QuickTime Compression and Decompression Guide

QuickTime > Compression & Decompression

2006-01-10

Apple Inc.
© 2005, 2006 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, ColorSync, FireWire, Mac, Macintosh, MovieTalk, QuickDraw, and QuickTime are trademarks of Apple Inc., registered in the United States and other countries.

PowerPC and and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to QuickTime Compression and Decompression Guide 11**

Organization of This Document 12
See Also 13

Chapter 1 **The Image Compression Manager 15**

Overview of the ICM 15
Data That Is Suitable for Compression 16
Storing Images 17
Working With Pictures 17
Understanding Compressor Components 19
Banding and Extending Images 20
Fast Dithering 22

Chapter 2 **Extensions to the Image Compression Manager 23**

ColorSync Support 23
Asynchronous Decompression 23
Timecode Support 23
Data Source Support 23
Working with Alpha Channels 24
Working With Video Fields 25
Packetization Information 25

Chapter 3 **Image Compression Characteristics 27**

Compression Ratio 27
Compression Speed 27
Image Quality 28

Chapter 4 **Compressors Supplied by Apple 29**

The Photo Compressor 29
The Video Compressor 29
The Compact Video Compressor 30
The Animation Compressor 30
The Graphics Compressor 30
The Raw Compressor 31
Types of Images Suitable for Different Compressors 31

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Chapter 5 **Working with the Image Compression Manager 43**

Getting Information About Compressors and Compressed Data 43
Getting Information About Compressor Components 43
Getting Information About Compressed Data 44
Compressing Images 44
Spooling Compressed Data 46
Application-Defined Functions 47
Changing Sequence-Compression Parameters 47
Changing Sequence-Decompression Parameters 48
Working With Images 48
Working With Sequences 49
Working With Pictures and PICT Files 49
Decompressing Images 49
Image Transcoding Functions 50
The Image Description Structure 51
Compression Quality Constants 53
The Compressor Name List Structure 54

Chapter 6 **How to Compress and Decompress Sequences of Images 55**

Compressing Sequences 55
Decompressing Sequences 56
- The Basic Functions to Use 57
- Decompressing Still Images From a Sequence 58
- Using Screen Buffers and Image Buffers 58

Defining Key Frame Rates 58
A Sample Program for Compressing and Decompressing a Sequence of Images 59
- A Sample Function for Saving a Sequence of Images to a Disk File 60
- A Sample Function for Creating, Compressing, and Drawing a Sequence of Images 61
- A Sample Function for Decompressing and Playing Back a Sequence From a Disk File 64

Chapter 7 **ICM Functions, Data Types, and Constants 67**

Making Thumbnail Pictures 67
Constraining Compressed Data 67
The Compressor Information Structure 68
The Compressor Name Structure 71
Controlling Hardware Scaling 72
Working With the StdPix Function 72
Aligning Windows 72
Alignment Functions 73
Working With Graphics Devices and Graphics Worlds 73
Data-Loading Functions 74
Data-Unloading Functions 74
Progress Functions 75

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Completion Functions 75
Constants 76
Image Compression Manager Function Control Flags 77

Chapter 8 About Image Compressor Components 81

Compressor Types 81
Utility and Callback Functions 81
Banding and Extending Images 82
Compressing or Decompressing Images Asynchronously 83
Spooling of Compressed Data 84
    Data Loading 84
    Data Unloading 85
Progress Functions 86

Chapter 9 Using Image Compressor Components 87

Performing Image Compression 87
    Choosing a Compressor 87
    Compressing a Horizontal Band of an Image 89
Decompressing an Image 91
    Choosing a Decompressor 92
    Decompressor Operations 93
    Decompressing a Horizontal Band of an Image 95
Asynchronous Decompression 98
Hardware Cursors 99
Timecode Support 99
Working With Video Fields 99
Accelerated Video Support 99
Packetization Information 102
DV Image Compressor Component 103
DV Image Decompressor Component 103
Specifying the Size of an Image Buffer 103

Chapter 10 Codec Components API 105

Data Structures 105
Functions 105
    Direct Functions 106
    Indirect Functions 106
    Image Compression Manager Utility Functions 107

Chapter 11 About the Base Image Decompressor 109

Using the Base Image Decompressor 109
Connecting to the Base Image Decompressor 109

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Providing Storage for Frame Decompression 110
Initializing Your Decompressor Component 110
Specifying Other Capabilities of Your Component 110
Implementing Functions for Queues 112
Decompressing Bands 112
Implementing ImageCodecBeginBand 112
Implementing ImageCodecDrawBand 114
Implementing ImageCodecEndBand 115
Providing Information About the Decompressor 115
Providing Progress Information 116
Handling and Delegating Other Calls 116
Closing the Component 116

Chapter 12 **Using Data Codec Components 117**

Component Types 117
Functions 118

Chapter 13 **Standard Image Compression Dialog Components 119**

Types of Dialog Boxes 119

Chapter 14 **Working With Standard Image Compression Dialog Components 121**

Opening a Connection to a Standard Image Compression Dialog Component 122
Displaying the Dialog Box to the User 123
Setting Default Parameters 123
Designating a Test Image 123
Displaying the Dialog Box and Retrieving Parameters 125
Getting Default Settings for an Image or a Sequence 125
Working With Image or Sequence Settings 126
Extending the Basic Dialog Box 126
Creating a Standard Image Compression Dialog Component 128

Chapter 15 **Image Compression Dialog Types and Functions 129**

Request Types 129
- Spatial Settings Request Type 129
- Temporal Settings Request Type 131
- Data-Rate Settings Request Type 132
- Color Table Settings Request Type 133
- Progress Function Request Type 133
- Extended Functions Request Type 133
- Preference Flags Request Type 134
- Settings State Request Type 136
- Sequence ID Request Type 136

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Window Position Request Type 136
Control Flags Request Type 136
Standard image compression Dialog Component Functions 137
Displaying the Standard image compression Dialog Box 137
Compressing Still Images 137
Compressing Image Sequences 138
Specifying a Test Image 138
Positioning Dialog Boxes and Rectangles 138

Chapter 16 **Using Image Transcoder Components 139**

Invoking an Image Transcoding Process 139
Transcoding Paths 140

Chapter 17 **Creating Image Transcoder Components 143**

An Example 143

Document Revision History 145

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figures, Tables, and Listings

Chapter 1 **The Image Compression Manager 15**

Figure 1-1 Image bands and their measurements 21
Table 1-1 Fields of the PICT opcode for compressed QuickTime images 18
Table 1-2 Fields of the PICT opcode for uncompressed QuickTime images 19

Chapter 4 **Compressors Supplied by Apple 29**

Figure 4-1 24-bit photographic image 32
Figure 4-2 A 24-bit synthetic image 33
Figure 4-3 An 8-bit graphic image 34
Figure 4-4 An 8-bit photographic image 34
Figure 4-5 Compressor performance for a 921 KB, 24-bit, photographic image 36
Figure 4-6 Compressor performance for a 502 KB, 24-bit, synthetic image 38
Figure 4-7 Compressor performance for a 30 KB, 8-bit, graphic image 40
Figure 4-8 Compressor performance for a 302 KB, 8-bit, dithered, photographic image 42

Chapter 5 **Working with the Image Compression Manager 43**

Listing 5-1 Compressing and decompressing an image 45

Chapter 6 **How to Compress and Decompress Sequences of Images 55**

Listing 6-1 Compressing and decompressing a sequence of images: The main program 59
Listing 6-2 Saving a sequence of images to a disk file 60
Listing 6-3 Creating and compressing an image sequence 62
Listing 6-4 Playing back a sequence of images from a disk file 64

Chapter 8 **About Image Compressor Components 81**

Figure 8-1 Image bands and their measurements 85

Chapter 9 **Using Image Compressor Components 87**

Listing 9-1 Preparing for simple compression operations 88
Listing 9-2 Performing simple compression on a horizontal band of an image 89
Listing 9-3 Preparing for simple decompression 92
Listing 9-4 Performing a decompression operation 95
Listing 9-5 Specifying the size of an image buffer for a codec 104

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Chapter 11 **About the Base Image Decompressor 109**

Listing 11-1 Connecting to the base image decompressor component 109
Listing 11-2 Specifying the capabilities of a decompressor component. 110
Listing 11-3 Sample implementation of ImageCodecPreflight 110
Listing 11-4 Sample implementation of ImageCodecBeginBand 112
Listing 11-5 Sample implementation of ImageCodecDrawBand 114

Chapter 13 **Standard Image Compression Dialog Components 119**

Figure 13-1 Dialog box for single-frame compression 120
Figure 13-2 Dialog box for image-sequence compression 120

Chapter 14 **Working With Standard Image Compression Dialog Components 121**

Figure 14-1 Elements of the standard image compression dialog box 122
Listing 14-1 Specifying a test image 124
Listing 14-2 Displaying the dialog box to the user and compressing an image 125
Listing 14-3 Defining a custom button in the dialog box 127
Listing 14-4 A sample hook function 127
Listing 14-5 Positioning related dialog boxes 127

Chapter 17 **Creating Image Transcoder Components 143**

Listing 17-1 An image transcoder component that converts a compressed data format to uncompressed RGB pixels 143

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Introduction to QuickTime Compression and Decompression Guide

This book introduces you to the QuickTime Image Compression Manager and its associated components, which provide image-compression and image-decompression services to applications and to other QuickTime components. The components used to compress and decompress movies, images, and image sequences fall into these classes:

- The Image Compression Manager, which lets your application
  - use a common interface for all image-compression and image-decompression operations;
  - take advantage of any compression software or hardware that may be present in a given Macintosh configuration;
  - store compressed image data in pictures;
  - temporally compress sequences of images, further reducing the storage requirements of movies;
  - display compressed PICT files without the need to modify your application; and
  - use an interface that is appropriate for your application: a high-level interface if you do not need to manipulate many compression parameters or a low-level interface that provides you greater control over the compression operation.

- Codec components, code resources that provides image compression or decompression services for image data. For historical reasons, these components are sometimes referred to as image compressor components, rather than codec components, as a generic term for both compressors and decompressors. To make it easier for you to create new decompressor components, Apple provides a Base Image Decompressor component. You can use this component to perform most of the services that are common to all decompressors, allowing you to focus on the tasks that are specific to your decompressor.
- Data codecs, which enable you to compress and decompress media data from tracks other than sound and video tracks, such as sprites or 3D models, that are not compressed and decompressed automatically. Data codecs also allow you to compress or decompress arbitrary blocks of data from other sources.
- Image compression dialog components, which provide user interfaces for setting the parameters that control the compression of still images and sequences.
- Image transcoders, which translate compressed image data from one format to another. Image transcoders are typically used when a movie has been compressed in a format for which there is no decompressor on the playback machine, or when an application provides an export function.

Note: This book replaces five previously separate Apple documents: “Image Compression Manager,” “Codec Components,” “Data Codecs,” “Image Compression Dialog,” and “Image Transcoders.”

To get the whole story of QuickTime compression and decompression, read this whole book; or you can read only parts of this book, depending on your immediate task or interest:

- If you are developing an application that works with images (including sequences of images), you should read the first seven chapters (which describe the Image Compression Manager) plus the chapters About Image Compressor Components (page 81) and Using Image Compressor Components (page 87).

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- If you are going to write your own codec components, you should also read Codec Components API (page 105) and About the Base Image Decompressor (page 109).
- If you need to work with compressed data outside of the normal capture, storage, import, export, and playback of sound and video, you should read Using Data Codec Components (page 117).
- If your application creates movies, you should read about image-compression dialog components in the chapters Standard Image Compression Dialog Components (page 119), Working With Standard Image Compression Dialog Components (page 121), and Image Compression Dialog Types and Functions (page 129).
- If you intend to use or create image transcoder components, you need to read the chapters Using Image Transcoder Components (page 139) and Creating Image Transcoder Components (page 143).

Organization of This Document

This book consists of the following chapters:

- The Image Compression Manager (page 15) provides an overview of the Image Compression Manager.
- Extensions to the Image Compression Manager (page 23) describes the extensions to the Image Compression Manager introduced in various releases of QuickTime, including support for ColorSync, asynchronous decompression, timecodes, and compressed fields of video data.
- Image Compression Characteristics (page 27) provides a brief overview of the characteristics of image compression algorithms, including discussions of compression ratios, compression speed, and image quality.
- Compressors Supplied by Apple (page 29) discusses the compressors that Apples supplies with the Image Compression Manager. Included also is a detailed discussion of which types of images are most suitable for which compressors, with illustrative graphs of compression ratios.
- Working with the Image Compression Manager (page 43) which describes how to use the Gestalt Manager to determine what version of the Image Compression Manager is available. Various functions you can use to gather information about the Image Compression Manager and the installed compressor components are also discussed.
- How to Compress and Decompress Sequences of Images (page 55) provides a sample program illustrating the processes used to compress and decompress image sequences.
- ICM Functions, Data Types, and Constants (page 67) describes the various functions, data types, and constants your application can take advantage of in working with the Image Compression Manager.
- About Image Compressor Components (page 81) describes the general characteristics of an image compressor component, including its component type.
- Using Image Compressor Components (page 87) describes what the Image Compression Manager does that affects compressors. It also provides sample code that shows how the compressor components prepare for image compression, and how to compress an image or a horizontal band from an image.
- Codec Components API (page 105) lists the data types, functions, and constants in QuickTime that support image compression and decompression.
- About the Base Image Decompressor (page 109) describes the base image decompressor, an Apple-supplied component that makes it easier for developers to create new decompressors.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- Using Data Codec Components (page 117) describes the features of data codec components and shows how to find and control them. A list of useful codec component functions is provided.
- Standard Image Compression Dialog Components (page 119) introduces the standard image compression dialog component and illustrates the two standard dialog boxes.
- Working With Standard Image Compression Dialog Components (page 121) describes in detail how you can use the standard image compression dialog component.
- Image Compression Dialog Types and Functions (page 129) describes the request types and functions associated with the standard image compression dialog components and an application-defined function.
- Using Image Transcoder Components (page 139) describes what image transcoding is and why it is useful, providing an overview of the relationship between applications, QuickTime services, and transcoder components. This chapter also describes the process of creating an image transcoding sequence for the Image Compression Manager and lists the functions applications use to access transcoding services.
- Creating Image Transcoder Components (page 143) describes when and how to create image transcoder components, and provides a code listing with an example transcoder for component authors.

See Also

The following Apple books cover related aspects of QuickTime programming:

- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.
- *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.
- *QuickTime Media Types and Media Handlers Guide* introduces the idea of QuickTime media handler components and provides details of the video, sound, text, timecode, and tween media handlers.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Image Compression Manager

This chapter describes the QuickTime Image Compression Manager (ICM), which compresses images by invoking **image compressor components** and decompresses images using **image decompressor components.**

The Image Compression Manager enables your application to perform a variety of tasks, including the storage of compressed image data in pictures, compression of image sequences, display of compressed image files, and so on.

Overview of the ICM

The Image Compression Manager provides your application with an interface for compressing and decompressing images and sequences of images that is independent of devices and algorithms.

Uncompressed image data requires a large amount of storage space. Storing a single 640-by-480 pixel image in 32-bit color can require as much as 1.2 MB. Sequences of images, like those that might be contained in a QuickTime movie, demand substantially more storage than single images. This is true even for sequences that consist of fairly small images, because the movie consists of such a large number of those images. Consequently, minimizing the storage requirements for image data is an important consideration for any application that works with images or sequences of images.

The Image Compression Manager compresses images by invoking **image compressor components** and decompresses images using **image decompressor components.** Compressor and decompressor components are code resources that present a standard interface to the Image Compression Manager and provide image-compression and image-decompression services, respectively. The Image Compression Manager receives application requests and coordinates the actions of the appropriate components. The components perform the actual compression and decompression. Compressor and decompressor components are standard components and are managed by the Component Manager.

Because the Image Compression Manager is independent of specific compression algorithms and drivers, it provides a number of advantages to developers of image-compression algorithms. Specifically, compressor and decompressor components can

- present a common application interface for software-based compressors and hardware-based compressors.
- provide several different compressors and compression options, allowing the Image Compression Manager or the application to choose the appropriate tool for a particular situation.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Data That Is Suitable for Compression

One way to represent an image is with a pixel map, which stores a color for every pixel. For most images, however, a pixel map is an inefficient storage format. For example, a pixel map containing a solid black image would contain the color black stored over and over and over again. By compressing the image, some of this redundant information can be eliminated. The compressed image can occupy much less storage than a pixel map and can be decompressed to a pixel map when necessary.

In addition, human perception of visual images exhibits special qualities that can be exploited to further compress image data. Image-compression algorithms take advantage of these properties to reduce the amount of information required to describe an image well enough to allow a person to see it.

A **lossless compression** technique can recreate an exact copy of the original image from the compressed form. Small changes in the image are not objectionable in most applications, however, so most compressors sacrifice some accuracy in order to further decrease the size of the compressed data. However, the compressor carefully chooses the data to omit so that the human visual system compensates for the loss and fools the user into seeing what appears to be the original image.

