to have access to compressed image data as it is displayed, a new graphics function has been added to the `grafProcs` field of the color graphics port structure (defined by the `CGrafPort` data type). See *Inside Macintosh: Imaging* for more information about the color graphics port structure.

The `StdPix` function extends the current `grafProcs` field to support compressed data, mattes, and matrices. The new function supports pixel maps and allows you to intercept image data in compressed form before it is decompressed and displayed. For example, you can use the `StdPix` function to collect compressed image data that is to be processed and printed. In addition, your application can call the `StdPix` function directly.

The replaced `grafProcs` field is referred to in the original QuickDraw documentation as the `newProc1` field. The standard handler is called `StdPix`, and you obtain its address by calling QuickDraw's `SetStdCProcs` routine. Alternatively, your application can call the `StdPix` function directly, using the interface described here. Your application can intercept the low-level `grafProcs` drawing routines just as it would any of the other routines, except that you must call `SetStdCProcs` to gain access to the standard `grafProcs` handler.

## Image Compression Manager

**Note**

QuickDraw's `CopyDeepMask` function uses the `StdPix` function if QuickTime is present. ♦

See *Inside Macintosh: Imaging* for more information about the QuickDraw low-level drawing routines, the `SetStdCProcs` routine, the `QDProcs` structure, and the `CopyDeepMask` routine.

To work with the control information associated with a compressed image, you can use the `SetCompressedPixMapInfo` and `GetCompressedPixMapInfo` functions (described on page 3-143 and page 3-145, respectively).

**StdPix**

---

The Image Compression Manager lets you invoke QuickDraw's `StdPix` function as follows:

```
pascal void StdPix (PixMapPtr src, const Rect *srcRect,
                   MatrixRecordPtr matrix, short mode,
                   RgnHandle mask, PixMapPtr matte,
                   Rect *matteRect, short flags);
```

<code>src</code>	Contains a pointer to a pixel map containing the image to draw. Use the <code>GetCompressedPixMapInfo</code> function (described on page 3-145) to retrieve information about this pixel map.
<code>srcRect</code>	Points to a rectangle defining the portion of the image to display. This rectangle must lie within the boundary rectangle of the compressed image or within the source image. If this parameter is set to <code>nil</code> , the entire image is displayed.
<code>matrix</code>	Contains a pointer to a matrix structure that specifies the mapping of the source rectangle to the destination. It is a fixed-point, 3-by-3 matrix. This roughly corresponds to the <code>dstRect</code> parameter to QuickDraw's <code>StdBits</code> routine. See the chapter "Movie Toolbox" in this book for more information about matrix operations.
<code>mode</code>	Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw's <code>CopyBits</code> routine.  Note that this parameter also controls the accuracy of any decompression operation that may be required to display the image. If bit 7 (0x80) of the mode parameter is set to 1, the <code>StdPix</code> function sets the decompression accuracy to <code>codecNormalQuality</code> . If this bit is set to 0, the function sets the accuracy to <code>codecHighQuality</code> .
<code>mask</code>	Contains a handle to a clipping region in the destination coordinate system. If specified, the compressor applies this mask to the destination image. If there is no mask, this parameter is set to <code>nil</code> .

## Image Compression Manager

<code>matte</code>	<p>Points to a pixel map that contains a blend matte. The blend matte causes the decompressed image to be blended into the destination pixel map. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. However, the matte must be in the coordinate system of the source image. If there is no matte, this parameter is set to <code>nil</code>.</p> <p>The matte may be compressed. Use the <code>GetCompressedPixMapInfo</code> function (described on page 3-145) to determine if the matte pixel map contains compressed data.</p>
<code>matteRect</code>	Contains a pointer to a rectangle defining a portion of the blend matte to apply. This parameter is set to <code>nil</code> if there is no matte or if the entire matte is to be used.
<code>flags</code>	<p>Contains control flags. The following flags are available:</p> <p><code>callOldBits</code></p> <p>If this flag is set, then the <code>StdPix</code> function calls QuickDraw's <code>bitsProc</code> routine with the decompressed image data. A pointer to this routine is located in the <code>bitsProc</code> field of the <code>CQDProcs</code> record. If the <code>bitsProc</code> routine is not customized, then it is not called unless the <code>callStdBits</code> flag is also set. See the description of the <code>CQDProcs</code> record in <i>Inside Macintosh: Imaging</i> for more on the <code>bitsProc</code> routine.</p> <p><code>callStdBits</code></p> <p>If this flag is set, the <code>callOldBits</code> flag is set, and the <code>CQDProcs</code> record's <code>bitsProc</code> field is set to the <code>StdBits</code> routine, then the <code>StdBits</code> routine is called with the decompressed image data.</p> <p><code>noDefaultOpcodes</code></p> <p>If this flag is set and a picture is open for writing, the default picture opcodes (for displaying a warning when QuickTime is not installed) are not added to the output picture. This can be useful when storing multiple <code>StdPix</code> opcodes in a single picture.</p>

## SetCompressedPixMapInfo

The `SetCompressedPixMapInfo` function allows your application to store information about a compressed image for the `StdPix` function (described in the previous section).

```
pascal OSErr SetCompressedPixMapInfo (PixMapPtr pix,
                                       ImageDescriptionHandle desc,
                                       Ptr data, long bufferSize,
                                       DataProcRecordPtr dataProc,
                                       ProgressProcRecordPtr progressProc);
```

## Image Compression Manager

<code>pix</code>	Points to a structure that holds encoded compressed image data.
<code>desc</code>	Contains a handle to the image description structure that defines the compressed image.
<code>data</code>	Points to the buffer for the compressed image data. If the entire compressed image cannot be stored at this location, you may assign a data-loading function (see the discussion of the <code>dataProc</code> parameter to this function). This pointer must contain a 32-bit clean address.
<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 there is no data-loading function defined for this operation, set this parameter to 0.
<code>dataProc</code>	Points to a data-loading function structure. If there is not enough memory to store the compressed image, the decompressor calls a function you provide that loads more compressed data (see “Data-Loading Functions” beginning on page 3-153 for more information about data-loading functions). If you do not want to assign a data-loading function, set this parameter to <code>nil</code> .
<code>progressProc</code>	Points to a progress function structure. During the decompression operation, the decompressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If you do not want to assign a progress function, set this parameter to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function.

## DESCRIPTION

The `SetCompressedPixmapInfo` function stores information in a structure that is identical to a `Pixmap` structure, but the structure represents the compressed data, not the actual pixel map. You can use the `SetCompressedPixmapInfo` if you are working with the `StdPix` function (described on page 3-142).

## RESULT CODES

<code>paramErr</code>	<code>-50</code>	Invalid parameter specified
-----------------------	------------------	-----------------------------

## SEE ALSO

You can retrieve information about a compressed pixel map by calling the `GetCompressedPixmapInfo` function, which is described in the next section.

## GetCompressedPixMapInfo

---

The `GetCompressedPixMapInfo` function allows your application to retrieve information about a compressed image.

```
pascal OSErr GetCompressedPixMapInfo (PixMapPtr pix,
                                       ImageDescriptionHandle *desc,
                                       Ptr *data, long *bufferSize,
                                       DataProcRecord *dataProc,
                                       ProgressProcRecord *progressProc);
```

<code>pix</code>	Points to a structure that holds encoded compressed image data.
<code>desc</code>	Contains a pointer to a field that is to receive a handle to the image description structure that defines the compressed image. If you are not interested in this information, you may specify <code>nil</code> in this parameter.
<code>data</code>	Contains a pointer to a field that is to receive a pointer to the compressed image data. If the entire compressed image cannot be stored at this location, you can define a data-loading function for this operation (see the discussion of the <code>dataProc</code> parameter to this function). If you are not interested in this information, you may specify <code>nil</code> in this parameter.
<code>bufferSize</code>	Contains a pointer to a field that is to receive the size of the buffer to be used by the data-loading function specified by the <code>dataProc</code> parameter. If there is no data-loading function defined for this operation, this parameter is ignored. If you are not interested in this information, you may specify <code>nil</code> in this parameter.
<code>dataProc</code>	Contains a pointer to a data-loading function structure. If there is not enough memory to store the compressed image, the decompressor calls a function you provide that loads more compressed data (see “Data-Loading Functions” beginning on page 3-153 for more information about data-loading functions). If there is no data-loading function for this image, the function sets the <code>dataProc</code> field in the function structure to <code>nil</code> . If you are not interested in this information, you may specify <code>nil</code> in this parameter.
<code>progressProc</code>	Contains a pointer to a progress function structure. During a decompression operation, the decompressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-156 for more information about progress functions). If there is no progress function for this image, the function sets the <code>progressProc</code> field in the function structure to <code>nil</code> . If you pass a value of <code>-1</code> , you obtain a standard progress function. If you are not interested in this information, you may specify <code>nil</code> in this parameter.

## Image Compression Manager

## DESCRIPTION

The data in the compressed image has been encoded in a `PixelFormat` structure with the `SetCompressPixelFormatInfo` function. This is the kind of pixel map that may be passed into the `StdPix` function. If you pass a normal, non-encoded pixel map, `GetCompressedPixelFormatInfo` returns a `paramErr` result code. You use the `GetCompressedPixelFormatInfo` function if you are intercepting calls to the `StdPix` function.