The Image Compression Manager works only with image data. The Image Compression Manager is primarily useful for compressing pictures that have pixel map images, such as those obtained from scanned still images or digitized video images, or from paint or three-dimensional rendering applications. You do not achieve significant compression treating pictures that are stored as groups of graphics primitives, such as those created by drawing, computer-aided design (CAD), or three-dimensional modeling applications. These applications create images in a compact format that precisely states the characteristics of the objects in the image. In fact, if you were to convert such images to pixel map representations and then compress the resulting image with the Image Compression Manager, you would probably end up with a larger, less precise image than the original. If a picture contains both primitives and pixel map image data (such as text or lines drawn over a painted or digitized image) the Image Compression Manager compresses the pixel map data and leaves the graphics primitives unchanged.

The Image Compression Manager also provides services for compressing and decompressing sequences of images or **frames** (another term for a single visual image in an image sequence). When processing a sequence, compressors may perform **temporal compression,** compressing the sequence by eliminating information that is redundant from one frame to the next. This temporal compression differs from **spatial compression,** which is performed on individual images or frames within a sequence. You may use both techniques on a single sequence.

Compressor components perform temporal compression by comparing the current frame in a sequence with the previous frame. The compressor then stores information about only those pixels that change significantly between the two images. When adjacent images contain substantially similar visual information, as is often the case in movies, temporal compression can significantly reduce the amount of data required to describe the images in the sequence. Your application indicates the desired quality level for the compressed image. The compressor uses this value to govern the extent to which it takes advantage of temporal redundancy between images. There is also a spatial quality level that you can use to control the amount of spatial compression applied to each individual image. Both of these quality values govern the amount of accuracy that is lost in the compressed image.

Note that the Image Compression Manager does not maintain any time information for an image sequence. Rather, the Image Compression Manager maintains the order and content of the images in the sequence while the Movie Toolbox handles all timing considerations.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Storing Images

The Image Compression Manager can compress two kinds of image data: pictures and pixel maps. Pictures may be stored in memory, in a resource, or in a PICT file. Pixel maps are normally stored in a window or offscreen buffer. When compressing an image from a PICT file, the Image Compression Manager provides facilities that allow applications to spool data to and from the disk file, as appropriate to the operation. These application-provided data-loading and data-unloading functions allow arbitrarily large images to be compressed or decompressed without requiring large amounts of memory.

Applications must convert images that are not stored as pictures or pixel maps into one of these formats before compressing them. The Image Compression Manager contains several high-level functions that make it quite easy for applications to work with compressed images that are stored as PICT files, as discussed in the next section.

Working With Pictures

The Image Compression Manager provides a set of functions that allow applications to work easily with compressed pictures stored in version 2 PICT files. These functions constitute a high-level interface to image compression and decompression. Applications that require little control over the compression process may use these functions to display pictures that contain compressed image data.

Existing programs can display (without changes) pictures that contain compressed image data. When the Image Compression Manager is installed on a system, it installs a new `StdPix` graphics function (see Working With the `StdPix` Function (page $@) for more information on the `StdPix` graphics function). This function handles all requests to display compressed images. Whenever an application issues the standard QuickDraw `DrawPicture` routine to display an image that contains compressed image data, the `StdPix` function decompresses the image by invoking the Image Compression Manager. The function then delivers the decompressed image to the application.

The Image Compression Manager also provides a simple mechanism for creating a picture that contains compressed image data. For example, to place an existing compressed image into a picture, your application could open the picture with QuickDraw's `OpenPicture` (or `OpenCPicture`) function and then call the Image Compression Manager's `DecompressImage` function, as if you were going to display the image. The Image Compression Manager places the compressed image and the other data that describe the image into the picture for you.

The Image Compression Manager stores the following information about a compressed picture:

- the image description, which describes the compression format and characteristics of the compressed image data
- the compressed data for the image
- the transfer mode (source copy mode, dither copy mode, and so on)
- the matte pixel map
- the mask region
- the mapping matrix
- the source rectangle of the image

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Image Compression Manager stores this information in the picture as a new PICT opcode (described in the following paragraphs). When an application draws the compressed picture on a Macintosh computer that is running the Image Compression Manager, the `StdPix` function instructs the Image Compression Manager to decompress the image. If an application tries to read a picture file that contains compressed data on a Macintosh that does not have the Image Compression Manager installed, the system ignores the new opcodes and displays a message that indicates that the user needs QuickTime in order to display the compressed image data. The message states "QuickTime and a <Codec Name> decompressor are needed to see this picture".

The Color QuickDraw version 2 picture format includes PICT opcodes for compressed and uncompressed QuickTime images. (An opcode is a hexidecimal number that represents drawing commands and the parameters that affect those drawing commands in a picture.)

The PICT opcodes for compressed and uncompressed QuickTime images are

- opcode $8200, which signals a compressed QuickTime image
- opcode $8201, which signals an uncompressed QuickTime image

Table 1-1 gives an overview of the opcode for QuickTime compressed pictures.

Table 1-1 Fields of the PICT opcode for compressed QuickTime images

Field name	Description	Data size (in bytes)
`Opcode`	Compressed picture data	2
`Size`	Size in bytes of data for this opcode	4
`Version`	Version of this opcode	2
`Matrix`	3 by 3 fixed transformation matrix	36
`MatteSize`	Size of matte data in bytes	4
`MatteRect`	Rectangle for matte data	8
`Mode`	Transfer mode	2
`SrcRect`	Rectangle for source	8
`Accuracy`	Preferred accuracy	4
`MaskSize`	Size of mask region in bytes	4

Warning: Do not attempt to read opcodes directly. For compatibility reasons, use Toolbox routines to access this information.

The `MaskSize` field of opcode $8200 is followed by five variable fields:

- The matte image description, which contains the image description structure for the matte. The variable size is specified in the first long integer in the opcode. This field is not included if the `MatteSize` field is 0.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- The matte data, which contains the compressed data for the matte. The size of this field is defined by the `MatteSize` field listed in Table 1-1. This field is not included if the `MatteSize` field is 0.
- The mask region, which contains the region for masking. The size of this variable is defined by the `MaskSize` field listed in Table 1-1. This field is not included if the `MaskSize` field is 0.
- The image description structure for this data. The size of this variable is specified in the first long integer in the `idSize` field of this image description.
- The image data, which contains the compressed data for the image. The size of the image data is specified in the image description's `dataSize` field.

See The Image Description Structure (page 51) for details on the `idSize` and `dataSize` fields.

Table 1-2 provides an overview of the structure of uncompressed QuickTime images.

Table 1-2 Fields of the PICT opcode for uncompressed QuickTime images

Field name	Description	Data size (in bytes)
`Opcode`	Uncompressed picture data	2
`Size`	Size in bytes of data for this opcode	4
`Version`	Version of this opcode	2
`Matrix`	3 by 3 fixed transformation matrix	36
`MatteSize`	Size of matte data in bytes	4
`MatteRect`	Rectangle for matte data	8

The `MatteRect` field of opcode $8201 is followed by three variable fields and a subopcode:

- The matte image description, which contains the image description structure for the matte. The size of this variable is specified in the first long integer in this opcode. This field is not included if the `MatteSize` field is 0.
- The matte data, which contains information for the matte. The size of this variable is defined by the `MatteSize` field.
- A subopcode (2 bytes in length) which describes the image and mask and is entirely within the other opcode. Its size is included in the size for the main opcode; hence it is not included if the QuickTime opcode is skipped. This subopcode can be either $98, $99, $9A, or $9B.
- The data for the subopcode variable which contains information for the image.

Understanding Compressor Components

This section discusses key attributes of compressor components and the functional interfaces these components must support. (**Compressor components** here refers to both image compressor components and image decompressor components.) This information is intended for developers of compressor components. Application developers do not need to be familiar with this material to use the Image Compression Manager.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

A compressor component is a code resource that provides image compression or decompression services for image data. These components may also utilize additional hardware to provide their services. Compressor components are registered by the Component Manager, and they present a standard set of function interfaces to the Image Compression Manager. A compressor can be a system-wide resource, or it can be local to a particular application.

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

The Image Compression Manager knows about two types of compressor components. Components that can compress image data carry a component type (described by the `compressorComponentType` data type) of `'imco'` and are referred to as *compressors*. Components that can decompress images have a component type (described by the `decompressorComponentType` data type) of `'imdc'` and are called *decompressors*. The value of the component subtype indicates the compression algorithm supported by the component. All compressor components with the same subtype must be able to handle the same format of compressed data. During decompression a component should handle all variations of the data specified for a subtype. Conversely, while compressing an image a compressor must not produce data that decompressors of the same subtype cannot handle during decompression.

The Image Compression Manager defines four callback functions that may be provided to compressors or decompressors by applications. A **callback function** is an application-defined function that is invoked at a specified time or based on specified criteria. These callback functions are data-loading functions, data-unloading functions, completion functions, and progress functions. Data-loading functions and data-unloading functions support spooling of compressed data. Completion functions allow 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. For more information about these callback functions, see Application-Defined Functions (page 47).

Banding and Extending Images

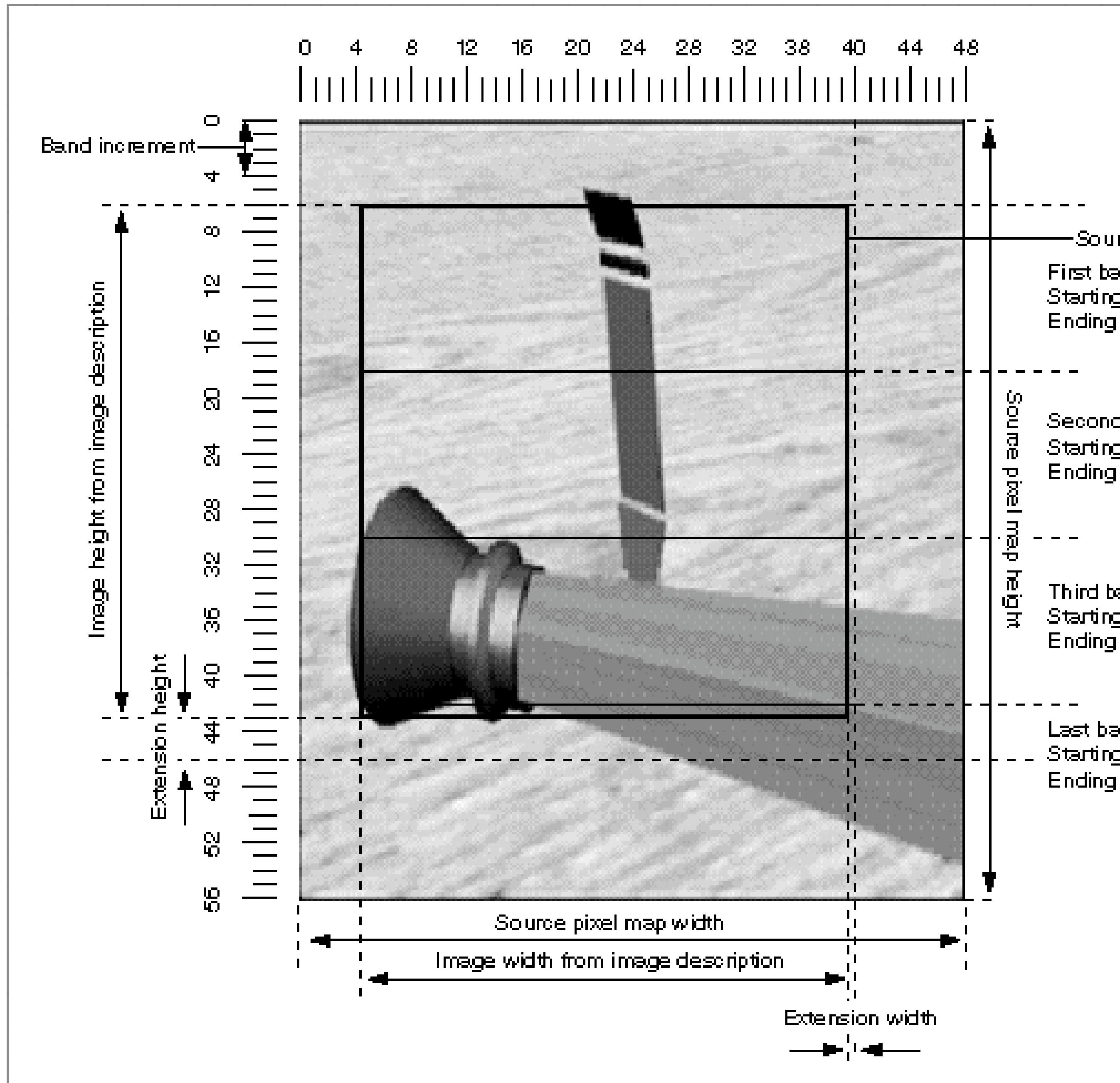
Occasionally a compressor component may not be able to accommodate the destination rectangle for an image decompression or the source for an image-compression operation. This situation may result from compressors that are optimized to work at certain depths or that cannot perform scaling, translation, dithering, or masking during decompression. In such circumstances the Image Compression Manager allocates a temporary buffer that is acceptable to the compressor component and breaks the image up to fit into that new buffer. Since there often is not enough memory to allocate a buffer to hold the entire image, the Image Compression Manager may allocate one that holds a band of the image. A **band** is one horizontal piece of the image. Its height is some portion of the desired image height (before scaling or rotation), and it is at least as wide as the desired image.

The height of the band is determined both by the amount of memory available and the block size of the compressor component. The block size of a compressor is the natural size at which it handles images, and it is peculiar to the image-compression algorithm. The block size for the photo compressor is usually 16 pixels by 16 pixels, for example. Usually the block width and height are equal, but this is not always the case. The minimum height of a band is one strip of blocks. A strip is defined to be a part of an image that is as high as the block height (for the compressor in question) and as wide as the band. The width of a band is either the width of the desired unscaled image, or that width increased by an extension.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 1-1 shows the measurements of several image bands.

Figure 1-1 Image bands and their measurements



Some compressors can only handle images with dimensions that are a multiple of their block size. If the desired image does not comply with this restriction in either dimension, the Image Compression Manager extends the band on the right side and bottom by the amount required to meet the needs of the compressor. During compression, the compressor fills the extended region with the same pixel value as the pixels adjacent to the extension. During decompression, the Image Compression Manager writes only the pixels that are part of the source image. The extended portion remains only in the offscreen buffer.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Fast Dithering

QuickDraw provides a means of displaying images with high color resolution in pixel maps or on screens with lower color resolution. By dithering the destination image, QuickDraw fools your eyes into seeing colors that are not actually available on the display screen. Unfortunately, the error-diffusion technique used by QuickDraw takes longer than just drawing pixels by directly looking them up in a color table. The drawing delays imposed by standard dithering are unacceptable when working with movies.

To alleviate this problem, Apple has developed a technique that allows faster dithering to destinations that use 8 bits per pixel. Fast dithering uses lookup tables created by the Image Compression Manager. All the decompressors supplied by Apple can use fast dithering.

Apple decompressors use fast dithering when copying from image band buffers to 8-bit destinations. If the accuracy for decompression is above normal, then the decompressors use true error diffusion rather than fast dithering. Note that video sequences are normally displayed at normal or low accuracy so that you can obtain maximum display speed during decompression.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Extensions to the Image Compression Manager

This chapter describes the extensions to the Image Compression Manager introduced in various releases of QuickTime, including support for ColorSync, asynchronous decompression, timecodes, and compressed fields of video data.

ColorSync Support

ColorSync is a system extension that provides a platform for consistent color reproduction between widely varying output devices. ColorSync color matching capability was added to the Image Compression Manager picture drawing functions in QuickTime. You can accurately reproduce color images (not movies) with the `DrawPicture` functions by setting the `useColorMatching` flag in the flags parameter to these functions.

```
enum {
    useColorMatching    = 4
};
```

Asynchronous Decompression

QuickTime introduced the concept of scheduled asynchronous decompression operations. Decompressor components can allow applications to queue decompression operations and specify when those operations should take place. The Image Compression Manager provides a function, `DecompressSequenceFrameWhen`, that allows applications to schedule an asynchronous decompression operation.

Timecode Support

The Image Compression Manager and compressor components have been enhanced to support timecode information. The Image Compression Manager function `SetDSequenceTimeCode` allows you to set the timecode value for a frame that is to be decompressed.

Data Source Support

QuickTime introduced support for an arbitrary number of sources of data for an image sequence. This functionality forms the basis for dynamically modifying parameters to a decompressor. It also allows for codecs to act as special effects components, providing filtering and transition type effects. A client can attach

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

an arbitrary number of additional inputs to the codec. It is up to the particular codec to determine whether to use each input and how to interpret the input. For example, an 8-bit gray image could be interpreted as a blend mask or as a replacement for one of the RGB data planes.

To create a new data source, use the function `CDSequenceNewDataSource`.

Working with Alpha Channels

QuickTime supports compressing and storing images with alpha channels. QuickTime supports use of the alpha channel when displaying images. Display of alpha channels is supported only for images with an uncompressed bit depth of 32 bits or greater. For 32-bit ARGB images, for example, the high byte of each pixel contains the alpha channel. For 64-bit ARGB (`k64ARGBCodecType`), there is a 16-bit alpha field for each pixel, in addition to 16-bit fields for red, green, and blue. The alpha channel can be interpreted in one of three ways:

- straight alpha
- pre-multiplied with white
- pre-multiplied with black

QuickTime uses the alpha channel to define how an image is to be combined with the image that is already present at the location to which it will be drawn. This is similar to how QuickDraw's blend mode works. To combine an image containing an alpha channel with another image, you specify how the alpha channel should be interpreted by specifying one of the new alpha channel graphics modes defined by QuickTime.

Straight alpha means that the color components of each pixel should be combined with the corresponding background pixel based on the value contained in the alpha channel. For example, if the alpha value is 0, only the background pixel will appear. If the alpha value is 255, only the foreground pixel will appear. If the alpha value is 127, then (127/255) of the foreground pixel will be blended with (128/255) of the background pixel to create the resulting pixel, and so on.

Pre-multiplied with white means that the color components of each pixel have already been blended with a white pixel, based on their alpha channel value. Effectively, this means that the image has already been combined with a white background. To combine the image with a different background color, QuickTime must first remove the white from each pixel and then blend the image with the actual background pixels. Images are often pre-multipled with white as this reduces the appearance of jagged edges around objects.

Pre-multipled with black is the same as pre-multipled with white, except the background color that the image has been blended with is black instead of white.

Note: Although you pass these new alpha channel graphics modes to QuickTime in the same way as you would traditional QuickDraw transfer modes, these modes are not supported by QuickDraw and will cause unpredictable results if passed to QuickDraw routines.

The Image Compression Manager defines the following constants for specifying alpha channel graphics modes:

```
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
```

© 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    graphicsModePreBlackAlpha        = 258
    graphicsModeStraightAlphaBlend   = 260
};
```

The `graphicsModeStraightAlphaImage` Compression Manager, `graphicsModePreWhiteAlphaImage` Compression Manager, and `graphicsModePreBlackAlpha` graphics modes cause QuickTime to draw the image interpreting the alpha channel as specified. The graphics mode `graphicsModeStraightAlphaBlend` causes QuickTime to interpret the alpha channel as a straight alpha channel, but when it draws, combines the pixels together and applies the `opColor` supplied with the graphics mode to the alpha channel. This provides an easy way to combine images using both an alpha channel and a blend level. This can be useful when compositing 3D rendered images over video.

To draw a compressed image containing an alpha channel, that image must be compressed using an image-compression format that is capable of storing the alpha channel information. The Animation, Planar RGB and None compressors store alpha channel data in the "Millions of Colors" + (32-bit) mode.

You use the `MediaSetGraphicsMode` function to set a movie track to use an alpha channel graphics mode. You use the `SetDSequenceTransferMode` function to set an image sequence to use an alpha channel graphics mode.

Working With Video Fields

QuickTime introduced support for working directly with fields of interlaced video, such as those created by some motion JPEG compressors.

Because video processing applications sometimes need to perform operations on individual fields (for example, reversing them or combining one field of a frame with a field from another frame), QuickTime now provides a method for accessing the individual fields without having to decompress them first. Previously such operations required decompressing each frame, copying the appropriate fields, and then recompressing. This was a time consuming process that could result in a loss of image quality due to the decompression and recompression of the video data.

Three functions (`ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd`) allow an application to request that field operations be performed directly on the compressed data. These functions accept one or two compressed images as input and create a single compressed image on output.

The Apple Component Video and Motion JPEG compressors support image field functions in QuickTime. See the description of the `ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd` functions in the QuickTime API Reference for information on how to process image fields in your application.

Packetization Information

QuickTime video compressors are increasingly being used for videoconferencing applications. Image data from a compressor is typically split into network-packet-sized pieces, transmitted through a packet-based protocol (such as UDP or DDP), and reassembled into a frame by the receiver(s). Typically, a lost packet causes

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

an entire frame to be dropped; without all the data for a given frame, the decompressor cannot decode the image. When the loss of one packet forces others to be unusable, the loss rate is effectively multiplied by a large factor.

Some compression methods, however, such as H.261, can divide a compressed image into pieces which can be decoded independently. Some videoconferencing protocols, such as the Internet's Real Time Protocol (RTP, RFC#1889), specify that data compressed using H.261 must be packetized into independently decodable chunks. While RTP demands this packetization information from the compressor, other protocols, such as QuickTime Conferencing's MovieTalk protocol, can optionally use this information to effectively reduce loss rates.

QuickTime added four functions to support packetization: `SetCSequencePreferredPacketSize`, `SGSetPreferredPacketSize`, `SGGetPreferredPacketSize`, and `VDSetPreferredPacketSize`. In addition, the `CodecCompressParams` structure includes a field, `preferredPacketSizeInBytes`.

For application developers, the important function is `SGSetPreferredPacketSize`.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Image Compression Characteristics

This chapter a brief overview of the characteristics of image compression algorithms, including discussions of compression ratios, compression speed, and image quality.

There are three main characteristics by which you can judge image-compression algorithms: compression ratio, compression speed, and image quality. You can use these characteristics to determine the suitability of a given compression algorithm to your application. The following paragraphs discuss each of these attributes in more detail.

Compression Ratio

The compression ratio is equal to the size of the original image divided by the size of the compressed image. This ratio gives an indication of how much compression is achieved for a particular image.

The compression ratio achieved usually indicates the picture quality. Generally, the higher the compression ratio, the poorer the quality of the resulting image. The trade-off between compression ratio and picture quality is an important one to consider when compressing images.

Furthermore, some compression schemes produce compression ratios that are highly dependent on the image content. This aspect of compression is called **data dependency.** Using an algorithm with a high degree of data dependency, an image of a crowd at a football game (which contains a lot of detail) may produce a very small compression ratio, whereas an image of a blue sky (which consists mostly of constant colors and intensities) may produce a very high compression ratio.

Compression Speed

Compression time and decompression time are defined as the amount of time required to compress and decompress a picture, respectively. Their value depends on the following considerations:

- the complexity of the compression algorithm
- the efficiency of the software or hardware implementation of the algorithm
- the speed of the utilized processor or auxiliary hardware

Generally, the faster that both operations can be performed, the better. Fast compression time increases the speed with which material can be created. Fast decompression time increases the speed with which the user can display and interact with images.

 © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Image Quality

Image quality describes the fidelity with which an image-compression scheme recreates the source image data. Compression schemes can be characterized as being either lossy or lossless. Lossless schemes preserve all of the original data. **Lossy compression** does not preserve the data precisely; image data is lost, and it cannot be recovered after compression. Most lossy schemes try to compress the data as much as possible, without decreasing the image quality in a noticeable way. Some schemes may be either lossy or lossless, depending upon the quality level desired by the user.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Compressors Supplied by Apple

Apple supplies six image-compression algorithms with the Image Compression Manager. This chapter discusses each of these compressors and identifies their strengths and weaknesses in light of the compression characteristics just discussed. You can use this discussion as a guideline for choosing a compression algorithm for your specific situation. All the compressors support both temporal and spatial compression except for the Photo and Raw Compressors, which support only spatial compression.

The Photo Compressor

The Photo Compressor implements the **Joint Photographic Experts Group** (JPEG) algorithm for image compression. JPEG is an international standard for compressing still images. The version of JPEG supplied with QuickTime complies with the baseline International Standards Organization (ISO) standard bitstream, version 9R9.

The Photo Compressor performs best on images that vary smoothly or that do not have a large percentage of their areas devoted to edges or other types of sharp detail. This is the case for most natural (that is, nonsynthetic) images. In practice, you will find that compression ratios are highly dependent on source images, but they generally range from 5:1 to 50:1 at 24 bits per pixel, with good picture quality resulting from compression ratios between 10:1 and 20:1.

Picture quality is generally very good to excellent and is often good enough for use in demanding desktop publishing applications. Very high-resolution images obtained through the use of 24-bit color scanners would best be compressed using the Photo Compressor. This compressor is good for 8-bit grayscale images; it is not well suited to 1-bit images or non-natural images that usually have high contrast.

The Video Compressor

The Video Compressor employs an image-compression method developed by Apple. This method was designed to permit very fast decompression times while maintaining reasonably good picture quality. This algorithm's rapid decompression allows applications to display color images or drawings at interactive speeds. This algorithm is best suited for use with sequences of video data.

The Video Compressor is better suited to digitized video content rather than synthetically generated images. This compressor supports both spatial and temporal compression. If you use only spatial compression, you may obtain compression ratios from 5:1 to 8:1 with reasonably good quality at 24-bit pixel depths. If you use both spatial and temporal compression, the compression ratio range extends from 5:1 to 25:1.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Compact Video Compressor

The Compact Video Compressor is best suited to compressing 16-bit and 24-bit video sequences. It employs a lossy algorithm developed by Apple that is highly asymmetrical. In other words, it takes significantly longer to compress a frame than it does to decompress that frame.

Compared to the Video Compressor, the Compact Video Compressor obtains higher compression ratios, better image quality, and faster playback speeds. The Compact Video Compressor can constrain data rates to user-definable levels. This is particularly important when compressing material for playback from CD-ROM discs.

For best quality results, the Compact Video Compressor should be used on raw source data that has not been compressed with a highly lossy compressor, such as the Video Compressor.

The Animation Compressor

The Animation Compressor employs a compression algorithm developed by Apple. This technique is best suited to animation and computer-generated video content. In addition, the Animation Compressor can be used to compress sequences of screen images, such as might be generated for a training application.

The Animation Compressor stores images in run-length encoded format, and it can work in either a lossy or a lossless mode. The lossless mode maintains picture content precisely, storing an animation as a series of run-length encoded images. The lossy mode loses some image quality.

The Animation Compressor's performance and achieved compression ratios are highly dependent on the type of images in a scene. The Animation Compressor is very sensitive to picture changes, and it works best on a clean image that has been generated synthetically. Images captured from videotape generally have considerable visual noise, which can corrupt the inherent similarity of the pixels and make it more difficult for the Animation Compressor to achieve good compression. This compressor works at all pixel depths.

The Graphics Compressor

The Graphics Compressor employs a compression algorithm developed by Apple. This compressor is best suited to 8-bit still images and image sequences in applications where compression ratio is more important than decompression speed.

The Graphics Compressor is a good alternative to the Animation Compressor whenever performance is less important than compression ratio. In general, the Graphics Compressor generates a compressed image that is one-half the size of the same image compressed by the Animation Compressor. However, the Graphics Compressor can decompress the image at only half the speed of the Animation Compressor. Therefore, you should consider using the Graphics Compressor with relatively slow storage devices, such as CD-ROM discs. In these circumstances, the Graphics Compressor has sufficient time to decompress the image or image sequence.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Raw Compressor

The Raw Compressor can reduce image storage requirements by converting an image from one pixel depth to another. For example, converting a 32-bit image to 16-bit format achieves a 2:1 compression ratio. The Raw Compressor can also convert a 32-bit image to 24-bit format by dropping the pad byte. This achieves a 4:3 compression with no loss of quality. The Raw Compressor accomplishes this conversion quickly, and the resulting image retains excellent image quality in most cases.

The Image Compression Manager often uses the Raw Compressor to extend the capabilities of other compressors. For example, the Photo Compressor works directly with only 32-bit color images and 8-bit grayscale images. For color images, the Image Compression Manager uses the Raw Compressor to convert the pixel depth of the original image to 32-bit color or to convert the 32-bit decompressed image to another pixel depth for display.

Image quality can deteriorate when the pixel depth is reduced; however, this technique is generally lossless when converting from a lower pixel depth to a higher depth. With 1, 2, 4, 8, and 24-bit images, the Raw Compressor allows colors to be mapped through a color table.

Note that the resulting image may be larger than the corresponding pixel image in PICT format, because QuickDraw stores PICT images in a run-length encoded format.

Note: These uncompressed QuickTime-specific PICT images cannot be used without QuickTime.

Performance figures for the Raw Compressor are dependent upon the source and destination pixel depths. (The Raw Compressor is signified by the None option in the standard compression dialog box.)

Types of Images Suitable for Different Compressors

This section presents a series of graphs that indicate the amount of compression you can obtain when you compress still images with the Apple-supplied QuickTime compressors.

Note: Since some compressors make use of temporal compression, these results cannot be used to directly infer results for compressing image sequences (as in QuickTime movies).

The different compressors take advantage of different properties of an image to achieve their compression; hence, the type of image being compressed significantly affects the amount of compression achieved, as well as the fidelity of the compressed image to the original.

For this comparison, three images that represent three classes of digital images are used. Figure 4-1 provides a photographic image scanned from a photographic slide. This is a natural image and contains no computer-synthesized characters or graphics elements.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-1 24-bit photographic image

Figure 4-2 shows a full-color image created by a three-dimensional graphics rendering program. It does not contain the detail of a natural image, but it is a full-color image that needs significantly more than 256 colors to portray it accurately. It is possible to create such an image with a full-color paint or drawing program as well as from a three-dimensional rendering program. Note also that, if an image created by these means has enough detail, it becomes more like a photographic image. Likewise, a natural image with some overlaid graphics or text may fit more closely into this category than the photographic category depending on the proportions of each type of imagery.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-2 A 24-bit synthetic image

Figure 4-3 is an example of a nondithered simple graphic image with fewer than 256 colors. The image is adequately represented by 8 bits per pixel. This image is also special in that it has large horizontal areas that are all of a single color, which is an important characteristic exploited by several compression algorithms, including the normal PICT packing used by QuickDraw.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-3 An 8-bit graphic image

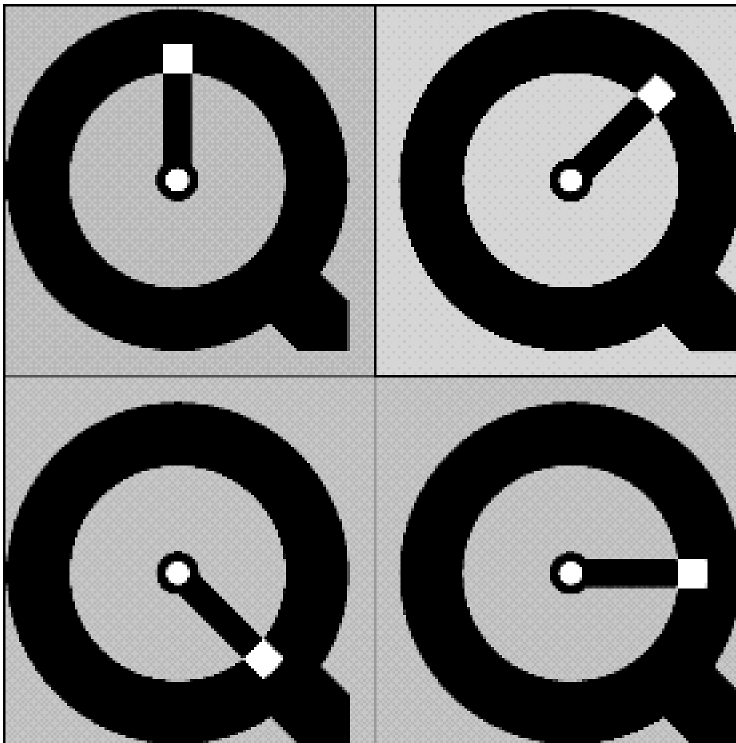


Figure 4-4 is a natural photographic image dithered to 8 bits per pixel.

Figure 4-4 An 8-bit photographic image

© 2005, 2006 Apple Computer, Inc. All Rights Reserved.

All of the graphs show the compressed data size (in kilobytes) versus the quality of an image at minimum, low, normal, high, and maximum compression settings. The Raw Compressor is included to show the size of the image in raw pixels. The Raw Compressor is not useful for storing still images, since it does not even use the simple packing technique used by QuickDraw (notice that the 24-bit raw format is larger than the uncompressed PICT file).

Figure 4-5 provides a graph that compares compressor performance for the photographic image shown in Figure 4-1. The best compression is obtained by the Compact Video Compressor. The Photo Compressor performs as well as the Compact Video Compressor at minimum, low, and normal compression settings, but does not perform as well at high and maximum settings. However, as you might expect, the Photo Compressor retains the best image quality. The Graphics Compressor stores the image at a smaller size than the highest quality setting of the Photo Compressor, but only stores 256 colors, which significantly degrades the quality of the image. The Video Compressor does almost as well as the Photo Compressor, but the image quality is lower, because of compression artifacts and reduced color resolution. The Animation Compressor retains the color resolution and detail of the image when storing millions of colors and the detail when storing thousands of colors, but it does not achieve nearly as much compression as the other compressors.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-5 Compressor performance for a 921 KB, 24-bit, photographic image

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The graph in Figure 4-6 compares compressor performance for the full-color, computer-synthesized image shown in Figure 4-2. The Compact Video Compressor again achieves the best overall compression, followed by the Photo and Video Compressors. Again the Graphics Compressor cannot accurately represent all of the colors of the image and is not suitable for use on this type of image. With this image, the Animation Compressor does better than it did with the natural image, and it may be suitable if space constraints are not as important as speed constraints. Because computer-generated images tend to have smoother color gradations than natural images, the loss of color resolution with the Video Compressor and the 16-bit Raw and Animation Compressors is more apparent.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-6 Compressor performance for a 502 KB, 24-bit, synthetic image

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-7 compares compressor performance for the simple graphic image shown in Figure 4-3. The Graphics Compressor is the only reasonable choice. Not only does it produce the best compression, but also it stores the image without losing any of the image's detail, since there are fewer than 256 colors in the source image. The Photo and Compact Video Compressors get some compression, but do not store the image as accurately as the Graphics Compressor. The Video Compressor stores the image even less accurately and does not compress the image well at all. The Animation Compressor also does not store the image with complete accuracy at 16 or even 24 bits per pixel, and the resulting files are much larger than the uncompressed PICT. Although the 8-bit Animation Compressor does store the image accurately, it only achieves half as much compression as the Graphics Compressor and its file is also larger than the original PICT.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-7 Compressor performance for a 30 KB, 8-bit, graphic image

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The graph in Figure 4-8 compares performance for the 8-bit, dithered, photographic image shown in Figure 4-4. The best results are obtained by the Compact Video Compressor. The rest of the results are almost the same as for the full-color, natural image shown in Figure 4-5, but this time the Graphics Compressor stores the image exactly, since it had only 256 colors to start with. The other compressors do almost as well as they did for the full-color, natural image, but the compression for all of them is a bit worse, due to the added artifacts introduced when the image was converted to 8 bits per pixel. The 16-bit and 24-bit versions of the Animation Compressor do not make sense for this image, since their results are always larger than the original PICT. The Photo and Video Compressors still do well on this image, but they do lose some detail that the Graphics Compressor retains. The losses are minor, however, and the sizes approach the size of the Graphics Compressor's image only at high-quality settings, where the losses are negligible.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 4-8 Compressor performance for a 302 KB, 8-bit, dithered, photographic image

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Working with the Image Compression Manager

This chapter describes how to use the Gestalt Manager to determine what version of the Image Compression Manager is available. Included is a brief discussion of the information about compressors and compressed data that can be obtained from the Image Compression Manager.

The chapter also describes the various functions you can use to

- gather information about the Image Compression Manager and the installed compressor components
- collect information about compressed images and images that are about to be compressed
- manipulate the parameters that control sequence decompression and to get information about memory that the decompressor has allocated

Getting Information About Compressors and Compressed Data

Use the Gestalt environmental selector `gestaltCompressionMgr` to determine whether the Image Compression Manager is available. Gestalt returns a 32-bit value indicating the version of the Image Compression Manager that is installed. This return value is formatted in the same way as the value returned by the `CodecManagerVersion` function, and it contains the version number specified as an integer value.