## SPECIAL CONSIDERATIONS

The pixel map structure filled in by the `GetCompressedPixelFormatInfo` function should not be used by any other Macintosh functions. It is only to be used by the `StdPix` function.

## RESULT CODES

<code>paramErr</code>	-50	Invalid parameter specified
-----------------------	-----	-----------------------------

## SEE ALSO

You can set information about a compressed pixel map by calling the `SetCompressedPixelFormatInfo` function, which is described in the previous section.

## Aligning Windows

This section describes the functions that allow your application to position and drag windows to optimal screen positions based on the depth of the screen. These functions are useful for movie playback performance considerations that depend on where you draw on the screen.

The Image Compression Manager places the windows at an optimal position on the screen by aligning rectangles horizontally on 1-bit and 2-bit screens to multiples of 16 pixels, aligning 4-bit screens to multiples of 8, aligning 8-bit screens to multiples of 4, and aligning 16-bit screens to multiples of 2. (Alignment on 32-bit screens is to multiples of 4 pixels and only occurs on Macintosh computers of class 68040 or greater.) When the alignment rectangle crosses more than one screen, the Image Compression Manager uses the alignment of the strictest screen.

Decompression to non-optimally aligned destinations can reduce performance by as much as 50 percent, so you should use these functions whenever possible.

The alignment behavior provided by these functions is adequate in the vast majority of situations. However, if you need customized alignment behavior (for example, justification specifications geared to particular video hardware), you can use the application-defined function described in “Alignment Functions” on page 3-159 to override the standard alignment. See the chapter “Sequence Grabber Components” in *Inside Macintosh: QuickTime Components* for more information on application-defined

alignment functions and video hardware. All the alignment functions provide a parameter in which you can specify a function with customized alignment behavior.

The `AlignWindow` function enables you to transport a specified window to the nearest optimal alignment position. The `DragAlignedWindow` function drags the specified window along an optimal alignment grid. The `DragAlignedGrayRgn` function drags a specified gray region along an optimal alignment grid. The `AlignScreenRect` function aligns a specified rectangle to the strictest screen that the rectangle intersects.

## AlignWindow

---

The `AlignWindow` function moves a specified window to the nearest optimal alignment position.

```
pascal void AlignWindow (WindowPtr wp, Boolean front,
                        const Rect *alignmentRect,
                        AlignmentProcRecordPtr alignmentProc);
```

`wp` Points to the window to be aligned.

`front` Specifies the frontmost window. If the `front` parameter is `true` and the window specified in the `wp` parameter isn't the active window, `AlignWindow` makes it the active window by calling the Window Manager's `SelectWindow` routine.

`alignmentRect` Contains a pointer to a rectangle in window coordinates that allows you to align the window to a rectangle within the window. Set this parameter to `nil` to align using the bounds of the window.

`alignmentProc` Points to a function that allows you to provide your own alignment behavior. Set this parameter to `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc(Rect *rp, long refcon);
```

See "Alignment Functions" on page 3-159 for details.

### SEE ALSO

The `AlignWindow` function is similar to the Window Manager's `MoveWindow` routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details.

## DragAlignedWindow

---

The DragAlignedWindow function drags the specified window along an optimal alignment grid.

```
pascal void DragAlignedWindow (WindowPtr wp, Point startPt,
                               Rect *boundsRect,
                               Rect *alignmentRect,
                               AlignmentProcRecordPtr alignmentProc);
```

- |               |   |
|---------------|---|
| wp            | Contains a window pointer to the window to be dragged.  |
| startPt       | Specifies a point that is equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event structure). DragAlignedWindow pulls a gray outline of the window around the screen, following the movements of the mouse until the button is released.  |
| boundsRect    | Points to the boundary rectangle in global coordinates. If the mouse button is released when the mouse position is outside the limits of the boundary rectangle, DragAlignedWindow returns without moving the window or making it the active window. For a document window, the boundary rectangle typically is four pixels in from the menu bar and from the other edges of the screen, to ensure that there won't be less than a four-pixel-square area of the title bar visible on the screen. |
| alignmentRect | Points to a rectangle in window coordinates that allows you to align the window to a rectangle within the window. Set this parameter to nil to align using the bounds of the window.  |
| alignmentProc | Allows you to provide your own alignment behavior. Set this parameter to nil to use the standard alignment behavior. Your alignment function must be in the following form:<br><br><pre>pascal void MyAlignmentProc (Rect *rp, long refcon);</pre> <p>See "Alignment Functions" on page 3-159 for details.</p>  |

### SEE ALSO

The DragAlignedWindow is similar to the Window Manager's DragWindow routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details on DragWindow.



## DragAlignedGrayRgn

---

The `DragAlignedGrayRgn` function drags the specified gray region along an optimal alignment grid.

```
pascal long DragAlignedGrayRgn (RgnHandle theRgn, Point startPt,
                                Rect *boundsRect, Rect *slopRect,
                                short axis, ProcPtr actionProc,
                                Rect *alignmentRect,
                                AlignmentProcRecordPtr alignmentProc);
```

<code>theRgn</code>	Contains a region handle to the specified region for this operation. When the user holds down the mouse button, <code>DragAlignedGrayRgn</code> pulls a gray outline of the region around following the movement of the mouse until the mouse button is released.
<code>startPt</code>	Specifies the point where the mouse button was originally pressed in the local coordinates of the current graphics port.
<code>boundsRect</code>	Contains a pointer to the boundary rectangle of the current graphics port. The offset point follows the mouse location except that <code>DragAlignedGrayRgn</code> never moves the offset point outside this rectangle. This limits the travel of the region's outline, not the movements of the mouse.
<code>slopRect</code>	Contains a pointer to the <code>slop</code> rectangle that completely encloses the boundary rectangle so that the user is allowed some flexibility in moving the mouse.
<code>axis</code>	Allows you to constrain the region's motion to only one axis. Set this parameter to 0 to specify no constraint. To indicate constraint along a horizontal axis, set this parameter to 1. To indicate constraint along a vertical axis, set this parameter to 2. See <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for details on the constants for the axis parameter of the Window Manager's <code>DragGrayRgn</code> routine.
<code>actionProc</code>	Points to a function that defines some action to be performed repeatedly as long as the user holds down the mouse button. The function should have no parameters. If the <code>actionProc</code> parameter is <code>nil</code> , <code>DragAlignedGrayRgn</code> simply retains control until the mouse button is released.
<code>alignmentRect</code>	Contains a pointer to a rectangle within the bounds of the region specified in the parameter <code>theRgn</code> . Pass <code>nil</code> to align using the bounds of the parameter <code>theRgn</code> .

## Image Compression Manager

`alignmentProc`

Points to your own alignment behavior function. Pass `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refcon);
```

See “Alignment Functions” on page 3-159 for details.

## DESCRIPTION

The `DragAlignedGrayRgn` function is not normally made directly. The `DragAlignedWindow` function (described on page 3-148) calls this function.

## SEE ALSO

The `DragAlignedGrayRgn` function is nearly identical to the Window Manager’s `DragGrayRgn` routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details on `DragGrayRgn`.

## AlignScreenRect

---

The `AlignScreenRect` function aligns a specified rectangle to the strictest screen that the rectangle intersects.

```
pascal void AlignScreenRect (Rect *rp,
                             AlignmentProcRecordPtr alignmentProc);
```

`rp`                Contains a pointer to a rectangle defined in global screen coordinates.

`alignmentProc`

Points to your own alignment behavior function. Set this parameter to `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refCon);
```

See “Alignment Functions” on page 3-159 for details.

## DESCRIPTION

Normally, the `AlignScreenRect` function is not called directly.

## Working With Graphics Devices and Graphics Worlds

---

This section describes two Image Compression Manager functions that enable you to select graphics devices and create graphics worlds. You can use the `GetBestDeviceRect` function to select the best available graphics device. The `NewImageGWorld` function allows you to create a graphics world based on the width, height, depth, and color table of a specified image description structure.

### GetBestDeviceRect

---

The `GetBestDeviceRect` function selects the deepest of all available graphics devices, while treating 16-bit and 32-bit screens as having equal depth.

```
pascal OSErr GetBestDeviceRect (GDHandle *gdh, Rect *rp);
```

<code>gdh</code>	Contains a pointer to the handle of the rectangle for the chosen device. If you do not need the information in this parameter returned, specify <code>nil</code> .
<code>rp</code>	Contains a pointer to the rectangle that is adjusted for the height of the menu bar if the device is the main device. If you do not need the information in this parameter returned, specify <code>nil</code> .

#### DESCRIPTION

If multiple 16-bit and 32-bit monitors are available, the `GetBestDeviceRect` function selects the 16-bit or 32-bit device upon which the cursor has currently been detected. If a cursor is not on one of the devices in question, the first of those in the list is chosen.

Note that the `GetBestDeviceRect` function does not center a rectangle on a device. Rather, it returns the rectangle for the best device.

### NewImageGWorld

---

The `NewImageGWorld` function creates a graphics world from the width, height, depth, and color table of a specified image-description structure.