```
#define gestaltCompressionMgr 'icmp'
```

The Image Compression Manager provides a number of functions that allow your application to obtain information about the facilities available for image compression or about compressed images. Your application may use some of these functions to select a specific compressor or decompressor for a given operation or to determine how much memory to allocate to receive a decompressed image. In addition, your application may use some of these functions to determine the capabilities of the components that are available on the user's computer system. You can then condition the options your program makes available to the user based on the user's system configuration.

Getting Information About Compressor Components

This section describes the functions that allow your application to gather information about the Image Compression Manager and the installed compressor components.

You can use the `CodecManagerVersion` function to retrieve the version number associated with the Image Compression Manager that is installed on a particular computer.

You can use the `FindCodec`, `GetCodecInfo`, and `GetCodecNameList` functions to locate and retrieve information about the compressor components that are available on a computer.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Getting Information About Compressed Data

This section describes the functions that enable your application to collect information about compressed images and images that are about to be compressed. Your application may use some of these functions in preparation for compressing or decompressing an image or sequence.

You can use the `GetCompressionTime` function to determine how long it will take for a compressor to compress a specified image. Similarly, you can use the `GetMaxCompressionSize` function to find out how large the compressed image may be after the compression operation.

You can use the `GetCompressedImageSize` to determine the size of a compressed image that does not have a complete image description.

The `GetSimilarity` function allows you to determine how similar two images are. This information is useful when you are performing temporal compression on an image sequence.

Compressing Images

The Image Compression Manager provides a rich set of functions that allow applications to compress images. Some of these functions present a straightforward interface that is suitable for applications that need little control over the compression operation. Others permit applications to control the parameters that govern the compression operation.

This section describes the basic steps that your application follows when compressing a single frame of image data. Following this discussion, Listing 5-1 shows a sample function that compresses an image.

First, determine the parameters for the compression operation. Typically, the user specifies these parameters in a user dialog box you may supply via the standard compression dialog component. Your application may choose to give the user the ability to specify such parameters as the compression algorithm, image quality, and so on.

Your application may give the user the option to specify a compression algorithm based on an important performance characteristic. For example, the user may be most concerned with size, speed, or quality. The Image Compression Manager allows your application to choose the compressor component that meets the specified criterion.

To determine the maximum size of the resulting compressed image, your application should then call the Image Compression Manager's `GetMaxCompressionSize` function. You provide the specified compression parameters to this function. In response, the Image Compression Manager invokes the appropriate compressor component to determine the maximum number of bytes required to store the compressed image. Your application should then reserve sufficient memory to accommodate the compressed image or use a data-unloading function to spool the compressed data to disk (see Spooling Compressed Data (page 46) for more information about data-unloading functions).

Once the user has specified the compression parameters and your application has established an appropriate environment for the operation, call the `CompressImage` (or `FCompressImage`) function to compress the image. Use the `CompressImage` function if your application does not need to control all the parameters governing compression. If your application needs access to other compression parameters, use the `FCompressImage` function.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Image Compression Manager manages the compression operation and invokes the appropriate compressor. The manager returns the compressed image and its associated image description structure to your application. Note that the image description structure contains a field indicating the size of the resulting image.

Note: You should use the standard compression dialog component to set up the parameters for compression.

Listing 5-1 Compressing and decompressing an image

```
#include <Types.h>
#include <Traps.h>
#include <Memory.h>
#include <Errors.h>
#include <FixMath.h>
#include "Movies.h"
#include "ImageCompression.h"
#include "StdCompression.h"

#define kMgrChoose 0
PicHandle GetQTCompressedPict (PixMapHandle myPixMap);

PicHandle GetQTCompressedPict( PixMapHandle myPixMap )
{
    long                            maxCompressedSize = 0;
    Handle                          compressedDataH = nil;
    Ptr                             compressedDataP;
    ImageDescriptionHandle          imageDescH = nil;
    OSErr                           theErr;
    PicHandle                       myPic = nil;
    Rect                            bounds = (**myPixMap).bounds;
    CodecType                       theCodecType = 'jpeg';
    CodecComponent                  theCodec = (CodecComponent)anyCodec;
    CodecQ                          spatialQuality = codecNormalQuality;
    short                           depth = 0;/* let ICM choose depth */

    theErr = GetMaxCompressionSize( myPixMap, &bounds, depth,
                                            spatialQuality, theCodecType,
                                            (CompressorComponent)theCodec,
                                             &maxCompressedSize);

    if ( theErr ) return nil;

    imageDescH = (ImageDescriptionHandle)NewHandle(4);
    compressedDataH = NewHandle(maxCompressedSize);
    if ( compressedDataH != nil && imageDescH != nil )
    {
        MoveHHi(compressedDataH);
        HLock(compressedDataH);
        compressedDataP = StripAddress(*compressedDataH);

        theErr = CompressImage( myPixMap,
                        &bounds,
                        spatialQuality,
                        theCodecType,
                        imageDescH,
                        compressedDataP);
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
        if ( theErr == noErr )
        {
            ClipRect(&bounds);
            myPic = OpenPicture(&bounds);
            theErr = DecompressImage( compressedDataP,
                                      imageDescH,
                                      myPixMap,
                                      &bounds,
                                      &bounds,
                                      srcCopy,
                                      nil );
            ClosePicture();
        }
        if ( theErr
                || GetHandleSize((Handle)myPic) == sizeof(Picture) )
        {
            KillPicture(myPic);
            myPic = nil;
        }
    }
    if (imageDescH) DisposeHandle( (Handle)imageDescH);
    if (compressedDataH) DisposeHandle( compressedDataH);
    return myPic;
}
```

Spooling Compressed Data

During compression and decompression operations it may be necessary to spool the image data to or from storage other than computer memory. If your application uses the Image Compression Manager functions that handle picture files, the Image Compression Manager manages this spooling for you. However, if you use the functions that work with pixel maps or sequences and your application cannot store the image data in memory, it is your application's responsibility to spool the data.

The Image Compression Manager provides a mechanism that allows the compressors and decompressors to invoke spooling functions provided by your application. There are two kinds of data-spooling functions: data-loading functions and data-unloading functions. Decompressors call data-loading functions during image decompression. The data-loading function is responsible for providing compressed image data to the decompressor. The decompressor then decompresses the data and writes the resulting image to the appropriate location. See Application-Defined Functions (page 47) for a detailed description of the calling sequence used by the decompressor component when it invokes your data-loading function.

Compressors call data-unloading functions during image compression. The data-unloading function must remove the compressed image data from memory. The compressor can then compress more of the image and write the compressed image data into the available buffer space. See Application-Defined Functions (page 47) for a detailed description of the calling sequence used by the compressor component when it invokes your data-unloading function.

When compressing sequences, your application assigns a data-unloading function by calling the `SetCSequenceFlushProc` function. When decompressing sequences, you assign a data-loading function by calling the `SetDSequenceDataProc` function.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

When your application assigns a spooling function to an image or sequence operation, you must also specify a data buffer and the size of that buffer. The `codecMinimumDataSize` value specifies the smallest data buffer you may allocate for image data spooling.

```
#define codecMinimumDataSize 32768  /* minimum data size */
```

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.

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. 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.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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. 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 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.

Working With Images

This section discusses the functions that allow your application to compress and decompress single-frame images stored as pixel maps (of data type `PixMap`). See Working With Sequences (page 49) for information on compressing and decompressing sequences of images. See Working With Pictures and PICT Files (page 49) for information on compressing and manipulating single-frame images stored as pictures or picture files (in PICT format).

The Image Compression Manager provides two sets of functions for compressing and decompressing images. If you do not need to assert a lot of control over the compression operation, you can use the `CompressImage` and `DecompressImage` functions to work with compressed images. If you need more control over the compression parameters, you can use the `FCompressImage` and `FDecompressImage` functions.

You can convert a compressed image from one compression format to another by calling the `ConvertImage` function.

You can alter the spatial characteristics of a compressed image by calling the `TrimImage` function.

You can work with an image's color table with the `SetImageDescriptionCTable` and `GetImageDescriptionCTable` functions.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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 (page 47) for information on functions that affect sequence compression. See Changing Sequence-Decompression Parameters (page 48) for information on functions that affect sequence decompression.

Working With Pictures and PICT Files

This section describes the functions that let your application compress and decompress single-frame images stored as pictures and PICT files. See Working With Images (page 48) for information on compressing and manipulating single-frame images stored as pixel map structures. See Working With Sequences (page 49) for information on compressing and decompressing sequences of images.

As with image compression, the Image Compression Manager provides two sets of functions for working with compressed pictures. If you do not need to control the compression parameters, use the `CompressPicture` or `CompressPictureFile` functions. If you need more control over the operation, use the `FCompressPicture` or `FCompressPictureFile` functions.

The Image Compression Manager automatically expands compressed pictures when you display them. Use the `DrawPictureFile` function to display the contents of a picture file. If you want to alter the spatial characteristics of the image, use the `DrawTrimmedPicture` or `DrawTrimmedPictureFile` functions.

You can work with an image's control information by calling the `GetPictureFileHeader` function.

Decompressing Images

Working With Pictures (page $@) discusses how applications can display compressed images that are stored as pictures by calling the `DrawPicture` function. The Image Compression Manager also provides functions that allow your application to display single-frame compressed images. As with image compression, your application can choose to specify all the parameters that govern the operation, or it can leave many of these choices to the Image Compression Manager.

This section describes the steps your application must follow to decompress an image into a pixel map.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

First, your application determines where to display the decompressed image. Your application must specify the destination graphics port to the Image Compression Manager. In addition, you may indicate that only a portion of the source image is to be displayed. You describe the desired portion of the image by specifying a rectangle in the coordinate system of the source image. You can determine the size of the source image by examining the image description structure associated with the image (see The Image Description Structure (page 51) for more information about image description structures).

Your application may also specify that the image is to be mapped into the destination graphics port. The Image Compression Manager provides two mechanisms for mapping images during decompression. The `DecompressImage` function accepts a second rectangle as a parameter. During decompression the Image Compression Manager maps the desired image to the destination rectangle, scaling the resulting image as appropriate to fit the destination rectangle. The `FDecompressImage` function allows your application to specify a mapping matrix for the operation. Currently, the Image Compression Manager supports only scaling and translation matrix operations.

Your application can invoke further effects by specifying a mask region or blend matte for the image. Mask regions and mattes control which pixels in the source image are drawn to the destination. **Mask regions** define the part of the source image that is displayed. During decompression the Image Compression Manager displays only those pixels in the source image that correspond to bits in the mask that are set to 1. Mask regions must be defined in the destination coordinate system.

Blend mattes contain several bits per pixel and are defined in the coordinate system of the source image. Mattes provide a mechanism for mixing two images. The Image Compression Manager displays the weighted average of the source and destination based on the corresponding pixel in the matte.

Decompress the image by calling the Image Compression Manager's `DecompressImage` or `FDecompressImage` function. Your application must provide an image description structure along with the other parameters governing the operation. Use the `DecompressImage` function for simple decompression operations. If your application needs greater control, use the `FDecompressImage` function. See Working With Images (page 48) for detailed descriptions of these functions.

The Image Compression Manager manages the decompression operation and invokes the appropriate decompressor component. The manager returns the decompressed image to the location specified by your application.

Image Transcoding Functions

A transcoder translates an image compressed in one format into a different compression format. A transcoder can use an algorithm that directly translates from one format into another, which is often faster and more accurate than decompression and recompression. If your application requests decompression of an image, but no decompressor for the image can be found, QuickTime will search for a transcoder that can be used to convert the image into a format for which a decompressor is available. The transcoder functions directly available to your application are:

- `ImageTranscodeSequenceBegin`
- `ImageTranscodeFrame`
- `ImageTranscodeDisposeFrameData`
- `ImageTranscodeSequenceEnd`

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The Image Description Structure

An image description structure contains information that defines the characteristics of a compressed image or sequence. Data in the image description structure indicates the type of compression that was used, the size of the image when displayed, the resolution at which the image was captured, and so on. One image description structure may be associated with one or more compressed frames.

The `ImageDescription` data type defines the layout of an image description structure. In addition, an image description structure may contain additional data in extensions and custom color tables. The Image Compression Manager provides functions that allow you to get and set the data in image description structure extensions and custom color tables.

- See Working With Images (page 48) for more information about the functions `GetImageDescriptionCTable` and `SetImageDescriptionCTable`, which allow you to work with custom color tables in image description structures.
- See the `GetImageDescriptionExtension`, `SetImageDescriptionExtension`, `RemoveImageDescriptionExtension`, `CountImageDescriptionExtensionType`, and `GetNextImageDescriptionExtensionType` functions, which allow you to work with image description structure extensions.

```
struct ImageDescription {
    long idSize;            /* total size of this structure */
    CodecType cType;        /* compressor creator type */
    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;    /* compressor that created data */
    long vendor;            /* compressor developer 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 the color table for image */
};
typedef struct ImageDescription ImageDescription;
typedef ImageDescription *ImageDescriptionPtr, **ImageDescriptionHandle;
```

Field	Description
`idSize`	Defines the total size of this image description structure with extra data including color lookup tables and other per sequence data.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
cType	Indicates the type of compressor component that created this compressed image data. The value of this field indicates the compression algorithm supported by the component. The Codec data type defines a field in the compressor name list structure that identifies the compression method employed by a given compressor component. Apple Computer's Developer Technical Support group assigns these values so that they remain unique. These values correspond, in turn, to text strings that can identify the compression method to the user. See the description of GetCodecNameList for a list of valid values.
resvd1	Reserved for Apple. This field must be set to 0.
resvd2	Reserved for Apple. This field must be set to 0.
dataRefIndex	Reserved for Apple. This field must be set to 0.
version	Indicates the version of the compressed data. The contents of this field should indicate the version of the compression algorithm that was used to create the compressed data. By examining this field, decompressors that support many versions of an algorithm can determine the proper way to decompress the image.
revisionLevel	Indicates the version of the compressor that created the compressed image. Developers of compressors and decompressors assign these version numbers.
vendor	Identifies the developer of the compressor that created the compressed image.
temporalQuality	Indicates the degree of temporal compression performed on the image data associated with this description. This field is valid only for sequences.
spatialQuality	Indicates the degree of spatial compression performed on the image data associated with this description. This field is valid for sequences and still images.
width	Contains the width of the source image, in pixels.
height	Contains the height of the source image, in pixels.
hRes	Contains the horizontal resolution of the source image, in dots per inch.
vRes	Contains the vertical resolution of the source image, in dots per inch.
dataSize	Indicates the size of the compressed image, in bytes. This field is valid only for still images. Set this field to 0 if the size is unknown.
frameCount	Contains the number of frames in the image data associated with this description.
name	Indicates the compression algorithm used to create the compressed data. This algorithm is stored in Pascal string format. It always takes up 32 bytes no matter how long the string is. The 32 bytes consist of 31 bytes plus one length byte. The value of this field should correspond to the compressor type specified by the cType field, as well as to the value of the typeName field in the appropriate compressor name structure returned by the GetCodecNameList function. Applications may use the contents of this field to indicate the type of compression used for the associated image.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
depth	Contains the pixel depth specified for the compressed image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the depth of color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
clutID	Contains the ID of the color table for the compressed image, or other special values. If this field is set to 0, then a custom color table is defined for the compressed image. You can use the GetImageDescriptionCTable function to retrieve the color table. If this field is set to -1, the image does not use a color table.

See Compression Quality Constants (page 53) for a list of available values for the temporalQuality and spatialQuality fields.

Compression Quality Constants

Compressor components may allow applications to assert some control over the image quality that results from a compression or decompression operation. For example, the CompressSequenceBegin function provides the spatialQuality and temporalQuality parameters so that applications can indicate the level of image accuracy desired within individual frames and across adjacent frames in a sequence, respectively. These quality values become a property of the compressed data and are stored in the image description structure (described on The Image Description Structure (page 51)) associated with the image or sequence.

For a given compression operation, your application can determine the quality that the component supports by issuing the GetCompressionTime function.

The CodecQ data type defines a field that identifies the quality characteristics of a given image or sequence. Note that individual components may not implement all the quality levels shown here. In addition, components may implement other quality levels in the range from codecMinQuality to codecMaxQuality. Relative quality should scale within the defined value range. Values above codecLosslessQuality are reserved for use by individual components.

```
/* compression quality values */
#define codecMinQuality         0x000L      /* minimum valid value */
#define codecLowQuality         0x100L      /* low-quality reproduction */
#define codecNormalQuality
                                0x200L      /* normal-quality repro */
#define codecHighQuality
                                0x300L      /* high-quality repro */
#define codecMaxQuality         0x3FFL      /* maximum-quality repro */
#define codecLosslessQuality
                                0x400L      /* lossless-quality repro */
typedef unsigned long CodecQ;
```

Field	Description
codecMinQuality	Specifies the minimum valid value for a CodecQ field.
codecLowQuality	Specifies low-quality image reproduction. This value should correspond to the lowest image quality that still results in acceptable display characteristics.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
codecNormalQuality	Specifies image reproduction of normal quality.
codecHighQuality	Specifies high-quality image reproduction. This value should correspond to the highest image quality that can be achieved with reasonable performance.
codecMaxQuality	Specifies the maximum standard value for a CodecQ field.
codecLosslessQuality	Specifies lossless compression or decompression. This special value is valid only for components that can support lossless compression or decompression.

The Compressor Name List Structure

The compressor name list structure contains a list of compressor name structures. (A compressor name structure identifies a compressor or decompressor component.) The data structure contains name and type information for the component. The GetCodecNameList function returns an array of these structures, formatted into a compressor name list structure. The CodecNameSpecList data type defines a compressor name list structure.

```
/* compressor name list structure */
struct CodecNameSpecList {
    short count;        /* how many compressor name structures */
    CodecNameSpec list[1];
                        /* array of compressor name structures */
};
typedef struct CodecNameSpecList CodecNameSpecList;
typedef CodecNameSpecList *CodecNameSpecListPtr;
```

Field	Description
count	Indicates the number of compressor name structures contained in the list array that follows.
list	Contains an array of compressor name structures. Each structure corresponds to one compressor component or type that meets the selection criteria your application specifies when it issues the GetCodecNameList function. The count field indicates the number of structures stored in this array.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

How to Compress and Decompress Sequences of Images

This chapter provides a sample program illustrating the processes used to compress and decompress image sequences. The listing is broken up into a main program and a series of functions. Each function is introduced with a discussion of its purpose.

Functions for saving a sequence of images to a disk and for creating, compressing, and drawing a sequence of images are also discussed.

Compressing Sequences

The Image Compression Manager also provides functions that allow your application to compress and decompress sequences of images, such as might constitute a QuickTime movie. The tools provided by the Image Compression Manager focus on image compression and decompression and on the ordering of the images in a sequence, not on timing considerations. Use the Movie Toolbox to handle all the issues relating to the amount of time each image should be shown on the screen. For information on decompressing image sequences, see the next section, Decompressing Sequences (page 56).

A series of images can be compressed as a sequence if those images share an image description. That is, each image in the sequence must have the same compressor type, pixel depth, color lookup table, and boundary dimensions. To take best advantage of temporal compression, the images should also be related to each other (like frames in a movie), but this relationship is not necessary for them to be grouped as a sequence. If you create a sequence from completely unrelated images, you may not be able to achieve significant temporal compression.

When compressing image sequences, your application must perform several steps in addition to those required for single-frame image compression. This section describes a typical function for compressing an image sequence. Note that much of the setup processing is the same as that performed for single-frame images.

First, determine the parameters for the compression operation. As with single-image compression, the user may specify these parameters in a dialog box you can supply via the standard image-compression dialog component. Your application may choose to give the user the ability to specify such parameters as the compression algorithm, image quality, and so on. Note that image sequences require additional parameters, such as temporal quality.

Your application may give the user the option of specifying a compression algorithm based on an important performance characteristic. For example, the user may be most concerned with size, speed, or accuracy. The Image Compression Manager allows your application to choose the compressor component that meets the specified criterion.

Your application signals its intention to compress an image sequence by issuing the Image Compression Manager's `CompressSequenceBegin` function (see Working With Sequences (page 49) for more information about this function). At this time your application specifies many of the parameters that govern the sequence-compression operation. When you set the compression parameters and the `temporalQuality`

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

parameter is not 0, then be sure to set the value of either the `codecFlagUpdatePrevious` or `codecFlagUpdatePreviousComp` flag to 1 in the `flags` parameter of the `CompressSequenceBegin` function.

Once you have started the sequence, you then compress each image in the sequence by performing the following steps:

1. Your application must call the Image Compression Manager's `GetMaxCompressionSize` function to determine the maximum size of the compressed data that will result from the current image (see Getting Information About Compressed Data (page 44) for more information about this function). You provide the specified compression parameters to this function. In response, the Image Compression Manager invokes the appropriate compressor component to determine the number of bytes required to store the largest compressed image in the sequence. Your application should then reserve sufficient memory to accommodate that compressed image. You can use this returned value until you change the settings of the compression parameters.

2. Your application must call the `CompressSequenceFrame` function to compress the image (see Working With Sequences (page 49) for more information about this function). It may be necessary or desirable for your application to change one or more of the compression parameters while processing a sequence. The Image Compression Manager provides several functions that allow your application to modify such parameters as the spatial or temporal quality or the data-unloading function. See Changing Sequence-Compression Parameters (page 47) for more information about these functions.

3. The Image Compression Manager manages the compression operation and invokes the appropriate compressor. The manager returns the compressed image and its associated image description to your application.

4. Your application is then free to store the compressed image with the others in the sequence.

After the entire sequence is compressed, you end the process by calling the `CDSequenceEnd` function.

Decompressing Sequences

The Movie Toolbox handles the details of displaying compressed image sequences that are stored in QuickTime movies. However, if you want to work with sequences in your application, the Image Compression Manager provides tools for decompressing image sequences. As with still-image compression, decompressing sequences requires additional effort on the part of your application. In addition, there are some processing considerations that are particular to sequence decompression. This section describes the steps necessary to decompress an image sequence. Then it discusses several points you should consider before decompressing a sequence.

When decompressing an image sequence, your application must first determine where to display the decompressed sequence. Your application must specify the destination graphics port to the Image Compression Manager. In addition, you may indicate that only a portion of the source image is to be displayed. You describe the desired portion of the image by specifying a rectangle in the coordinate system of the source image. You can determine the size of the source image by examining the image description structure associated with the image (see The Image Description Structure (page 51) for more information about image description structures).

Your application may also specify that the image is to be mapped into the destination graphics port. The `DecompressSequenceBegin` function allows your application to specify a mapping matrix for the operation.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Your application can invoke additional effects by specifying a mask region or blend matte for the image. Mask regions and mattes control which pixels in the source image are drawn to the destination. Mask regions must be defined in the destination coordinate system. During decompression the Image Compression Manager displays only those pixels in the source image that correspond to bits in the mask that are set to 1. Mattes contain several bits per pixel and are defined in the coordinate system of the source image. Mattes provide a mechanism for blending pixels from source images.

Your application signals its intention to decompress an image sequence by issuing the Image Compression Manager's `DecompressSequenceBegin` function. At this time your application specifies many of the parameters that govern the sequence-decompression operation. The Image Compression Manager, in turn, allocates system resources that are necessary for the operation.

Once you have started the sequence, you then decompress each image in the sequence. Call the `DecompressSequenceFrame` function to decompress the image. It may be necessary or desirable for your application to change one or more of the decompression parameters while processing a sequence. The Image Compression Manager provides several functions that allow your application to modify such parameters as the accuracy, the transformation matrix, or the data-loading function. See Changing Sequence-Decompression Parameters (page 48) for more information about these functions.

The Image Compression Manager manages the decompression operation and invokes the appropriate compressor component. The manager returns the decompressed image to the location specified by your application and applies any effects you may have specified.

After the entire sequence is decompressed, you end the process by calling the `CDSequenceEnd` function.

The Basic Functions to Use

The basic functions used to compress and decompress a sequence of images include

- `SetSequenceProgressProc`
- `GetCSequenceMaxCompressionSize`
- `DecompressSequenceBeginS`
- `DecompressSequenceFrameWhen`
- `DecompressSequenceFrameS`
- `CDSequenceFlush`
- `SetDSequenceTimeCode`
- `CDSequenceEquivalentImageDescription`
- `CDSequenceNewMemory`
- `CDSequenceDisposeMemory`
- `CDSequenceInvalidate`
- `PtInDSequenceData`

Note that the sequence itself contains no time information, only the order in which images should appear. This book defines several new functions for working with sequences, including functions supporting timecodes and asynchronous decompression.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Decompressing Still Images From a Sequence

Your application can, of course, decompress individual images from a sequence. When doing so, you must be careful to select only those frames that do not depend on other frames. That is, do not decompress frames from a sequence that has been temporally compressed unless you first decompress all the frames in sequence starting from the preceding key frame (see Defining Key Frame Rates (page 58) for more information on key frames in image sequences). In general, you should decompress images from sequences as sequences, rather than as individual frames.

Using Screen Buffers and Image Buffers

The use of screen buffers has been discontinued in QuickTime. Use only image buffers. A request for a screen buffer will return an image buffer. Do not request a screen buffer in your application.

The Image Compression Manager uses image buffers when decompressing sequences that have been temporally compressed and therefore contain key frames. Image buffers are especially useful when you want to skip to random frames within a sequence. Random frame access in temporally compressed sequences forces the compressor to decompress all the frames between the nearest preceding key frame and the desired frame. Reconstructing the frame in this manner on the screen can result in jerky sequence display. As an alternative, the compressor can reconstruct the frame in the offscreen image buffer and then copy it to the screen when appropriate. Image buffers are allocated at an appropriate depth and size for the decompressor.

Your application can control the use of the image buffer by the compressor component. For example, you can force the compressor to draw images only to the image buffer, not to the screen. In this manner you can use the image buffer to build up sequences without making the process visible. You can also control when the compressor uses the image buffer. You may need to do this when your program is decompressing directly to the screen and suddenly is prevented from doing so (for example, when your window becomes hidden).

Defining Key Frame Rates

The process of temporal compression involves reducing or eliminating temporal redundancy from an image sequence. Temporal compression is most effective when a sequence contains frames that bear significant similarity to adjacent frames. This is typically true of movies and other video sequences. Reconstructing an individual frame within a sequence that has been temporally compressed requires knowledge of the previous frames. This does not present a problem if your application always plays compressed sequences from the beginning. However, if your application needs to start playing a sequence from a random point, or perhaps backward, the decompressor does not have enough information to decompress the frames.

To alleviate this problem, compressors insert key frames in compressed sequences at regular intervals. **Key frames** define starting points for portions of a temporally compressed sequence. Subsequent frames depend on the previous key frame.

At the start of a sequence compression your application can specify a rate at which the compressor is to insert key frames into the compressed data stream. This **key frame rate** indicates the maximum number of frames you will accept between key frames. The Image Compression Manager picks the best key frames from the source sequence and at the same time enforces the specified key frame rate (the best key frames are those that are least similar to adjacent frames, such as at scene changes; these frames would have the largest compressed images even if they were not selected as key frames).

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

During sequence compression your application can change the key frame rate by calling the SetCSequenceKeyFrameRate function. By manipulating the parameters for the sequence, you can force the Image Compression Manager to place a key frame at any arbitrary point in a sequence (set the codecFlagForceKeyFrame flag to 1 in the flags parameter of the CompressSequenceFrame function.

A Sample Program for Compressing and Decompressing a Sequence of Images

The sample program presented in this section illustrates the processes described in the previous sections. The program has been divided into several functions. Listing 6-1 shows the main program.

The data for each frame is written to the data fork of the disk file, preceded by a long word that contains the number of bytes of data for that frame. A description of the compressed images in the sequence is stored in a 'SEQU' resource in the same file with a resource ID of 128 or 129. This description is simply the image description structure maintained by the Image Compression Manager.

The image for each frame of the sequence is drawn into an offscreen graphics world that the SequenceSave function creates in the currWorld variable. SequenceSave calls the DrawOneFrame function (described in the next section) to draw each frame's image into the currWorld variable. Before any of the frames of the sequence are drawn, the Image Compression Manager is prepared to compress a sequence of images through the CompressSequence function.

Listing 6-1 Compressing and decompressing a sequence of images: The main program

```
WindowPtr       displayWindow;          /* window in which to display
                                            sequence */
Rect            windowRect;    /* rectangle of displayWindow */
main (void)
{
    WindowPtr       displayWindow;
    Rect            windowRect;

    InitGraf (&thePort);
    InitFonts ();
    InitWindows ();
    InitMenus ();
    TEInit ();
    InitDialogs (nil);

    SetRect (&windowRect, 0, 0, 256, 256);
    OffsetRect (&windowRect,/* middle of screen */
        ((qd.screenBits.bounds.right - qd.screenBits.bounds.left) -
                 windowRect.right) / 2,
        ((qd.screenBits.bounds.bottom - qd.screenBits.bounds.top) -
                 windowRect.bottom) / 2);
    displayWindow = NewCWindow (nil, &windowRect,
                                           "\pImage", true, 0,
                                           (WindowPtr)-1, true, 0);
    if (displayWindow)
    {
        SetPort (displayWindow);
        SequenceSave ();
        SequencePlay ();
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    }
}
```

A Sample Function for Saving a Sequence of Images to a Disk File

The `SequenceSave` function shown in Listing 6-2 saves a sequence of images to a disk file. This function creates and opens a disk file for the image sequence, calls the `CompressSequence` function to create and compress the image sequence into the file, and then calls the `MakeMyResource` function to save the image description resource in the file, so that the sequence can be played back later. For details on `CompressSequence`, see the next section.

Listing 6-2 Saving a sequence of images to a disk file

```
void SequenceSave (void)
{
    long                            filePos;
    StandardFileReply               fileReply;
    short                           dfRef = 0;
    OSErr                           error;
    ImageDescriptionHandle          description = nil;
    StandardPutFile ("\p", "\pSequence File", &fileReply);
    if (fileReply.sfGood)
    {
        if (! (fileReply.sfReplacing))
        {
            error = FSpCreate (&fileReply.sfFile, 'SEQM', 'SEQU',
                                    fileReply.sfScript);
            CheckError (error, "\pFSpCreate");
        }
        error = FSpOpenDF (&fileReply.sfFile, fsWrPerm, &dfRef);
        CheckError (error, "\pFSpOpenDF");

        error = SetFPos (dfRef, fsFromStart, 0);
        CheckError (error, "\pSetFPos");

        CompressSequence (&dfRef, &description);
        error = GetFPos (dfRef, &filePos);
        CheckError (error, "\pGetFPos");

        error = SetEOF (dfRef, filePos);
        CheckError (error, "\pSetEOF");