```
pascal QDErr NewImageGWorld (GWorldPtr *gworld,
                             ImageDescription **idh,
                             GWorldFlags flags);
```

<code>gworld</code>	Contains a pointer to a graphic world created using the width, height, depth, and color table specified in the image description structure pointed to in the <code>idh</code> parameter.
---------------------	--

## Image Compression Manager

<code>idh</code>	Contains a handle to an image description structure with information for the graphics world pointed to by the <code>gworld</code> parameter.
<code>flags</code>	Contains graphics world flags. These flags are passed directly through to the <code>NewGWorld</code> function. (For details on <code>NewGWorld</code> , see <i>Inside Macintosh: Devices</i> .)

## DESCRIPTION

The `NewImageGWorld` function selects the appropriate color table using the `depth` field or custom color table in the image description structure. It creates a 32-bit-deep graphics world if the depth specified in the image description structure is 24.

## SPECIAL CONSIDERATIONS

You are responsible for disposing of the graphics world with the `DisposeGWorld` routine. (For more on `DisposeGWorld`, see *Inside Macintosh: Devices*.)

## RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>cDepthErr</code>	-157	Invalid pixel resolution

## Application-Defined Functions

---

This section describes four callback functions that you may provide to compressor components and an application-defined function that specifies alignment behavior.

The Image Compression Manager defines four callback functions that applications may provide to compressors or decompressors. These callbacks 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 compressors and decompressors to report that asynchronous operations have completed.
- Progress functions provide a mechanism for compressors and decompressors to report their progress toward completing an operation.

This section describes the interfaces presented when compressors invoke your callback functions. These application-defined functions may be called by compressor components during a compression or decompression operation.

You identify a callback function to an Image Compression Manager function by specifying a pointer to a callback function structure. These structures contain two fields: a pointer to the callback function and a reference constant value. There is one callback function structure for each type of callback function. See the individual function descriptions in the sections that follow for descriptions of the structures.

## Data-Loading Functions

---

Compressors use the data-loading and data-unloading functions when working with images that do not fit into memory. The data-loading function supplies compressed data during a decompression operation.

The `DataProcPtr` data type defines a pointer to a data-loading function. You assign a data-loading function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate decompress function.

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

The data-loading function structure contains the following fields:

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

### Field descriptions

<code>dataProc</code>	Contains a pointer to your data-loading function.
<code>dataRefCon</code>	Contains a reference constant for use by your data-loading function.

### DESCRIPTION

If your data-loading function returns a nonzero result code, the Image Compression Manager terminates the current operation.

## MyDataLoadingProc

---

Your data-loading function should have the following form:

```
pascal OSErr MyDataLoadingProc (Ptr *dataP, long bytesNeeded,
                                long refcon);
```

<code>dataP</code>	Contains a pointer to the address of the data buffer. The decompressor uses this parameter to indicate where your data-loading function should return the compressed data. You establish this data buffer when you start the decompression operation. For example, the <code>data</code> parameter to the <code>FDecompressImage</code> function (described on page 3-83) defines the location of the data buffer for that operation. Upon return from your data-loading function, this pointer should refer to the beginning of the compressed data that you loaded.
--------------------	---

## Image Compression Manager

The decompressor may also use this parameter to indicate that it wants to reset the mark within the compressed data stream. If the `dataP` parameter is set to `nil`, the `bytesNeeded` parameter contains the new mark position, relative to the current position of the data stream. If your data-loading function does not support this operation, return a nonzero result code.

**bytesNeeded**

Specifies the number of bytes requested or the new mark offset. If the decompressor has requested additional compressed data (that is, the value of the `dataP` parameter is not `nil`), then this parameter specifies how many bytes to return. This value never exceeds the size of the original data buffer. Your data-loading function should read the data from the current mark in the input data stream.

If the decompressor has requested to set a new mark position in the data stream (that is, the value of the `dataP` parameter is `nil`), then this parameter specifies the new mark position relative to the current position of the data stream.

**refcon**

Contains a reference constant value for use by your data-loading function. Your application specifies the value of this reference constant in the data-loading function structure you pass to the Image Compression Manager.

**SPECIAL CONSIDERATIONS**

The pointer in the `dataP` parameter must contain a 32-bit clean address within the data buffer. If you have dereferenced a handle, you should call the Memory Manager's `StripAddress` routine before passing it to the `MyDataLoadingProc` function.

**RESULT CODES**

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

**Data-Unloading Functions**

Compressors use the data-loading and data-unloading functions when working with images that do not fit into the computer's memory. The data-unloading function writes compressed data to a storage device during a compression operation.

The `FlushProcPtr` data type defines a pointer to a data-unloading function.

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

You assign a data-unloading function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate compression function.

## Image Compression Manager

The data-unloading function structure contains the following fields:

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

**Field descriptions**

<code>flushProc</code>	Contains a pointer to your data-unloading function.
<code>flushRefCon</code>	Contains a reference constant for use by your data-unloading function.

## MyDataUnloadingProc

---

Your data-unloading function should have the following form:

```
pascal OSErr MyDataUnloadingProc (Ptr data, long bytesAdded,
                                   long refcon);
```

<code>data</code>	<p>Points to the data buffer. The compressor uses this parameter to indicate where your data-unloading function can find the compressed data. You establish this data buffer when you start the compression operation. For example, the <code>data</code> parameter to the <code>FCompressImage</code> function (described on page 3-79) defines the location of the data buffer for that operation. This pointer contains a 32-bit clean address. Your data-unloading function should make no other assumptions about the value of this address.</p> <p>The compressor may also use this parameter to indicate that it wants to reset the mark within the compressed data stream. If the <code>data</code> parameter is set to <code>nil</code>, the <code>bytesNeeded</code> parameter contains the new mark position, relative to the current position of the output data stream. If your data-unloading function does not support this operation, return a nonzero result code.</p>
<code>bytesAdded</code>	<p>Specifies the number of bytes to write or the new mark offset. If the compressor wants to write out some compressed data (that is, the value of <code>data</code> is not <code>nil</code>), then this parameter specifies how many bytes to write. This value never exceeds the size of the original data buffer. Your data-unloading function should write that data at the current mark in the output data stream.</p> <p>If the compressor has requested to set a new mark position in the output data stream (that is, the value of <code>data</code> is <code>nil</code>), then this parameter specifies the new mark position relative to the current position of the data stream.</p>

## Image Compression Manager

**refcon**      Contains a reference constant value for use by your data-unloading function. Your application specifies the value of this reference constant in the data-unloading function structure you pass to the Image Compression Manager.

**RESULT CODES**

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

**Progress Functions**

---

Compressors and decompressors call progress functions to report on their progress in the current operation. When a component calls your progress function, it supplies you with a number that indicates the completion percentage. This fixed-point value may range from 0.0 through 1.0. Your program can cause the component to terminate the current operation by returning a result code of `codecAbortErr`.

The Image Compression Manager calls your progress function only during long operations, and it does not call your function more than 30 times per second.

The `ProgressProcPtr` data type defines a pointer to a progress function. You assign a progress function to an image or a sequence by passing a pointer to a structure that identifies the progress function to the appropriate function.

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

The progress function structure contains the following fields:

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

**Field descriptions**

**progressProc**      Contains a pointer to your progress function.

**progressRefCon**      Contains a reference constant for use by your progress function.



## MyProgressProc

---

Your progress function should have the following form:

```
pascal OSErr MyProgressProc (short message, Fixed completeness,
                             long refcon);
```

message	Indicates why the Image Compression Manager called your function. The following values are valid:
codecProgressOpen	Indicates the start of a long operation. This is always the first message sent to your function. Your function can use this message to trigger the display of your progress window.
codecProgressUpdatePercent	Passes completion information to your function. The Image Compression Manager repeatedly sends this message to your function. The completeness parameter indicates the relative completion of the operation. You can use this value to update your progress window.
codecProgressClose	Indicates the end of a long operation. This is always the last message sent to your function. Your function can use this message as an indication to remove its progress window.
completeness	Contains a fixed-point value indicating how far the operation has progressed. Its value is always between 0.0 and 1.0. This parameter is valid only when the message field is set to codecProgressUpdatePercent.
refcon	Contains a reference constant value for use by your progress function. Your application specifies the value of this reference constant in the progress function structure you pass to the Image Compression Manager.

### DESCRIPTION

The following functions have parameters that allow you to provide application-defined progress functions: `FCompressImage`, `FDecompressImage`, `TrimImage`, `FCompressPicture`, `FCompressPictureFile`, `DrawPictureFile`, `DrawTrimmedPicture`, `DrawTrimmedPictureFile`, `MakeThumbnailFromPicture`, `MakeThumbnailFromPictureFile`, `MakeThumbnailFromPixmap`, `SetCompressedPixmapInfo`, and `GetCompressedPixmapInfo`. If you pass a value of -1 in the `progressProc` parameter of any of these functions, you obtain a standard progress function.

**RESULT CODES**

noErr	0	No error
paramErr	-50	Invalid parameter specified
codecAbortErr	-8967	Operation aborted by the progress function

**Completion Functions**

---

Compressor components call completion functions when they have finished an asynchronous operation. The component supplies a result code to your completion function. This result code indicates the success or failure of the asynchronous operation. Note that any other result data that may be produced by the asynchronous operation is not valid until the component calls your completion function.

The `CompletionProcPtr` data type defines a pointer to a completion function. You assign a completion function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate function.

```
typedef struct CompletionProcRecord CompletionProcRecord;
```

The completion function structure contains the following fields:

```
typedef CompletionProcRecord *CompletionProcRecordPtr;

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

**Field descriptions**

`completionProc`

Contains a pointer to your completion function. Your completion function may be called at interrupt time. Therefore, the value of the A5 register is unknown, and your function may not use Memory Manager functions or other functions that move memory.

`completionRefCon`

Contains a reference constant for use by your completion function.

**MyCompletionProc**

---

Your completion function should have the following form:

```
pascal OSErr MyCompletionProc (OSErr result, short flag,
                                long refcon);
```

## Image Compression Manager

result	Indicator of success of current operation.
flag	Indicates which part of the operation is complete. The following flags are defined: <div> <div>codecCompletionSource</div> <div>The Image Compression Manager is done with the source buffer. The Image Compression Manager sets this flag to 1 when it is done with the processing associated with the source buffer. For compression operations, the source is the uncompressed pixel map you are compressing. For decompression operations, the source is the decompressed data you are decompressing.</div> </div> <div> <div>codecCompletionDest</div> <div>The Image Compression Manager is done with the destination buffer. The Image Compression Manager sets this flag to 1 when it is done with the processing associated with the destination buffer.</div> </div>
	Note that more than one of these flags may be set to 1.
refcon	Contains a reference constant value for use by your completion function. Your application specifies the value of this reference constant in the callback function structure you pass to the Image Compression Manager.

## RESULT CODES

noErr      0      No error

## Alignment Functions

Your application can use alignment functions to specify the alignment in any of the Image Compression Manager's alignment functions (described in "Aligning Windows" beginning on page 3-146). You call the alignment function with a rectangle (defined in global screen coordinates) that has already been aligned using the default behavior. The alignment function then has the option of applying some additional alignment criteria to the rectangle, such as vertical alignment of some form. In the case of supporting hardware alignment, it is the function's responsibility to determine if the rectangle applies to the relevant device.

The `AlignmentProcPtr` data type defines a pointer to an alignment function. You assign an alignment function by passing a pointer to the alignment function structure, which identifies the alignment function to the appropriate function.

```
/* alignment function structure */
typedef struct
{
    AlignmentProcPtr alignmentProc; /* pointer to your
                                     alignment function */
    long alignmentRefCon; /* reference constant */
} AlignmentProcRecord, *AlignmentProcRecordPtr;
```

## Image Compression Manager

**Field descriptions**

`alignmentProc` Points to your alignment function.

`alignmentRefCon` Contains a reference constant for use by your alignment function.

## **MyAlignmentProc**

---

Your alignment function should have the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refcon);
```

`rp` Contains a pointer to a rectangle that has already been aligned with a default alignment function.

`refcon` Contains a reference constant value for use by your alignment function. Your application specifies the value of this reference constant in the alignment function structure you pass to the Image Compression Manager.

## Summary of the Image Compression Manager

---

### C Summary

---

#### Constants

---

```

/* determines if Image Compression Manager is available */
#define gestaltCompressionMgr 'icmp'

/* smallest data buffer you may allocate for image data spooling */
#define codecMinimumDataSize 32768

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

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

/* Image Compression Manager function control flags */
#define codecFlagUseImageBuffer (1L<<0) /* (input) use image buffer */
#define codecFlagUseScreenBuffer(1L<<1) /* (input) use screen buffer */
#define codecFlagUpdatePrevious (1L<<2) /* (input) update previous
                                         buffer */
#define codecFlagNoScreenUpdate (1L<<3) /* (input) don't update screen */
#define codecFlagWasCompressed (1L<<4) /* (input) image compressed */
#define codecFlagDontOffscreen (1L<<5) /* don't go offscreen
                                         automatically */
#define codecFlagUpdatePreviousComp (1L<<6)
                                         /* (input) update previous
                                         buffer */
#define codecFlagForceKeyFrame (1L<<7) /* force key frame from image */
#define codecFlagOnlyScreenUpdate
                                         (1L<<8) /* decompress current frame */
#define codecFlagLiveGrab (1L<<9) /* sequence from live video grab */
#define codecFlagDontUseNewImageBuffer (1L<<10)
                                         /* (input) don't use new image
                                         buffer */

```

## Image Compression Manager

```

#define codecFlagInterlaceUpdate (1L<<11)
                                /* (input) update screen
                                interlacing */

/*
    status flags from outflags parameter of DecompressSequenceFrame
    function
*/
#define codecFlagUsedNewImageBuffer (1L<<14)
                                /* (output) used new image buffer */
#define codecFlagUsedImageBuffer (1L<<15)
                                /* (output) used image buffer */

/* completion flags from application-defined completion functions */
#define codecCompletionSource (1<<0) /* Image Compression Manager done
                                with source buffer */
#define codecCompletionDest (1<<1) /* Image Compression Manager done with
                                destination buffer */

/* compression quality values */
#define codecMinQuality 0x000L /* minimum-quality image reproduction */
#define codecLowQuality 0x100L /* low-quality image reproduction */
#define codecNormalQuality 0x200L /* normal-quality image reproduction */
#define codecHighQuality 0x300L /* high-quality image reproduction */
#define codecMaxQuality 0x3FFL /* maximum-quality image reproduction */
#define codecLosslessQuality 0x400L /* lossless-quality reproduction */

/*
    special compressor and decompressor identifiers let you choose an
    image compressor component
*/
#define anyCodec (CodecComponent)0 /* first one or a
                                specified type */
#define bestSpeedCodec ((CodecComponent)-1) /* fastest of specified
                                type */
#define bestFidelityCodec (CodecComponent)-2 /* most accurate of
                                specified type */
#define bestCompressionCodec( (CodecComponent)-3) /* one with smallest
                                resulting data */

/*
    constants for doDither parameter of DrawTrimmedPictureFile and
    FCompressPictureFile functions
*/

```

## Image Compression Manager

```
#define defaultDither0      /* respect dithering instructions in
                           source picture */
#define forceDither1       /* dither image */
#define suppressDither2    /* don't dither image */
```

## Data Types

---

```
typedef Component CompressorComponent; /* compressor identifier */
typedef Component DecompressorComponent; /* decompressor identifier */
typedef Component CodecComponent;      /* compressor identifier */

typedef long CodecType;                 /* compressor type */

typedef unsigned short CodecFlags;      /* compressor component flags */

typedef unsigned long CodecQ;           /* compression quality */

typedef pascal OSErr (*DataProcPtr) (Ptr *dataP, long bytesNeeded,
                                     longrefCon); /* pointer to a data-loading function */

typedef pascal OSErr (*FlushProcPtr) (Ptr data, long bytesAdded,
                                     long refCon); /* pointer to a data-unloading function */

typedef pascal void (*CompletionProcPtr) (OSErr result, short flags,
                                          long refCon); /* pointer to a completion function */

typedef pascal OSErr (*ProgressProcPtr) (short message, Fixed completeness,
                                          long refCon); /* pointer to a progress function */

typedef long ImageSequence;             /* unique sequence identifier */

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

typedef struct ProgressProcRecord ProgressProcRecord;
typedef ProgressProcRecord *ProgressProcRecordPtr;

/* completion function structure */
struct CompletionProcRecord
{
```

## Image Compression Manager

```

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

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

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

typedef pascal void (*StdPixProcPtr)(PixMap *src, Rect *srcRect,
                                     MatrixRecord *matrix,
                                     short mode, RgnHandle mask,
                                     PixMap *matte, Rect *matteRect,
                                     short flags);

typedef struct
{
    AlignmentProcPtr alignmentProc; /* pointer to your alignment
                                     function */
    long alignmentRefCon; /* reference constant */
} AlignmentProcRecord;

typedef AlignmentProcRecord *AlignmentProcRecordPtr;

typedef struct
{
    long dataRate; /* bytes per second */
    long dataOverrun; /* number of bytes outside rate */
    long frameDuration; /* in milliseconds */

```



## Image Compression Manager

```

    long    keyFrameRate;          /* frequency of key frames */
    CodecQ   minSpatialQuality;     /* minimum spatial quality */
    CodecQ   minTemporalQuality;    /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;

/* image description structure */
struct ImageDescription
{
    long idSize;                   /* total size of this structure */
    CodecType cType;               /* compressor type of creator */
    long resvd1;                   /* reserved--must be set to 0 */
    short resvd2;                  /* reserved--must be set to 0 */
    short dataRefIndex;            /* reserved--must be set to 0 */
    short version;                 /* version of compressed data */
    short revisionLevel;           /* version of compressor that created data */
    long vendor;                   /* developer of compressor that created data */
    CodecQ temporalQuality;         /* degree of temporal compression */
    CodecQ spatialQuality;         /* degree of spatial compression */
    short width;                   /* width of source image in pixels */
    short height;                  /* height of source image in pixels */
    Fixed hRes;                    /* horizontal resolution of source image */
    Fixed vRes;                    /* vertical resolution of source image */
    long dataSize;                 /* size in bytes of compressed data */
    short frameCount;              /* number of frames in image data */
    Str31 name;                    /* name of compression algorithm */
    short depth;                   /* pixel depth of source image */
    short clutID;                  /* ID number of color table for image */
};
typedef struct ImageDescription ImageDescription;
typedef ImageDescription *ImageDescriptionPtr, **ImageDescriptionHandle;

/* compressor information structure */
struct CodecInfo
{
    Str31 typeName;                /* compression algorithm */
    short version;                 /* version of compressed data */
    short revisionLevel;           /* version of component */
    long vendor;                   /* developer of component */
    long decompressFlags;          /* decompression capability flags */
    long compressFlags;            /* compression capability flags */
    long formatFlags;              /* compression format flags */
    unsigned char compressionAccuracy;
                                   /* relative accuracy of compression */
};

```

## Image Compression Manager