        FSClose (dfRef);
        FlushVol (nil, fileReply.sfFile.vRefNum);
        MakeMyResource (fileReply, description);
        if (description != nil)
            DisposeHandle ((Handle) description);
    }
}
void MakeMyResource ( StandardFileReply fileReply,
                            ImageDescriptionHandle description)
{
    OSErr       error;
    short       rfRef;
    Handle      sequResource;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    FSpCreateResFile (&fileReply.sfFile, 'SEQM', 'SEQU',
                         fileReply.sfScript);
    error = ResError();
    if (error != dupFNErr)
        CheckError (error, "\pFSpCreateResFile");
    rfRef = FSpOpenResFile (&fileReply.sfFile, fsRdWrPerm);
    CheckError (ResError (), "\pFSpOpenResFile");
    SetResLoad (false);
    sequResource = Get1Resource ('SEQU', 128);
    if (sequResource)
        RmveResource (sequResource);
    SetResLoad (true);
    sequResource = (Handle) description;
    error = HandToHand (&sequResource);
    CheckError (error, "\pHandToHand");
    AddResource (sequResource,'SEQU', 128, "\p");
    CheckError (ResError (), "\pAddResource");
    UpdateResFile (rfRef);
    CheckError (ResError (), "\pUpdateResFile");
    CloseResFile (rfRef);
}
```

A Sample Function for Creating, Compressing, and Drawing a Sequence of Images

Listing 6-3 shows the `CompressSequence` function, which creates and then compresses the image sequence. `CompressSequenceBegin` informs the Image Compression Manager which compressor (of type `codectype`) to use, what the desired compression quality is, the key frame rate, the portion of the image to compress (in this example, the entire image is compressed), and the image to be compressed (in this example, the pixel map, of type `PixMap`, in the `currWorld` variable).

`CompressSequenceBegin` returns a unique number that identifies the sequence for subsequent image-compression routines, and it initializes a new image description structure, which is stored in the handle referenced by the `description` local variable.

Using a loop, the `DrawOneFrame` function draws each frame until the last frame is drawn, at which time the function returns the value of `false`. Each frame that it draws is copied to the window so that it can be seen during the compression sequence.

The `CompressSequenceFrame` function is used to compress each frame's image. `CompressSequenceFrame` tells the Image Compression Manager

- which image to compress (in this case, the pixel map of the `currWorld` variable)
- the portion of that image to compress (in this case, all of it)
- whether to update the previous frame's buffer for frame differencing
- the address of the buffer that's to receive the compressed image data

In updating the previous frame's buffer for frame differencing, the Image Compression Manager control flag `codecFlagUpdatePrevious` copies the uncompressed image to the previous frame's buffer; contrast this with the `codecFlagUpdatePreviousComp` flag, which copies the compressed image to the previous frame's buffer. The more lossy the compression, the more the difference between the compressed and uncompressed images.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The CompressSequenceBegin function returns a rating of the similarity between the current frame and the previous frame, but this example ignores this rating. After each frame is compressed, the number of bytes in the compressed image data is written to the disk file, followed by the compressed image data itself.

After all the images in the sequence have been compressed, the CDSequenceEnd function is called to tell the Image Compression Manager that the sequence is over. The data fork of the file is closed, and the image description is written to a 'SEQU' resource.

The DrawOneFrame function draws one frame of the sequence with QuickDraw. The frame's image is drawn into the rectangle specified by the destRect parameter. The image is a set of **color ramps** in which the shading goes from light to dark in smooth increments. The color ramps fill the destination rectangle and the current frame number centered within the destination rectangle over the ramps.

The PaintImage function paints a series of vertical color ramps into the rectangle specified by the destRect parameter into the current color graphics port. This is done through a nested loop. The outer loop iterates only twice, and half of the ramps are drawn in the first iteration and half in the second. The inner loop iterates over all the steps in a ramp.

Listing 6-3 Creating and compressing an image sequence

```
void CompressSequence (short* dfRef, ImageDescriptionHandle* description)
{
    GWorldPtr           currWorld = nil;
    PixMapHandle        currPixMap;
    CGrafPtr            savedPort;
    GDHandle            savedDevice;
    Handle              buffer = nil;
    Ptr                 bufferAddr;
    long                compressedSize;
    long                dataLen;
    Rect                imageRect;
    ImageSequence       sequenceID = 0;
    short               frameNum;
    OSErr               error;
    CodecType           codecKind = 'rle ';

    GetGWorld (&savedPort, &savedDevice);
    imageRect = savedPort->portRect;
    error = NewGWorld (&currWorld, 32, &imageRect, nil, nil, 0);
    CheckError (error, "\pNewGWorld");
    SetGWorld (currWorld, nil);

    currPixMap = currWorld->portPixMap;
    LockPixels (currPixMap);
/*
    Allocate an embryonic image description structure and the
    Image Compression Manager will resize.
*/
    *description = (ImageDescriptionHandle) NewHandle (4);

    error = CompressSequenceBegin (
            &sequenceID,
            currPixMap,
            nil,                        /* tell ICM to allocate previous
                                             image buffer */
            &imageRect,
            &imageRect,
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
            0,                          /* let ICM choose pixel depth */
            codecKind,
            (CompressorComponent) anyCodec,
            codecNormalQuality,         /* spatial quality */
            codecNormalQuality,         /* temporal quality */
            5,                          /* at least 1 key frame every
                                            5 frames */
            nil,                        /* use default color table */
            codecFlagUpdatePrevious,
            *description );
    CheckError (error, "\pCompressSequenceBegin");
    error = GetMaxCompressionSize(
            currPixMap,
            &imageRect,
            0,                              /* let ICM choose pixel depth */
            codecNormalQuality,             /* spatial quality */
            codecKind,
            (CompressorComponent) anyCodec,
            &compressedSize );
    CheckError (error, "\pGetMaxCompressionSize");
    buffer = NewHandle(compressedSize);
    CheckError (MemError(), "\pNewHandle buffer");
    MoveHHi (buffer);
    HLock (buffer);
    bufferAddr = StripAddress (*buffer);

    for (frameNum = 1; frameNum <= 10; frameNum++)
    {
        DrawFrame (&imageRect, frameNum);
        error = CompressSequenceFrame (
                        sequenceID,
                        currPixMap,
                        &imageRect,
                        codecFlagUpdatePrevious,
                        bufferAddr,
                        &compressedSize,
                        nil,
                        nil );
        CheckError (error, "\pCompressSequenceFrame");
        dataLen = 4;
        error = FSWrite (*dfRef, &dataLen, &compressedSize);
        CheckError (error, "\pFSWrite length");
        error = FSWrite (*dfRef, &compressedSize, bufferAddr);
        CheckError (error, "\pFSWrite buffer");
    }
    CDSequenceEnd (sequenceID);

    DisposeGWorld (currWorld);
    SetGWorld (savedPort,savedDevice);
    if (buffer) DisposeHandle ( buffer );
    }
void DrawFrame (const Rect *imageRect, long frameNum)
{
    Str255 numStr;
    ForeColor( redColor );
    PaintRect( imageRect );
    ForeColor( blueColor );
    NumToString (frameNum, numStr);
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    MoveTo ( imageRect->right / 2, imageRect->bottom / 2);
    TextSize ( imageRect->bottom / 3);
    DrawString (numStr);
}
```

A Sample Function for Decompressing and Playing Back a Sequence From a Disk File

The `SequencePlay` function, shown in Listing 6-4, plays back a sequence of images from a disk file that was created by the `SequenceSave` function.

The `SequencePlay` function begins by grabbing the image description structure from the file that the user specified from a `'SEQU'` resource ID 128. This structure is needed to decompress the images in the file.

Before these compressed images are read, the Image Compression Manager is told to prepare to decompress a sequence of images through the `DecompressSequenceBegin` function. This routine tells the Image Compression Manager

- how the images were compressed with the image description structure
- where to display the decompressed image (the current port in this example)
- what part of the image to decompress (all of it)
- what transfer mode to use when displaying the image (`srcCopy`)
- whether to buffer the image for frame differences

A loop iterates for each frame in the file. For each frame, a long word with the number of bytes in the frame is read from the file, and then that many bytes are read from the file into a compressed-image buffer. This buffer is passed to `DecompressSequenceFrame`, which decompresses the image to the screen (the destination doesn't have to be the screen, but it is in this example). The loop iterates until the end of the file has been reached.

Listing 6-4 Playing back a sequence of images from a disk file

```
void SequencePlay (void)
{
    ImageDescriptionHandle          description;
    long                        compressedSize;
    Handle                      buffer = nil;
    Ptr                         bufferAddr;
    long                    dataLen;
    long                        lastTicks;
    ImageSequence               sequenceID;
    Rect                        imageRect;
    StandardFileReply           fileReply;
    SFTypeList                  typeList = {'SEQU',0,0,0};
    short                       dfRef = 0;          /* sequence data fork */
    short                       rfRef = 0;          /* sequence resource fork
*/
    OSErr                       error;

    StandardGetFile (nil, 1, typeList, &fileReply);
    if (!fileReply.sfGood) return;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    rfRef = FSpOpenResFile (&fileReply.sfFile, fsRdPerm);
    CheckError (ResError (), "\pFSpOpenResFile");
    description = (ImageDescriptionHandle)
                            Get1Resource ('SEQU', 128);
    CheckError (ResError (), "\pGet1Resource");
    DetachResource ((Handle) description );
    HNoPurge ((Handle) description );
    CloseResFile (rfRef);
    error = FSpOpenDF (&fileReply.sfFile, fsRdPerm, &dfRef);
    CheckError (error, "\pFSpOpenDF");
    buffer = NewHandle (4);
    CheckError (MemError (), "\pNewHandle buffer");
    SetRect (&imageRect, 0, 0, (**description).width,
                                     (**description).height);
    error = DecompressSequenceBegin (
                        &sequenceID,
                        description,
                        nil,                        /* use the current port */

                        nil,                        /* go to screen */
                        &imageRect,
                        nil,                        /* no matrix */
                        ditherCopy,
                        nil,                        /* no mask region */
                        codecFlagUseImageBuffer,
                        codecNormalQuality,         /* accuracy */
                        (CompressorComponent) anyCodec);
    while (true)
    {
        dataLen = 4;
        error = FSRead (dfRef, &dataLen, &compressedSize);
        if (error == eofErr)
            break;
        CheckError( error, "\pFSRead" );

        if (compressedSize > GetHandleSize (buffer))
        {
            HUnlock (buffer);
            SetHandleSize (buffer, compressedSize);
            CheckError (MemError(), "\pSetHandleSize");
        }
        HLock (buffer);
        bufferAddr = StripAddress (*buffer);
        error = FSRead (dfRef, &compressedSize, bufferAddr);
        CheckError (error, "\pFSRead");

        error = DecompressSequenceFrame (
                            sequenceID,
                            bufferAddr,
                            0,  // flags
                            nil,
                            nil );
        CheckError (error, "\pDecompressSequenceFrame");
        Delay (30, &lastTicks);
    }

    CDSequenceEnd (sequenceID);
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    if (dfRef) FSClose (dfRef);
    if (buffer) DisposeHandle (buffer);
    if (description) DisposeHandle ((Handle)description);
}
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

ICM Functions, Data Types, and Constants

This chapter describes the various functions, data types, and constants your application can take advantage of in working with the Image Compression Manager.

Making Thumbnail Pictures

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.

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 (page $@).

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	Description
`dataRate`	Specifies the bytes per second to which the data rate must be constrained.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
dataOverrun	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.
frameDuration	Specifies the duration of the current frame in milliseconds.
keyFrameRate	Indicates the frequency of key frames. This frequency is normally identical to the key frame rate passed to the CompressSequenceBegin function.
minSpatialQuality	Specifies the minimum spatial quality the compressor should use to meet the requested data rate.
minTemporalQuality	Indicates the minimum temporal quality the compressor should use to meet the requested data rate.

See Compression Quality Constants (page 53) for available values of the minSpatialQuality and minTemporalQuality fields.

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

The Compressor Information Structure

Your application can retrieve information describing the capabilities of compressors with the GetCodecInfo function. The CodecInfo data type defines the format of the compressor information structure.

```
/* compressor information structure */
struct CodecInfo {
    Str31 typeName;             /* compression algorithm (codec type) */
    short version;              /* version supported by component */
    short revisionLevel;        /* version assigned by developer */
    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 this algorithm */
    unsigned char decompressionAccuracy;
                                /* relative accuracy of this algorithm */
    unsigned short compressionSpeed;
                                /* relative compression speed */
    unsigned short decompressionSpeed;
                                /* relative decompression speed */
    unsigned char compressionLevel;
                                /* relative compression of component */
    char resvd;                 /* reserved--set to 0 */
    short minimumHeight;        /* minimum image height for component */
    short minimumWidth;         /* minimum image width for component */
    short decompressPipelineLatency;
                                /* in milliseconds (asynchronous) */
    short compressPipelineLatency;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
                            /* in milliseconds (asynchronous) */
    long privateData;       /* reserved for use by Apple */
};
typedef struct CodecInfo CodecInfo;
```

Field	Description
typeName	Indicates the compression algorithm used by the component; for example, 'Animation'. This Pascal string may be used to identify the compression algorithm to the user. The string always takes up 32 bytes no matter how long it is. The 32 bytes consist of 31 bytes plus one length byte. Apple Computer's Developer Technical Support group assigns these type names. The value of this field should correspond to the value of the typeName field in the appropriate compressor name structure returned by the GetCodecNameList function.
version	Indicates the version of compressed data this component supports. The contents of this field should indicate the most recent version of the compression algorithm that the component can understand.
revisionLevel	Indicates the version of the component; for example, 0x00010001 (1.0.1). Developers of compressors assign these version numbers.
vendor	Identifies the developer of the component; for example, 'appl'. The value of this field corresponds to the manufacturer code or application signature assigned to the developer.
decompressFlags	Contains flags that specify the decompression capabilities of the component. Typically, these flags are of interest only to developers of image decompressors.
compressFlags	Contains flags that specify the compression capabilities of the component. Typically, these flags are of interest only to developers of image compressors.
formatFlags	Contains flags that describe the possible format for compressed data produced by this component and the format of compressed files that the component can handle during decompression. Typically, these flags are of interest only to developers of compressor components.
compressionAccuracy	Indicates the relative accuracy of the compression algorithm employed by the component. Valid values for this field range from 0 to 255. A value of 0 means that the accuracy is unknown. Values from 1 to 255 provide a gauge for the relative accuracy of the compression algorithm; higher values indicate better accuracy. The Image Compression Manager examines this field to determine which compressor component can most accurately compress a given image. The compressionAccuracy field can only approximate the accuracy of a compression algorithm. Typically, compression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a compression request is issued, a precise measure of accuracy is not possible. However, the value of this field should still give a rough idea of the accuracy of the supported algorithm.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
decompression-Accuracy	Indicates the relative accuracy of the decompression algorithm employed by the component. Valid values for this field range from 0 to 255. A value of 0 means that the accuracy is unknown. Values from 1 to 255 indicate the relative accuracy of the decompression technique; higher values mean better accuracy. The Image Compression Manager examines this field to determine which decompressor component can most accurately decompress a given image. The decompressionAccuracy field can only approximate the accuracy of a decompression algorithm. Typically, decompression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a decompression request is issued, a precise measure of accuracy is not possible. However, the value of this field should still give a rough idea of the accuracy of the supported algorithm.
compressionSpeed	Indicates the relative speed of the component for compression operations. Valid values for this field lie in the range from 0 to 65,535. A value of 0 means that the speed is unknown. Values from 1 to 65,535 correspond to the number of milliseconds the component requires to compress a 320-by-240 pixel image on a Macintosh II computer. The Image Compression Manager examines this field to determine which compressor component can most quickly compress a given image.
decompressionSpeed	Indicates the relative speed of the component for decompression operations. Valid values for this field lie in the range from 0 to 65,535. A value of 0 means that the speed is unknown. Values from 1 to 65,535 correspond to the number of milliseconds the component requires to decompress a 320-by-240 pixel image on a Macintosh II computer. The Image Compression Manager examines this field to determine which compressor component can most quickly decompress a given image.
compressionLevel	Indicates the relative compression achieved by this component. Valid values for this field lie in the range from 0 to 255. A value of 0 means that the compression level is unknown. Values from 1 to 255 map to percentage values of relative compression; lower values mean lesser compression. A value of 1 means no compression (0 percent); a value of 255 means maximum compression (100 percent). The Image Compression Manager examines this field to determine which available compressor component will yield the smallest resulting data for a given image. The compressionLevel field can only approximate the effectiveness of a compression algorithm. Typically, compression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a compression request is issued, a precise measure of compression is not possible. However, the value of this field should still give a rough idea of the effectiveness of the supported algorithm.
resvd	Reserved for Apple. This field must be set to 0.
minimumHeight	Specifies the height in pixels of the smallest image the component can handle. Together with the minimumWidth field, this field defines the block size for the component. The Image Compression Manager does not issue compression or decompression requests for images smaller than the block size.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
minimumWidth	Specifies the width in pixels of the smallest image the component can handle. Together with the minimumHeight field, this field defines the block size for the component. The Image Compression Manager does not issue compression or decompression requests for images smaller than the block size.
decompressPipeline-Latency	Reserved for future use. This field must be set to 0.
compressPipeline-Latency	Reserved for future use. This field must be set to 0.
privateData	Reserved for use by Apple. This field must be set to 0.