```

unsigned char decompressionAccuracy;
                        /* relative accuracy of decompression */
unsigned short compressionSpeed;
                        /* relative speed of compressor */
unsigned short decompressionSpeed;
                        /* relative speed of decompressor */
unsigned char compressionLevel;
                        /* relative level of compression */
char resvd;            /* reserved--set to 0 */
short minimumHeight;   /* minimum height */
short minimumWidth;    /* minimum width */
short decompressPipelineLatency;
                        /* in milliseconds (asynchronous) */
short compressPipelineLatency;
                        /* in milliseconds (asynchronous) */
long privateData;      /* reserved for use by Apple */
};
typedef struct CodecInfo CodecInfo;

/* compressor name structure returned by GetCodecNameList function */
struct CodecNameSpec
{
    CodecComponent codec; /* component ID for compressor */
    CodecType cType;      /* type identifier for compressor */
    Str31 typeName;       /* string identifier of compression algorithm */
    Handle name;          /* name of compressor component */
};
typedef struct CodecNameSpec CodecNameSpec;

/* compressor name list structure */
struct CodecNameSpecList
{
    short count;          /* number of compressor name structures in list
                           array that follows */
    CodecNameSpec list[1]; /* array of compressor name structures */
};
typedef struct CodecNameSpecList CodecNameSpecList;
typedef CodecNameSpecList *CodecNameSpecListPtr;

/*
    flags from message parameter of application-defined progress functions
    tell why the Image Compression Manager called your function
*/
enum

```

## Image Compression Manager

```

{
    codecProgressOpen          = 0, /* start of a long operation */
    codecProgressUpdatePercent = 1, /* passes completion information */
    codecProgressClose         = 2  /* end of a long operation*/
};

typedef pascal void (*CompletionProcPtr) (OSErr result, short flags,
                                         long refCon);

/* data rate parameters structure */
typedef struct {
    long    dataRate;           /* bytes per second */
    long    dataOverrun;        /* number of bytes outside rate */
    long    frameDuration;      /* in milliseconds */
    long    keyFrameRate;       /* frequency of key frames */
    CodecQ   minSpatialQuality; /* minimum spatial quality */
    CodecQ   minTemporalQuality; /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;

```

## Image Compression Manager Functions

**Getting Information About Compressor Components**

```

pascal OSErr CodecManagerVersion
    (long *version);

pascal OSErr GetCodecNameList
    (CodecNameSpecListPtr *list, short showAll);

pascal OSErr DisposeCodecNameList
    (CodecNameSpecListPtr list);

pascal OSErr GetCodecInfo
    (CodecInfo *info, CodecType cType,
     CodecComponent codec);

pascal OSErr FindCodec
    (CodecType cType, CodecComponent specCodec,
     CompressorComponent *compressor,
     DecompressorComponent *decompressor);

```

**Getting Information About Compressed Data**

```

pascal OSErr GetMaxCompressionSize
    (PixMapHandle src, const Rect *srcRect,
     short colorDepth, CodecQ quality,
     CodecType cType, CompressorComponent codec,
     long *size);

```

## Image Compression Manager

```

pascal OSErr GetCompressionTime
    (PixMapHandle src, const Rect *srcRect,
     short colorDepth, CodecType cType,
     CompressorComponent codec,
     CodecQ *spatialQuality,
     CodecQ *temporalQuality,
     unsigned long *compressTime);

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

pascal OSErr GetCompressedImageSize
    (ImageDescriptionHandle desc, Ptr data,
     long bufferSize, DataProcRecordPtr dataProc,
     long *dataSize);

```

**Working With Images**

```

pascal OSErr CompressImage (PixMapHandle src, const Rect *srcRect,
    CodecQ quality, CodecType cType,
    ImageDescriptionHandle desc, Ptr data);

pascal OSErr FCompressImage
    (PixMapHandle src, const Rect *srcRect,
     short colorDepth, CodecQ quality,
     CodecType cType, CompressorComponent codec,
     CTabHandle clut, CodecFlags flags,
     long bufferSize, FlushProcRecordPtr flushProc,
     ProgressProcRecordPtr progressProc,
     ImageDescriptionHandle desc, Ptr data);

pascal OSErr DecompressImage
    (Ptr data, ImageDescriptionHandle desc,
     PixMapHandle dst, const Rect *srcRect,
     const Rect *dstRect, short mode,
     RgnHandle mask);

pascal OSErr FDecompressImage
    (Ptr data, ImageDescriptionHandle desc,
     PixMapHandle dst, const Rect *srcRect,
     MatrixRecordPtr matrix, short mode,
     RgnHandle mask, PixMapHandle matte,
     const Rect *matteRect, CodecQ accuracy,
     DecompressorComponent codec, long bufferSize,
     DataProcRecordPtr dataProc,
     ProgressProcRecordPtr progressProc);

```

## Image Compression Manager

```

pascal OSErr ConvertImage (ImageDescriptionHandle srcDD, Ptr srcData,
                           short colorDepth, CTabHandle clut,
                           CodecQ accuracy, CodecQ quality,
                           CodecType cType, CodecComponent codec,
                           ImageDescriptionHandle dstDD, Ptr dstData);

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

pascal OSErr SetImageDescriptionCTable
                        (ImageDescriptionHandle desc,
                        CTabHandle ctable);

pascal OSErr GetImageDescriptionCTable
                        (ImageDescriptionHandle desc,
                        CTabHandle *ctable);

```

**Working With Pictures and PICT Files**

```

pascal OSErr CompressPicture
                        (PicHandle srcPicture, PicHandle dstPicture,
                        CodecQ quality, CodecType cType);

pascal OSErr FCompressPicture
                        (PicHandle srcPicture, PicHandle dstPicture,
                        short colorDepth, CTabHandle clut,
                        CodecQ quality, short doDither,
                        short compressAgain,
                        ProgressProcRecordPtr progressProc,
                        CodecType cType, CompressorComponent codec);

pascal OSErr CompressPictureFile
                        (short srcRefNum, short dstRefNum,
                        CodecQ quality, CodecType cType);

pascal OSErr FCompressPictureFile
                        (short srcRefNum, short dstRefNum,
                        short colorDepth, CTabHandle clut,
                        CodecQ quality, short doDither,
                        short compressAgain,
                        ProgressProcRecordPtr progressProc,
                        CodecType cType, CompressorComponent codec);

pascal OSErr DrawPictureFile
                        (short refNum, const Rect *frame,
                        ProgressProcRecordPtr progressProc);

```

## Image Compression Manager

```

pascal OSErr DrawTrimmedPicture
    (PicHandle srcPicture, const Rect *frame,
     RgnHandle trimMask, short doDither,
     ProgressProcRecordPtr progressProc);

pascal OSErr DrawTrimmedPictureFile
    (short srcRefnum, const Rect *frame,
     RgnHandle trimMask, short doDither,
     ProgressProcRecordPtr progressProc);

pascal OSErr GetPictureFileHeader
    (short refNum, Rect *frame,
     OpenCPicParams *header);

```

**Making Thumbnail Pictures**

```

pascal OSErr MakeThumbnailFromPicture
    (PicHandle picture, short colorDepth,
     PicHandle thumbnail,
     ProgressProcRecordPtr progressProc);

pascal OSErr MakeThumbnailFromPictureFile
    (short refNum, short colorDepth,
     PicHandle thumbnail,
     ProgressProcRecordPtr progressProc);

pascal OSErr MakeThumbnailFromPixMap
    (PixMapHandle src, const Rect *srcRect,
     short colorDepth, PicHandle thumbnail,
     ProgressProcRecordPtr progressProc);

```

**Working With Sequences**

```

pascal OSErr CompressSequenceBegin
    (ImageSequence *seqID, PixMapHandle src,
     PixMapHandle prev, const Rect *srcRect,
     const Rect *prevRect, short colorDepth,
     CodecType cType, CompressorComponent codec,
     CodecQ spatialQuality, CodecQ temporalQuality,
     long keyFrameRate, CTabHandle clut,
     CodecFlags flags, ImageDescriptionHandle desc);

pascal OSErr CompressSequenceFrame
    (ImageSequence seqID, PixMapHandle src,
     const Rect *srcRect, CodecFlags flags,
     Ptr data, long *dataSize,
     unsigned char *similarity,
     CompletionProcRecordPtr asyncCompletionProc);

```

## Image Compression Manager

```

pascal OSErr DecompressSequenceBegin
    (ImageSequence *seqID,
     ImageDescriptionHandle desc, CGrafPtr port,
     GDHandle gdh, const Rect *srcRect,
     MatrixRecordPtr matrix, short mode,
     RgnHandle mask, CodecFlags flags,
     CodecQ accuracy, DecompressorComponent codec);

pascal OSErr DecompressSequenceFrame
    (ImageSequence seqID, Ptr data,
     CodecFlags inFlags, CodecFlags *outFlags,
     CompletionProcRecordPtr asyncCompletionProc);

pascal OSErr CDSequenceBusy
    (ImageSequence seqID);

pascal OSErr CDSequenceEnd (ImageSequence seqID);

```

**Changing Sequence-Compression Parameters**

```

pascal OSErr SetCSequenceQuality
    (ImageSequence seqID, CodecQ spatialQuality,
     CodecQ temporalQuality);

pascal OSErr SetCSequenceKeyFrameRate
    (ImageSequence seqID, long keyframerate);

pascal OSErr GetCSequenceKeyFrameRate
    (ImageSequence seqID, long *keyframerate);

pascal OSErr SetCSequenceFrameNumber
    (ImageSequence seqID, long frameNumber);

pascal OSErr GetCSequenceFrameNumber
    (ImageSequence seqID, long *frameNumber);

pascal OSErr SetCSequencePrev
    (ImageSequence seqID, PixMapHandle prev,
     const Rect *prevRect);

pascal OSErr SetCSequenceFlushProc
    (ImageSequence seqID,
     FlushProcRecordPtr flushProc, long bufferSize);

pascal OSErr GetCSequencePrevBuffer
    (ImageSequence seqID, GWorldPtr *gworld);

```

**Constraining Compressed Data**

```

pascal OSErr SetCSequenceDataRateParams
    (ImageSequence seqID,
     DataRateParamsPtr params);

pascal OSErr GetCSequenceDataRateParams
    (ImageSequence seqID, DataRateParamsPtr params);

```

## Image Compression Manager

**Changing Sequence-Decompression Parameters**

```

pascal OSErr SetDSequenceTransferMode
                (ImageSequence seqID, short mode,
                 const RGBColor *opColor);

pascal OSErr SetDSequenceSrcRect
                (ImageSequence seqID, const Rect *srcRect);

pascal OSErr SetDSequenceMatrix
                (ImageSequence seqID, MatrixRecordPtr matrix);

pascal OSErr SetDSequenceMask
                (ImageSequence seqID, RgnHandle mask);

pascal OSErr SetDSequenceMatte
                (ImageSequence seqID, PixMapHandle matte,
                 const Rect *matteRect);

pascal OSErr SetDSequenceAccuracy
                (ImageSequence seqID, CodecQ accuracy);

pascal OSErr SetDSequenceDataProc
                (ImageSequence seqID,
                 DataProcRecordPtr dataProc, long bufferSize);

pascal OSErr GetDSequenceImageBuffer
                (ImageSequence seqID, GWorldPtr *gworld);

pascal OSErr GetDSequenceScreenBuffer
                (ImageSequence seqID, GWorldPtr *gworld);

```

**Working With the StdPix Function**

```

pascal void StdPix
                (PixMapPtr src, const Rect *srcRect,
                 MatrixRecordPtr matrix, short mode,
                 RgnHandle mask, PixMapPtr matte,
                 Rect *matteRect, short flags);

pascal OSErr SetCompressedPixMapInfo
                (PixMapPtr pix, ImageDescriptionHandle desc,
                 Ptr data, long bufferSize,
                 DataProcRecordPtr dataProc,
                 ProgressProcRecordPtr progressProc);

pascal OSErr GetCompressedPixMapInfo
                (PixMapPtr pix, ImageDescriptionHandle *desc,
                 Ptr *data, long *bufferSize,
                 DataProcRecord *dataProc,
                 ProgressProcRecord *progressProc);

```



**Aligning Windows**

```

pascal void AlignWindow      (WindowPtr wp, Boolean front,
                             const Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);

pascal void DragAlignedWindow
                             (WindowPtr wp, Point startPt,
                             Rect *boundsRect, Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);

pascal long DragAlignedGrayRgn
                             (RgnHandle theRgn, Point startPt,
                             Rect *boundsRect, Rect *slopRect, short axis,
                             ProcPtr actionProc, Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);

pascal void AlignScreenRect
                             (Rect *rp,
                             AlignmentProcRecordPtr alignmentProc);

```

**Working With Graphics Devices and Graphics Worlds**

```

pascal OSErr GetBestDeviceRect
                             (GDHandle *gdh, Rect *rp);

pascal QDErr NewImageGWorld
                             (GWorldPtr *gworld,
                             ImageDescription **idh, GWorldFlags flags);

```

**Application-Defined Functions**

---

**Data-Loading Functions**

```

pascal OSErr MyDataLoadingProc
                             (Ptr *dataP, long bytesNeeded, long refcon);

```

**Data-Unloading Functions**

```

pascal OSErr MyDataUnloadingProc
                             (Ptr data, long bytesAdded, long refcon);

```

**Progress Functions**

```

pascal OSErr MyProgressProc
                             (short message, Fixed completeness,
                             long refcon);

```

**Completion Functions**

```
pascal OSErr MyCompletionProc
                                (OSErr result, short flag, long refcon);
```

**Alignment Functions**

```
pascal void MyAlignmentProc
                                (Rect *rp, long refcon);
```

**Pascal Summary**

---

**Constants**

---

```
CONST
    gestaltCompressionMgr          = 'icmp'; {determines if Image }
                                       { Compression Manager is }
                                       { available}

    codecMinimumDataSize           = 32768; {smallest data buffer you may }
                                       { allow for data spooling}

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

    {Image Compression Manager function control flags}
    codecFlagUseImageBuffer         = $1;    {(input) use offscreen buffer}
    codecFlagUseScreenBuffer        = $2;    {(input) use screen buffer}
    codecFlagUpdatePrevious         = $4;    {(input) previous image is }
                                       { updated}

    codecFlagNoScreenUpdate         = $8;    {(input) no screen image update}
    codecFlagWasCompressed          = $10;   {(input) image has been }
                                       { compressed}

    codecFlagDontOffscreen          = $20;   {don't use offscreen buffer}
    codecFlagUpdatePreviousComp     = $40;   {(input) previous image buffer }
                                       { updated}

    codecFlagForceKeyFrame          = $80;   {force key frame from image}
    codecFlagOnlyScreenUpdate       = $100;  {decompresses current frame}
    codecFlagLiveGrab               = $200;  {sequence from live video grab}
```

## Image Compression Manager

```

codecFlagDontUseNewImageBuffer    = $400;  {(input) return error if image }
                                         { buffer is new or reallocated}
codecFlagInterlaceUpdate          = $800;  {(input) use interlaced update}

{status flags from outflags parameter of DecompressSequenceFrame function}
codecFlagUsedNewImageBuffer       = $4000; {(output) used new image buffer}
codecFlagUsedImageBuffer         = $8000; {(output) used offscreen image }
                                         { buffer}

{completion flags from application-defined completion functions}
codecCompletionSource             = 1;      {done with source buffer}
codecCompletionDest               = 2;      {done with destination buffer}

{flags from application-defined progress functions message parameter-- }
{ tell why the Image Compression Manager called your function}
codecProgressOpen                 = 0;      {start of a long operation}
codecProgressUpdatePercent        = 1;      {passing completion data}
codecProgressClose                = 2;      {end of a long}

{compression quality values}
codecMinQuality                   = $000;  {minimum-quality image reproduction}
codecLowQuality                   = $100;  {low-quality image reproduction}
codecNormalQuality                = $200;  {normal-quality image reproduction}
codecHighQuality                  = $300;  {high-quality image reproduction}
codecMaxQuality                   = $3FF;  {maximum-quality image reproduction}
codecLosslessQuality              = $400;  {lossless-quality image reproduction}

{special compressor and decompressor identifiers}
anyCodec                          = 0;  {first component of specified type}
bestSpeedCodec                    = -1;  {fastest component of specified type}
bestFidelityCodec                 = -2;  {most accurate component of specified type}
bestCompressionCodec              = -3;  {component with smallest resulting data}

{constants for doDither parameter of DrawTrimmedPictureFile and }
{ FCompressPictureFile functions}

defaultDither                     = 0;  {respect dithering instructions in source }
                                         { picture}
forceDither                       = 1;  {dither image}
suppressDither                    = 2;  {don't dither image}

```

## Data Types

---

### TYPE

```

CompressorComponent      = Component; {compressor identifier}
DecompressorComponent    = Component; {decompressor identifier}
CodecComponent           = Component; {compressor identifier}

CodecType                 = OSType;    {compressor type}

CodecFlags                = Integer;   {compressor component flags}

CodecQ                    = LongInt;   {compression quality}

DataProcPtr              = ProcPtr;   {pointer to a data-loading function}
FlushProcPtr             = ProcPtr;   {pointer to a data-unloading function}
CompletionProcPtr        = ProcPtr;   {pointer to a completion function}
ProgressProcPtr          = ProcPtr;   {pointer to a progress function}

ImageSequence            = LongInt;   {unique sequence identifier}

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

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

DataProcRecordPtr      = ^DataProcRecord;
DataProcRecord =       {data-loading function record}
RECORD
    dataProc:    DataProcPtr;          {pointer to data-loading function}
    dataRefCon:  LongInt;              {reference constant}
END;

FlushProcRecordPtr     = ^FlushProcRecord;
FlushProcRecord =      {data-unloading function record}

```

## Image Compression Manager

```

RECORD
    flushProc:      FlushProcPtr;      {pointer to data-unloading }
                                         { function}
    flushRefCon:    LongInt;            {reference constant}
END;