The Compressor Name Structure

The CodecNameSpec data type defines a compressor name structure.

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

Field	Description
codec	Uniquely identifies the component or, in some cases, contains a special value that selects all components. If your application requests a list of components, the codec field in each compressor name structure contains the component ID for that compressor. If your application requests a list of component types, the codec field is set to 0 in each compressor name structure.
cType	Contains the type identifier for the compressor. The value of this field indicates the compression algorithm supported by the component. See the documentation for GetCodecNameList for a list of valid values.
typeName	Contains a text string in Pascal format that identifies the compression algorithm supported by the component. This string may be used to identify the compression algorithm to the user. The value of this field should correspond to the value of the typeName field in the appropriate compressor information structure returned by the component in response to a GetCodecInfo function.
name	Specifies the name of the compressor component. Developers assign these names to uniquely identify their products. This name may be used to identify the component to the user.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Controlling Hardware Scaling

QuickTime provides three functions that allow applications to zoom a monitor (GDHasScale, GDGetScale, and GDSetScale). These three functions are considered low-level calls (comparable to SetEntries) that you should use only when playing back QuickTime movies in a controlled environment with no user interaction. Also, because this capability is not present on all computers, applications should not depend on its availability.

These functions provide a standard way for you to access the resizing abilities of a user's monitor for playback. Effectively, this allows you to have full screen Cinepak playback on low-end Macintosh computers.

Working With the StdPix Function

To allow applications to have access to compressed image data as it is displayed, a graphics function has been added to the grafProcs field of the color graphics port structure (defined by the CGrafPort data type).

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.

Note: QuickDraw's CopyDeepMask function uses the StdPix function if QuickTime is present.

To work with the control information associated with a compressed image, you can use the SetCompressedPixMapInfo and GetCompressedPixMapInfo functions.

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.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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 (page $@) to override the standard alignment. 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.

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 (page $@)). 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
{
    ICMAlignmentUPP         alignmentProc;          /* pointer to your
                                                       alignment function */
    long                    alignmentRefCon;        /* reference constant */
} ICMAlignmentProcRecord, *ICMAlignmentProcRecordPtr;
```

Field	Description
`alignmentProc`	Points to your alignment function.
`alignmentRefCon`	Contains a reference constant for use by your alignment function.

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.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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 ICMDataProcRecord ICMDataProcRecord;
typedef ICMDataProcRecord *ICMDataProcRecordPtr;
```

The data-loading function structure contains the following fields:

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

Field	Description
`dataProc`	Contains a pointer to your data-loading function.
`dataRefCon`	Contains a reference constant for use by your data-loading function.

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

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 ICMFlushProcRecord ICMFlushProcRecord;
typedef ICMFlushProcRecord *ICMFlushProcRecordPtr;
```

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.

The data-unloading function structure contains the following fields:

```
struct ICMFlushProcRecord
{
    ICMFlushUPP flushProc;      /* pointer to data-unloading function */
    long            flushRefCon;/* reference constant */
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
};
```

Field	Description
flushProc	Contains a pointer to your data-unloading function.
flushRefCon	Contains a reference constant for use by your data-unloading function.

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 ICMProgressProcRecord ICMProgressProcRecord;
typedef ICMProgressProcRecord *ICMProgressProcRecordPtr;
```

The progress function structure contains the following fields:

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

Field	Description
progressProc	Contains a pointer to your progress function.
progressRefCon	Contains a reference constant for use by your 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.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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 ICMCompletionProcRecord *ICMCompletionProcRecordPtr;

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

Field	Description
`completionProc`	Contains a pointer to your completion function. Your completion function may be called at interrupt time. Therefore 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.

Constants

This section describes constants provided by the Image Compression Manager.

```
/* alpha channel graphics modes */
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
    graphicsModePreBlackAlpha       = 258,
    graphicsModeStraightAlphaBlend  = 260
};

/* fieldFlags for the ImageFieldSequenceExtractCombine function */
enum {
    evenField1ToEvenFieldOut        = 1<<0,
    evenField1ToOddFieldOut         = 1<<1,
    oddField1ToEvenFieldOut         = 1<<2,
    oddField1ToOddFieldOut          = 1<<3,
    evenField2ToEvenFieldOut        = 1<<4,
    evenField2ToOddFieldOut         = 1<<5,
    oddField2ToEvenFieldOut         = 1<<6,
    oddField2ToOddFieldOut          = 1<<7
};
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Image Compression Manager Function Control Flags

A number of Image Compression Manager functions take control flags that allow your application to exert greater control over the operation. In some cases, the Image Compression Manager returns status information about the results of the function in the same flags field. In general, you need to use only a few of these flags. The function descriptions in the reference section of this chapter indicate the flags that are valid for individual functions.

The `CodecFlags` data type defines these flag fields.

```
typedef unsigned short CodecFlags;

/* 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 was compressed */
#define codecFlagDontOffscreen              (1L<<5)
                                        /* don't go offscreen */

#define codecFlagUpdatePreviousComp (1L<<6)
                                        /* (input) update previous buffer */
#define codecFlagForceKeyFrame (1L<<7)
                                        /* force key frame from image */
#define codecFlagOnlyScreenUpdate
                                    (1L<<8)
                                        /* (input) only update screen */
#define     codecFlagLiveGrab (1L<<9)
                                        /* (input) grab live video */
#define     codecFlagUsedNewImageBuffer
                                    (1L<<14)
                                        /* (output) new image buffer used */
#define codecFlagUsedImageBuffer
                                    (1L<<15)
                                    /* (output) decompressor used
                                         offscreen buffer */
enum {
    codecFlagDontUseNewImageBuffer      = (1L << 10),
    codecFlagInterlaceUpdate            = (1L << 11),
    codecFlagCatchUpDiff                = (1L << 12)
};
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Constant	Description
codecFlagUse-ImageBuffer	Controls whether the decompressor allocates an offscreen buffer for decompression. If your application sets this flag to 1, the decompressor allocates an offscreen buffer the size of the compressed image. If you set this flag to 0, the decompressor does not use an offscreen image buffer. These image buffers are useful when decompressing sequences that were created using temporal compression.
codecFlagUseScreen-Buffer	Controls whether the decompressor allocates an offscreen destination buffer during decompression. If you set this flag to 1, the decompressor allocates an offscreen buffer the size of the destination screen. If you set this flag to 0, the decompressor does not use an offscreen screen buffer. Using a screen buffer helps to reduce tearing that may result when decompressing directly to the screen.
codecFlagUpdate-Previous	Controls whether the compressor updates the previous image buffer 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 source image into the previous frame buffer at the end of the frame compression.
codecFlagNoScreen-Update	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.
codecFlagWas-Compressed	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.
codecFlagDont-Offscreen	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.
codecFlagUpdate-PreviousComp	Controls whether the compressor updates the previous image buffer with the decompressed image data. 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 the 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.
codecFlagForce-KeyFrame	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.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Constant	Description
codecFlagOnly-ScreenUpdate	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 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.
codecFlagLiveGrab	Indicates to the compressor whether the current sequence results from grabbing live video. When working with live video, compressors operate as quickly as possible and disable some additional processing, such as compensation for previously compressed data. Set this flag to 1 when you are compressing from a live video source; the compressor then operates as quickly as it can.
codecFlagUsedNew-ImageBuffer	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. If this flag is set to 0, the decompressor did not use the image buffer.
codecFlagUsed-ImageBuffer	Indicates to your application that the decompressor used the offscreen image buffer for this frame. If this flag is set to 1, the decompressor used the image buffer. If this flag is set to 0, the decompressor did not use the image buffer.
codecFlagDontUse-NewImageBuffer	Forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.
codecFlagInterlace-Update	Updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).
codecFlagCatchUpDiff	Notifies the codec that the currently displayed frame is being displayed late in an attempt to "catch up" to the current frame, which only happens with compression formats that support frame differencing, You can pass this flag to any of the DecompressSequenceFrame calls.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

About Image Compressor Components

This chapter describes the general characteristics of image compressor components.

Image compressor components are registered by the Component Manager, and they present a standard interface to the Image Compression Manager. See Functions (page 105) for a detailed description of the functions that image compressor components must provide. An image compressor component can be a systemwide resource, or it can be local to a particular application.

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

Compressor Types

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

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

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

Utility and Callback Functions

The Image Compression Manager provides a set of utility functions for compressor components. These functions allow compressors and decompressors to create custom color lookup tables, among other things. For a complete description of these utility functions, along with the functions that must be supported by compressor components, see Image Compression Manager Utility Functions (page 107).

The Image Compression Manager defines four callback functions that may be provided to compressors and decompressors by applications. These callback functions are data-loading functions, data-unloading functions, completion functions, and progress functions. Data-loading functions and data-unloading functions support

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

spooling of compressed data. Completion functions allow components to report that asynchronous operations have completed. Progress functions provide a mechanism for components to report their progress toward completing an operation.

Banding and Extending Images

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

The Image Compression Manager determines the height of a band based on the amount of available memory and the `bandMin` and `bandInc` parameters provided by the compressor component in the compressor capability structure (see Data Structures (page 105)). The `bandMin` field specifies the minimum band height supported by a decompressor component. By providing a minimum height, decompressor components that operate on blocks of pixels can operate more efficiently since the minimum height ensures that a band has at least one row of pixel blocks. The `bandInc` field specifies the increment in pixels by which the height of a band is increased above the minimum when sufficient memory is available. This specification allows easier processing by ensuring that a band is an integral number of rows of blocks. The larger these two parameters, the more memory is required for the band buffer, which may limit the size of images used with a given amount of memory. By specifying a minimum height that is the size of the image, the compressor component can indicate that it cannot handle banded images. However, the specification of a full size is not recommended unless required by the compression format, since it requires large amounts of memory for large images.

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

The depth of the band is determined by the Image Compression Manager and the `wantedPixelSize` field of the compressor capability structure (see Data Structures (page 105)). That field is filled in by the image compressor component's `ImageCodecPreCompress` or `ImageCodecPreDecompress` function. The Image Compression Manager requests the depth that it decides is best for the image, and the compressor component can return the `wantedPixelSize` field set to that depth or another appropriate depth if the compressor cannot handle the one requested.

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

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

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Compressing or Decompressing Images Asynchronously

With the appropriate hardware, image compressor components can handle asynchronous compression and decompression of images using the `ImageCodecBandCompress` and `ImageCodecBandDecompress` functions. *Asynchronous* refers to the fact that the compression or decompression hardware performs its operations while the computer simultaneously continues its other activities. For example, the computer can read a movie for the next frame while the current frame is decompressed. The Image Compression Manager ensures that any asynchronous operation in progress is completed before starting the next operation.

If the Image Compression Manager wants the image compressor component to perform an operation asynchronously, then the `completionProcRecord` field in the compression or decompression parameters structure that the Image Compressor Manager sends to the image compressor component should be set to a nonzero value. If the value is -1, then the component should perform the operation asynchronously, but it does not need to call a completion function. If the value is not `nil` and not -1, then the component should perform the operation asynchronously, and it should call the completion function when the operation is done. For details on the compression parameters structure, see Data Structures (page 105). For more on the decompression parameters structure, see Data Structures (page 105).

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

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

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

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

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

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Spooling of Compressed Data

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

Data Loading

Decompressor components use data loading. The data buffer still exists when the calling application supplies a data-loading function; however, the data buffer holds only part of the data and you must use the data-loading function to load the remaining data into this buffer. The `bufferSize` parameter of the decompression parameters structure indicates the size of the data buffer (see Data Structures (page 105)).

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

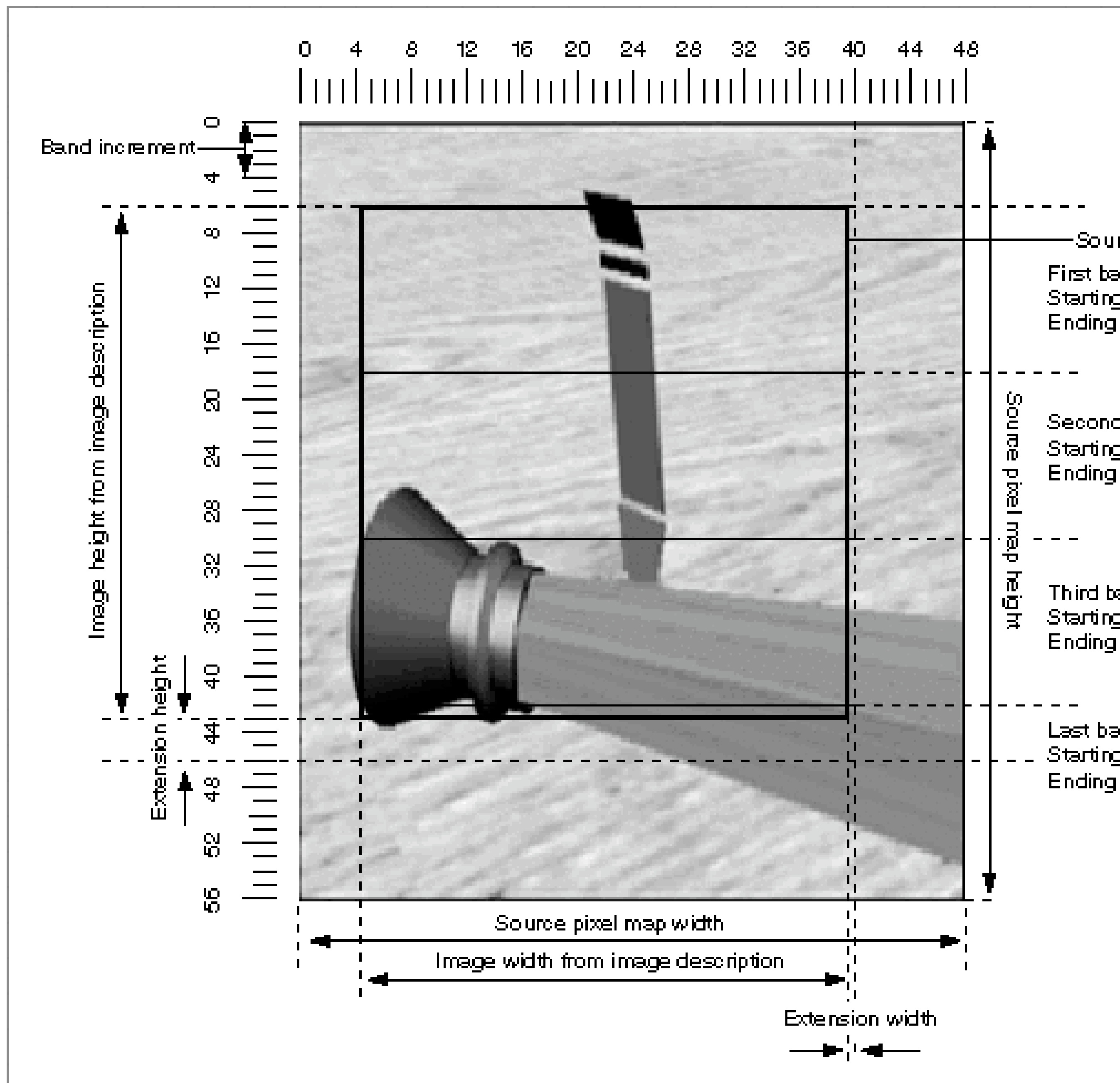
If the decompressor component needs to skip data in the compressed stream or go back to data earlier in the stream, the decompressor should call the data-loading function with a `nil` pointer (instead of the pointer to the data buffer of the data-loading function) and with the `size` parameter set to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

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

Figure 8-1 shows several image bands and their measurements.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 8-1 Image bands and their measurements



Data Unloading

Data-unloading functions are used by compressor components when there is insufficient memory to hold the buffer for the compressed data produced by the compressor component. The compressor component needs to use a data-unloading function if the `flushProcRecord` field in the compression parameters structure is not `nil`. (For details of the compression parameters structure, see Data Structures (page 105)). A

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

data buffer is provided even if the data-unloading function is present, and it should be used to hold the data to be unloaded by the data-unloading function. The size of the data buffer is indicated by the `bufferSize` field in the parameters.

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

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

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

Progress Functions

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

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

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

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Using Image Compressor Components

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

Performing Image Compression

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

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

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

Choosing a Compressor

Listing 9-1 shows how the Image Compression Manager calls the `ImageCodecPreCompress` function before an image is compressed. The compressor component returns information about how it is able to compress the image to the Image Compression Manager, so that it can fit the destination data to the requirements of the compressor component. This information includes compressor capabilities for

- depth of input pixels
- minimum buffer band size
- band increment size
- extension width and height

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

When your compressor component is called with the `ImageCodecPreCompress` function, it can handle all aspects of the function itself, or only the most common ones. All image compressor components must handle at least one case.