ImageDescriptionPtr      = ^ImageDescription;
ImageDescriptionHandle   = ^ImageDescriptionPtr;
ImageDescription =
PACKED RECORD
    idSize:          LongInt;          {total size of this record}
    cType:           CodecType;        {type of creator component}
    resvd1:          LongInt;          {reserved--must be set to 0}
    resvd2:          Integer;          {reserved--must be set to 0}
    dataRefIndex:    Integer;          {reserved--must be set to 0}
    version:         Integer;          {version of compressed data}
    revisionLevel:   Integer;          {version of creator compressor}
    vendor:          LongInt;          {developer of creator compressor}
    temporalQuality: CodecQ;            {degree of temporal compression}
    spatialQuality:  CodecQ;            {degree of spatial compression}
    width:           Integer;          {width of source image in pixels}
    height:          Integer;          {height of source image in pixels}
    hRes:            Fixed;            {horizontal resolution of source image}
    vRes:            Fixed;            {vertical resolution of source image}
    dataSize:        LongInt;          {byte size of compressed image data}
    frameCount:      Integer;          {number of frames in image data}
    name:            PACKED ARRAY[0..31] of char;
                                         {name of compression algorithm}
    depth:           Integer;          {pixel depth of source image}
    clutID:          Integer;          {ID number of the color table for image}
END;

CodecInfo = {compressor information record}
PACKED RECORD
    typeName:        PACKED ARRAY[0..31] of char;
                                         {compression algorithm}
    version:         Integer;          {version of compressed data}
    revisionLevel:   Integer;          {version of component}
    vendor:          LongInt;          {developer of component}
    decompressFlags: LongInt;          {decompression capability flags}
    compressFlags:   LongInt;          {compression capability flags}
    formatFlags:     LongInt;          {format flags}
    compressionAccuracy:
        Char;          {relative accuracy of compression}

```

## Image Compression Manager

```

decompressionAccuracy:
    Char;          {relative accuracy of decompression}
compressionSpeed:   Integer;    {relative compression speed}
decompressionSpeed: Integer;    {relative decompression speed}
compressionLevel:   Char;       {relative compression of component}
resvd:              Char;       {reserved--set to 0}
minimumHeight:      Integer;    {minimum height in pixels}
minimumWidth:       Integer;    {maximum width in pixels}
decompressPipelineLatency:
    Integer;       {milliseconds (asynchronous)}
compressPipelineLatency:
    Integer;       {milliseconds (asynchronous)}
privateData:        LongInt;    {reserved--must be set to 0}
END;

{compressor name record returned by GetCodecNameList}
CodecNameSpec =
PACKED RECORD
    codec:      CodecComponent; {component ID for compressor}
    cType:      CodecType;      {type identifier for compressor}
    typeName:   PACKED ARRAY[0..31] OF Char;
                                {string identifier of compression algorithm}
    name:       Handle;         {name of compressor component}
END;

{compressor name list record}
CodecNameSpecListPtr = ^CodecNameSpecList;
CodecNameSpecList =
RECORD
    count:      Integer;        {number of compressor name records}
    list:       ARRAY[0..0] OF CodecNameSpec;
                                {array of compressor name records}
END;

{data rate parameters record}
DataRateParamsPtr = ^DataRateParams;
DataRateParams =
RECORD
    dataRate:      LongInt;      {bytes per second}
    dataOverrun:   LongInt;      {number of bytes outside rate}
    frameDuration: LongInt;      {duration in milliseconds}
    keyFrameRate:  LongInt;      {frequency of key frames}

```

## Image Compression Manager

```

    minSpatialQuality:   CodecQ;      {minimum spatial quality}
    minTemporalQuality:  CodecQ;      {minimum temporal quality}
END;
```

## Image Compression Manager Routines

**Getting Information About Compressor Components**

```

FUNCTION CodecManagerVersion
    (VAR version: LongInt): OSerr;

FUNCTION GetCodecNameList
    (VAR list: CodecNameSpecListPtr;
     showAll: Integer): OSerr;

FUNCTION DisposeCodecNameList
    (list: CodecNameSpecListPtr): OSerr;

FUNCTION GetCodecInfo
    (VAR info: CodecInfo; cType: CodecType;
     codec: CodecComponent): OSerr;

FUNCTION FindCodec
    (cType: CodecType; specCodec: CodecComponent;
     VAR compressor: CompressorComponent;
     VAR decompressor: DecompressorComponent):
    OSerr;
```

**Getting Information About Compressed Data**

```

FUNCTION GetMaxCompressionSize
    (src: PixMapHandle; srcRect: Rect;
     colorDepth: Integer; quality: CodecQ;
     cType: CodecType; codec: CompressorComponent;
     VAR size: LongInt): OSerr;

FUNCTION GetCompressionTime
    (src: PixMapHandle; srcRect: Rect;
     colorDepth: Integer; cType: CodecType;
     codec: CompressorComponent;
     VAR spatialQuality: CodecQ;
     VAR temporalQuality: CodecQ;
     VAR compressTime: LongInt): OSerr;

FUNCTION GetSimilarity
    (src: PixMapHandle; srcRect: Rect;
     desc: ImageDescriptionHandle; data: Ptr;
     VAR similarity: Fixed): OSerr;

FUNCTION GetCompressedImageSize
    (desc: ImageDescriptionHandle; data: Ptr;
     bufferSize: LongInt;
     dataProc: DataProcRecordPtr;
     VAR dataSize: LongInt): OSerr;
```

## Image Compression Manager

**Working With Images**

```

FUNCTION CompressImage      (src: PixMapHandle; srcRect: Rect;
                             quality: CodecQ; cType: CodecType;
                             desc: ImageDescriptionHandle;
                             data: Ptr): OSErr;

FUNCTION FCompressImage     (src: PixMapHandle; srcRect: Rect;
                             colorDepth: Integer; quality: CodecQ;
                             cType: CodecType; codec: CompressorComponent;
                             clut: CTabHandle; flags: CodecFlags;
                             bufferSize: LongInt;
                             flushProc: FlushProcRecordPtr;
                             progressProc: ProgressProcRecordPtr;
                             desc: ImageDescriptionHandle;
                             data: Ptr): OSErr;

FUNCTION DecompressImage    (data: Ptr; desc: ImageDescriptionHandle;
                             dst: PixMapHandle; srcRect: Rect;
                             dstRect: Rect; mode: Integer;
                             mask: RgnHandle): OSErr;

FUNCTION FDecompressImage   (data: Ptr; desc: ImageDescriptionHandle;
                             dst: PixMapHandle; srcRect: Rect;
                             matrix: MatrixRecordPtr; mode: Integer;
                             mask: RgnHandle; matte: PixMapHandle;
                             matteRect: Rect; accuracy: CodecQ;
                             codec: DecompressorComponent;
                             bufferSize: LongInt;
                             dataProc: DataProcRecordPtr;
                             progressProc: ProgressProcRecordPtr): OSErr;

FUNCTION ConvertImage       (srcDD: ImageDescriptionHandle; srcData: Ptr;
                             colorDepth: Integer; clut: CTabHandle;
                             accuracy: CodecQ; quality: CodecQ;
                             cType: CodecType; codec: CodecComponent;
                             dstDD: ImageDescriptionHandle;
                             dstData: Ptr): OSErr;

FUNCTION TrimImage          (desc: ImageDescriptionHandle; inData: Ptr;
                             inBufferSize: LongInt;
                             dataProc: DataProcRecordPtr; outData: Ptr;
                             outBufferSize: LongInt;
                             flushProc: FlushProcRecordPtr;
                             VAR trimRect: Rect;
                             progressProc: ProgressProcRecordPtr): OSErr;

FUNCTION SetImageDescriptionCTable
                             (desc: ImageDescriptionHandle;
                             ctable: CTabHandle): OSErr;

```



## Image Compression Manager

```

FUNCTION GetImageDescriptionCTable
    (desc: ImageDescriptionHandle;
     VAR ctable: CTabHandle): OSerr;

```

**Working With Pictures and PICT Files**

```

FUNCTION CompressPicture    (srcPicture: PicHandle; dstPicture: PicHandle;
                           quality: CodecQ; cType: CodecType): OSerr;

FUNCTION FCompressPicture  (srcPicture: PicHandle; dstPicture: PicHandle;
                           colorDepth: Integer; clut: CTabHandle;
                           quality: CodecQ; doDither: Integer;
                           compressAgain: Integer;
                           progressProc: ProgressProcRecordPtr;
                           cType: CodecType;
                           codec: CompressorComponent): OSerr;

FUNCTION CompressPictureFile
    (srcRefNum: Integer; dstRefNum: Integer;
     quality: CodecQ; cType: CodecType): OSerr;

FUNCTION FCompressPictureFile
    (srcRefNum: Integer; dstRefNum: Integer;
     colorDepth: Integer; clut: CTabHandle;
     quality: CodecQ; doDither: Integer;
     compressAgain: Integer;
     progressProc: ProgressProcRecordPtr;
     cType: CodecType;
     odec: CompressorComponent): OSerr;

FUNCTION DrawPictureFile   (refNum: Integer; frame: Rect;
                           progressProc: ProgressProcRecordPtr): OSerr;

FUNCTION DrawTrimmedPicture
    (srcPicture: PicHandle; frame: Rect;
     trimMask: RgnHandle; doDither: Integer;
     progressProc: ProgressProcPtr): OSerr;

FUNCTION DrawTrimmedPictureFile
    (srcRefnum: Integer; frame: Rect;
     trimMask: RgnHandle; doDither: Integer;
     progressProc: ProgressProcRecordPtr): OSerr;

FUNCTION GetPictureFileHeader
    (refNum: Integer; VAR frame: Rect;
     VAR header: OpenCPicParams): OSerr;

```

**Making Thumbnail Pictures**

```

FUNCTION MakeThumbnailFromPicture
    (picture: PicHandle; colorDepth: Integer;
     thumbnail: PicHandle;
     progressProc: ProgressProcRecordPtr): OSerr;

```

## Image Compression Manager

```

FUNCTION MakeThumbnailFromPictureFile
    (refNum: Integer; colorDepth: Integer;
     thumbnail: PicHandle;
     progressProc: ProgressProcRecordPtr): OSErr;

FUNCTION MakeThumbnailFromPixMap
    (src: PixMapHandle; srcRect: Rect;
     colorDepth: Integer; thumbnail: PicHandle;
     progressProc: ProgressProcRecordPtr): OSErr;

```

**Working With Sequences**

```

FUNCTION CompressSequenceBegin
    (VAR seqID: ImageSequence;
     src: PixMapHandle; prev: PixMapHandle;
     srcRect: Rect; prevRect: Rect;
     colorDepth: Integer; cType: CodecType;
     codec: CompressorComponent;
     spatialQuality: CodecQ;
     temporalQuality: CodecQ;
     keyFrameRate: LongInt; clut: CTabHandle;
     flags: CodecFlags;
     desc: ImageDescriptionHandle): OSErr;

FUNCTION CompressSequenceFrame
    (seqID: ImageSequence;
     src: PixMapHandle; srcRect: Rect;
     flags: CodecFlags; data: Ptr;
     VAR dataSize: LongInt; VAR similarity: Char;
     asyncCompletionProc: CompletionProcRecordPtr):
     OSErr;

FUNCTION DecompressSequenceBegin
    (VAR seqID: ImageSequence;
     desc: ImageDescriptionHandle; port: CGrafPtr;
     gdh: GDHandle; srcRect: Rect;
     matrix: MatrixRecordPtr; mode: Integer;
     mask: RgnHandle; flags: CodecFlags;
     accuracy: CodecQ;
     codec: DecompressorComponent): OSErr;

FUNCTION DecompressSequenceFrame
    (seqID: ImageSequence; data: Ptr;
     inFlags: CodecFlags; VAR outFlags: CodecFlags;
     asyncCompletionProc: CompletionProcRecordPtr):
     OSErr;

FUNCTION CDSequenceBusy    (seqID: ImageSequence): OSErr;
FUNCTION CDSequenceEnd    (seqID: ImageSequence): OSErr;

```

**Changing Sequence-Compression Parameters**

```

FUNCTION SetCSequenceQuality
    (seqID: ImageSequence; spatialQuality: CodecQ;
     temporalQuality: CodecQ): OSerr;

FUNCTION SetCSequenceKeyFrameRate
    (seqID: ImageSequence;
     keyframerate: LongInt): OSerr;

FUNCTION GetCSequenceKeyFrameRate
    (seqID: ImageSequence;
     VAR keyframerate: LongInt): OSerr;

FUNCTION GetCSequenceKeyFrameRate
    (seqID: ImageSequence;
     VAR keyframerate: LongInt): OSerr;

FUNCTION SetCSequenceFrameNumber
    (seqID: ImageSequence;
     frameNumber: LongInt): OSerr;

FUNCTION GetCSequenceFrameNumber
    (seqID: ImageSequence;
     VAR frameNumber: LongInt): OSerr;

FUNCTION SetCSequencePrev
    (seqID: ImageSequence; prev: PixMapHandle;
     prevRect: Rect): OSerr;

FUNCTION SetCSequenceFlushProc
    (seqID: ImageSequence;
     flushProc: FlushProcRecordPtr;
     bufferSize: LongInt): OSerr;

FUNCTION GetCSequencePrevBuffer
    (seqID: ImageSequence;
     VAR gworld: GWorldPtr): OSerr;

```

**Constraining Compressed Data**

```

FUNCTION SetCSequenceDataRateParams
    (seqID: ImageSequence;
     params: DataRateParamsPtr): OSerr;

FUNCTION GetCSequenceDataRateParams
    (seqID: ImageSequence;
     params: DataRateParamsPtr): OSerr;

```

**Changing Sequence-Decompression Parameters**

```

FUNCTION SetDSequenceTransferMode
    (seqID: ImageSequence; mode: Integer;
     opColor: RGBColor): OSerr;

```

## Image Compression Manager

```

FUNCTION SetDSequenceSrcRect
    (seqID: ImageSequence; srcRect: Rect): OSErr;

FUNCTION SetDSequenceMatrix
    (seqID: ImageSequence;
     matrix: MatrixRecordPtr): OSErr;

FUNCTION SetDSequenceMask    (seqID: ImageSequence; mask: RgnHandle): OSErr;
FUNCTION SetDSequenceMatte   (seqID: ImageSequence; matte: PixMapHandle;
                              matteRect: Rect): OSErr;

FUNCTION SetDSequenceAccuracy
    (seqID: ImageSequence;
     accuracy: CodecQ): OSErr;

FUNCTION SetDSequenceDataProc
    (seqID: ImageSequence;
     dataProc: DataProcRecordPtr;
     bufferSize: LongInt): OSErr;

FUNCTION GetDSequenceImageBuffer
    (seqID: ImageSequence;
     VAR gworld: GWorldPtr): OSErr;

FUNCTION GetDSequenceScreenBuffer
    (seqID: ImageSequence;
     VAR gworld: GWorldPtr): OSErr;

```

**Working With the StdPix Routine**

```

FUNCTION StdPix
    (src: PixMapPtr; srcRect: Rect;
     matrix: MatrixRecordPtr; mode: Integer;
     mask: RgnHandle; matte: PixMapPtr;
     matteRect: Rect; flags: Integer): OSErr

FUNCTION SetCompressedPixMapInfo
    (pix: PixMapPtr; desc: ImageDescriptionHandle;
     data: Ptr; bufferSize: LongInt;
     dataProc: DataProcRecordPtr;
     progressProc: ProgressProcRecordPtr): OSErr;

FUNCTION GetCompressedPixMapInfo
    (pix: PixMapPtr;
     VAR desc: ImageDescriptionHandle;
     VAR data: Ptr; VAR bufferSize: LongInt;
     VAR dataProc: DataProcRecord;
     VAR progressProc: ProgressProcRecord): OSErr;

```

**Aligning Windows**

```

PROCEDURE AlignWindow
    (wp: WindowPtr; front: Boolean;
     alignmentRect: RectPtr;
     alignmentProc: AlignmentProcRecordPtr );

```

## Image Compression Manager

```
PROCEDURE DragAlignedWindow
```

```
(wp: WindowPtr; startPt: Point;
VAR boundsRect: Rect; VAR alignmentRect: Rect;
alignmentProc: AlignmentProcRecordPtr);
```

```
FUNCTION DragAlignedGrayRgn
```

```
(theRgn: RgnHandle; startPt: Point;
VAR boundsRect: Rect; VAR slopRect: Rect;
axis: Integer; actionProc: ProcPtr;
VAR alignmentRect: Rect;
alignmentProc: AlignmentProcRecordPtr): LongInt;
```

```
PROCEDURE AlignScreenRect
```

```
(VAR rp: Rect;
alignmentProc: AlignmentProcRecordPtr);
```

### Working With Graphics Devices and Graphics Worlds

```
FUNCTION GetBestDeviceRect (VAR gdh: GDHandle; VAR rp: Rect): OSErr;
```

```
FUNCTION NewImageGWorld (VAR gworld: GWorldPtr;
idh: ImageDescriptionHandle;
flags :GWorldFlags): OSErr;
```

### Application-Defined Routines

---

#### Data-Loading Functions

```
FUNCTION MyDataLoadingProc (VAR dataP: Ptr; bytesNeeded: LongInt;
refcon: LongInt): OSErr;
```

#### Data-Unloading Functions

```
FUNCTION MyDataUnloadingProc
(data: Ptr; bytesAdded: LongInt;
refcon: LongInt): OSErr;
```

#### Progress Functions

```
FUNCTION MyProgressProc (message: Integer; completeness: Fixed;
refcon: LongInt): OSErr;
```

#### Completion Functions

```
FUNCTION MyCompletionProc (result: OSErr; flag: Integer;
refcon: LongInt): OSErr;
```

#### Alignment Routines

```
PROCEDURE MyAlignmentProc (rp: RectPtr, refcon: LongInt);
```

## Result Codes

---

paramErr	-50	Invalid parameter specified
memFullErr	-108	Not enough memory available
codecErr	-8960	General error condition
noCodecErr	-8961	Image Compression Manager could not find the specified compressor
codecUnimpErr	-8962	Feature not implemented by this compressor
codecSizeErr	-8963	Invalid buffer size specified
codecScreenBufErr	-8964	Could not allocate the screen buffer
codecImageBufErr	-8965	Could not allocate the image buffer
codecSpoolErr	-8966	Error loading or unloading data
codecAbortErr	-8967	Operation aborted by the progress function
codecWouldOffScreenErr	-8968	Compressor would use screen buffer if it could
codecBadDataErr	-8969	Compressed data contains inconsistencies
codecDataVersErr	-8970	Compressor does not support the compression version used to compress the image
codecExtensionNotFoundErr	-8971	Requested extension is not in the image description
codecConditionErr	-8972	Component cannot perform requested operation
codecOpenErr	-8973	Could not open the compressor or decompressor