Listing 9-1 Preparing for simple compression operations

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

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

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

    capabilities->extendWidth = (*p->imageDescription)->width & 1;
    capabilities->extendHeight = (*p->imageDescription)->height & 1;
    /*
        For efficiency, if the compressor could perform extension,
        these flags would be set to 0.
    */
    return(noErr);
}
```

Here is a list of some of the operations your compressor component can perform during compression. It describes parameters in the compression parameters structure and indicates the operations that are required and which flags in the compressor capabilities flags field of the compressor capabilities structure must be set to allow your compressor to handle them (see Data Structures (page 105) and Data Structures (page 105)).

- **Depth conversion.** If your compressor component can compress from the pixel depth indicated by the `pixelSize` field (in the pixel map structure pointed to by the `srcPixmap` field of the compression parameters structure), it should set the `wantedPixelSize` field of the compressor capability structure to the same value. If it cannot handle that depth, it should specify the closest depth it can support in the `wantedPixelSize` field. The Image Compression Manager will convert the source image to that depth.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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

Compressing a Horizontal Band of an Image

Listing 9-2 shows how the Image Compression Manager calls the `ImageCodecBandCompress` function when it wants the compressor to compress a horizontal band of an image.

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

Listing 9-2 Performing simple compression on a horizontal band of an image

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

    /*
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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

    if (p->progressProcRecord.progressProc)
        p->progressProcRecord.progressProc (codecProgressOpen, 0,
            p->progressProcRecord.progressRefCon);
    width = (*desc)->width;
    height = (*desc)->height;
    depth = (*desc)->depth;
    dataStart = cDataPtr = p->data;
    /*
        Figure out offset to first pixel in baseAddr from the
        pixel size and bounds.
     */
    rowBytes = p->srcPixMap.rowBytes;
    sRect = p->srcPixMap.bounds;

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

    /*
        Adjust the source baseAddress to be at the beginning
        of the desired rect.
    */
    switch ( p->srcPixMap.pixelSize ) {
    case 32:
        offsetH = sRect.left<<2;
        break;
    case 16:
        offsetH = sRect.left<<1;
        break;
    case 8:
        offsetH = sRect.left;
        break;
    /*
        This compressor does not handle the other cases directly.
    */
    default:
        result = codecErr;
        goto bail;
    }
    offsetV = sRect.top * rowBytes;
    baseAddr = p->srcPixMap.baseAddr + offsetH + offsetV;
    /*
        If there is not a data-unloading function,
        adjust the pointer to the next band.
    */

    if ( p->flushProcRecord.flushProc == nil ) {
        cDataPtr += (p->startLine>>1) * stripBytes;
    }
    else { /*
                Make sure the compressor can deal with the
                data-unloading function in this case.
            */
        if ( p->bufferSize < stripBytes ) {
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
            result = codecSpoolErr;
            goto bail;
        }
    }
    /*
        Perform the slower data-loading or progress operation, as
        required.
    */

    if ( p->flushProcRecord.flushProc ||
        p->progressProcRecord.progressProc ) {
        SharedGlobals *sg = (*glob)->sharedGlob;
        for ( y=0; y < numStrips; y++) {
            SwapMMUMode(&mmuMode);
            CompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
            SwapMMUMode(&mmuMode);
            baseAddr += rowBytes<<1;
            if ( p->flushProcRecord.flushProc ) {
                if ( (result=
            p->flushProcRecord.flushProc(cDataPtr,stripBytes,
            p->flushProcRecord.flushRefCon)) != noErr) {
                    result = codecSpoolErr;
                    goto bail;
                }
            } else {
                cDataPtr += stripBytes;
            }
            if (p->progressProcRecord.progressProc) {
                if ( (result=
                    p->progressProcRecord.progressProc)
                        codecProgressUpdatePercent,
                        FixDiv(y,numStrips),
                        p->progressProcRecord.progressRefCon)
                )   != noErr ) {
                    result = codecAbortErr;
                    goto bail;
                }
            }
        }
    } else {
        SharedGlobals *sg = (*glob)->sharedGlob;
        short tRowBytes = rowBytes<<1;
        SwapMMUMode(&mmuMode);
        for ( y=numStrips; y--; ) {
            CompressStrip(cDataPtr,baseAddr,rowBytes,width,sg);
            cDataPtr += stripBytes;
            baseAddr += tRowBytes;
        }
        SwapMMUMode(&mmuMode);
    }
}
```

Decompressing an Image

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

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- The Image Compression Manager first determines which decompressor is best able to decompress the image. To do so, the Image Compression Manager examines the source image as well as the parameters specified by the application. If the application requested a specific decompressor, the Image Compression Manager uses that decompressor (unless it is not installed, in which case the Image Compression Manager returns an error to the application). If the application did not request a decompressor, the Image Compression Manager chooses the decompressor that will do the best job. The Image Compression Manager collects the information it needs to choose a decompressor by issuing the `ImageCodecPreDecompress` request to each qualifying decompressor.
- If the chosen decompressor can handle the image directly, the Image Compression Manager passes the request through to the decompressor. The decompressor then processes the image and returns the image to the specified location.
- If none of the decompressors can handle all of the conditions (matrix mapping, masking or matting, depth conversion, and so on) the Image Compression Manager allocates an offscreen buffer and passes image bands to the decompressor at a depth that the decompressor can handle by issuing a `ImageCodecBandDecompress` request. The decompressor processes each band, building the image as it goes. When the image has been completely decompressed, the Image Compression Manager returns control to the application.

Choosing a Decompressor

Listing 9-3 provides an example of how a decompressor is chosen. The Image Compression Manager calls the `ImageCodecPreDecompress` function before an image is decompressed. The decompressor returns information about how it can decompress an image. The Image Compression Manager can fit the destination pixel map to your decompressor's requirements if it is not able to support decompression to the destination directly. The capability information the decompressor returns includes

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

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

Listing 9-3 Preparing for simple decompression

```
pascal long ImageCodecPreDecompress( Handle storage,
                                 register CodecDecompressParams *p)
{
    register CodecCapabilities*capabilities = p->capabilities;
    RectdRect = p->srcRect;

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

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


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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

    switch ( (*p->imageDescription)->depth ) {
        case 16:
            break;
        default:
            return(codecConditionErr);
            break;
    }
        /*
            This decompressor can deal only with 32-bit pixels.
        */
    capabilities->wantedPixelSize = 32;

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

    capabilities->bandMin = 2;
    /* This decompressor can deal with 2 scan line high bands. */
    capabilities->bandInc = 2;

    /*
        If this decompressor needed its pixels be aligned on
        some integer multiple, you would set extendWidth and
        extendHeight to the number of pixels by which you need the
        destination extended. If you don't have such requirements
        or if you take care of them yourself, you set extendWidth
        and extendHeight to 0.
    */
    capabilities->extendWidth = p->srcRect.right & 1;
    capabilities->extendHeight = p->srcRect.bottom & 1;

    return(noErr);
}
```

Decompressor Operations

This section contains a bulleted list of some of the operations your decompressor component can perform during the decompression operation. The list describes which parameters in the decompression parameters structure indicate that the operations are required and which flags in the flags field of the compressor capabilities structure must be set to allow your decompressor to handle them (see Data Structures (page 105)).

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

Since your decompressor's capabilities depend on the full combination of parameters, it must inspect all the relevant parameters before indicating that it will perform one of the operations itself. For instance, if your decompressor has hardware that can perform scaling only if the destination pixel depth is 32 and there is no clipping, then the pre-decompression operation would have to check the following fields in the

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

decompression parameters structure: the `matrix` field, the `pixelSize` field of the destination pixel map structure pointed to by the `destPixMap` field, and the `maskBits` fields. Only then could the decompressor decide whether to set the `codecCanScale` flag in the `capabilities` field of the decompression parameters structure.

- **Scaling.** The decompressor component can look at the matrix and selectively decide which scaling operations it wishes to handle. If the scaling factor specified by the matrix is not unity and your decompressor can perform the scaling operation, it must set the `codecCanScale` flag in the `capabilities` field. If it does not, then the decompressor is asked to decompress without scaling, and the Image Compression Manager performs the scaling operation afterward.
- **Depth conversion.** If your component can decompress to the pixel depth indicated by the `pixelSize` field (of the pixel map structure pointed to by the `dstPixmap` field of the decompression parameters structure), it should set the `wantedPixelSize` field of the compressor capability structure to the same value. If it cannot handle that depth, it should specify the closest depth it can handle in the `wantedPixelSize` field.
- **Dithering.** When determining whether depth conversion can be performed (for converting an image to a lower bit depth, or to a similar bit depth with a different color table), dithering may be required. This is specified by the dither bit in the `transferMode` field (`0x40`) of the decompression parameters structure being set. The `accuracy` field of the decompression parameters structure indicates whether fast dithering is acceptable (`accuracy` less than or equal to `codecNormalQuality`) or whether true error diffusion dithering should be used (`accuracy` greater than `codecNormalQuality`). Most decompressors do not perform true error diffusion dithering, although they can. When a decompressor cannot perform the dither operation, it should specify the higher bit depth in the `wantedPixelSize` field of the compressor capability structure and let the Image Compression Manager perform the depth conversion with dithering. Dithering to 16-bit destinations is normally done only if the `accuracy` field is set to the `codecNormalQuality` value. However, if your decompressor component can perform dithering fast enough, it could be performed at the lower accuracy settings as well. To indicate that your decompressor can perform dithering as specified, it should set the `codecCanTransferMode` flag in the `capabilities` field of the decompression parameters structure.
- **Color remapping.** If the compressed data has an associated color lookup table that is different from the color lookup table of the destination pixel map, then the decompressor can remap the color indices to the closest available ones in the destination itself, or it can let the Image Compression Manager do the remapping. If the decompressor can do the mapping itself, it should set the `codecCanRemap` flag in the `capabilities` flags field of the decompression parameters structure.
- **Extending.** If the format for the compressed data is block-oriented, the decompressor can ask that the Image Compression Manager to allocate a buffer which is a multiple of the proper block size by setting the `extendWidth` and `extendHeight` fields of the compressor capabilities structure. If the right and bottom edges of the destination image (as determined by the transformed `srcRect` and `dstPixMap`.bounds fields of the decompression parameters structure) are not a multiple of the block size that your decompressor handles, and your decompressor cannot handle partial blocks (writing only the pixels that are needed for blocks that cross the left or bottom edge of the destination), then your decompressor component must set the `extendWidth` and `extendHeight` fields in the compressor capabilities structure. In this case, the Image Compression Manager creates a buffer large enough so that no partial blocks are needed. Your component can decompress into that buffer. This is then copied to the destination by the Image Compression Manager. Your component can avoid this extra step if it can handle partial blocks. In this case, it should leave the `extendWidth` and `extendHeight` fields set to 0.
- **Clipping.** If clipping must be performed on the image to be decompressed, the `maskBits` field of the decompression parameters structure is nonzero. In the `ImageCodecPreDecompress` function, it will be a region handle to the actual clipping region. If your decompressor can handle the clipping operation as specified by this region, it should set the `codecCanMask` bit in the `capabilities` flags field of the

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

decompression parameters structure. If it does this, then the parameter passed to the `ImageCodecBandDecompress` function in the `maskBits` field will be a `bitmap` instead of a region. If desired, your decompressor can save a copy of the actual region structure during the pre-decompression operation.

- **Matting.** If a matte must be applied to the decompressed image, the `transferMode` field of the decompression parameters structure is set to blend and the `mattePixMap` field is a handle to the pixel map to be used as the matte. If your decompressor can perform the matte operation, then it should set the `codecCanMatte` field in the compressor capabilities structure. If it does not, then the Image Compression Manager will perform the matte operation after the decompression is complete.
- **Pixel shifting.** For pixel sizes less than 8 bits per pixel, it may be necessary to shift the destination pixels so that they are at an aligned address. If the pixel size of the destination pixel map is less than 8 and your component handles that depth directly, and the left address of the image is not aligned and your component can handle these pixels directly, then it should set the `codecCanShift` flag in the `capabilities` field of the decompression parameters structure. If your component does not set this flag, the Image Compression Manager allocates a buffer for and performs the shifting after the decompression is completed.
- **Partial extraction.** If the source rectangle is not the entire image and the component can decompress only the part of the image specified by the source rectangle, it should set the `codecCanSrcExtract` flag in the `capabilities` field of the decompression parameters structure. If it does not, the Image Compression Manger asks the component to decompress the entire image and copy only the required part to the destination.

Decompressing a Horizontal Band of an Image

Listing 9-4 shows how to decompress the horizontal band of an image. The Image Compression Manager calls the `ImageCodecBandDecompress` function when it wants a decompressor to decompress an image or a horizontal band of an image. The pixel data indicated by the `baseAddr` field is guaranteed to conform to the criteria your decompressor specified in the `ImageCodecPreDecompress` function.

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

Listing 9-4 Performing a decompression operation

```
pascal long ImageCodecBandDecompress( Handle storage,
                                register CodecDecompressParams *p)
{
    Rect                dRect;
    long                offsetH,offsetV;
    Globals             **glob = (Globals **)storage;
    long                numLines,numStrips;
    short               rowBytes;
    long                stripBytes;
    short               width;
    register short      y;
    register char*      baseAddr;
    char                *cDataPtr;
    char                mmuMode = 1;
    OSErr               result = noErr;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
/*
    Calculate the real base address based on the boundary
    rectangle. If it's not a linear transformation, this
    decompressor does not perform the operation.
*/
dRect = p->srcRect;
if ( !TransformRect(p->matrix,&dRect,nil) )
    return(paramErr);
/*  If there is a progress function, give it an open call at
    the start of this band.
*/
if (p->progressProcRecord.progressProc)
    p->progressProcRecord.progressProc(codecProgressOpen,0,
        p->progressProcRecord.progressRefCon);

/*
    Initialize some local variables.
*/

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

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

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

/*
    If your decompressor component is skipping some data,
    it just skips it here. You can tell because
    firstBandInFrame indicates this is the first band for a new
    frame, and if startLine is not 0, then that many lines were
    clipped out.
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
     */
    if ( (p->conditionFlags & codecConditionFirstBand) &&
            p->startLine != 0 ) {
        if ( p->dataProcRecord.dataProc ) {
            for ( y=0; y < p->startLine>>1; y++ ) {
                if ( (result=p->dataProcRecord.dataProc
                         (&cDataPtr,stripBytes,
                        p->dataProcRecord.dataRefCon)) != noErr ) {
                    result = codecSpoolErr;
                    goto bail;
                }
                cDataPtr += stripBytes;
            }
        } else
            cDataPtr += (p->startLine>>1) * stripBytes;
    }
    /*
        If there is a data-loading function spooling the data to your
        decompressor, then you have to decompress the data in the
        chunk size that is specified, or, if there is a progress
        function, you must make sure to call it as you go along.
    */

    if ( p->dataProcRecord.dataProc ||
         p->progressProcRecord.progressProc ) {
        SharedGlobals *sg = (*glob)->sharedGlob;

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

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

/*
    Otherwise, do the fast case.
*/
    } else {
        SharedGlobals *sg = (*glob)->sharedGlob;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
        shorttRowBytes = rowBytes<<1;

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

    if ( p->conditionFlags & codecConditionLastBand ) {
        /*
            Tie up any loose ends on the last band of the frame.
        */
    }
bail:
    /*
        If there is a progress function, give it a close call
        at the end of this band.
    */
    if (p->progressProcRecord.progressProc)
        p->progressProcRecord.progressProc(codecProgressClose,0,
            p->progressProcRecord.progressRefCon);
    return(result);
}
```

Asynchronous Decompression

The Image Compression Manager (ICM) supports scheduled asynchronous decompression operations. By calling the Image Compression Manager function `DecompressSequenceFrameWhen`, applications can schedule decompression requests in advance. This allows decompressor components that support this functionality to provide reliable playback performance under a wider range of conditions.

The Apple Cinepak, Video, Animation, Component Video, and Graphics decompressors provided in QuickTime support scheduled asynchronous decompression to 8-, 16-, and 32-bit destinations (the Cinepak decompressor also supports 4-bit grayscale destinations). QuickTime also adds asynchronous decompression support to the JPEG and None decompressor components on PowerPC systems (with the QuickTime PowerPlug extension installed).

If you want to support this functionality, you must modify your decompressor component in the following ways:

- Report your component's new capabilities in its compressor capability structure by setting the `codecCanAsyncWhen` and `codecCanAsync` flags.
- Modify your component's `ImageCodecFlush` function to accept scheduled asynchronous decompression requests and process them correctly.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- Implement the new function `ImageCodecFlush`; this function allows the Image Compression Manager to instruct you to empty your input queue.
- Optionally, implement logic to manage the shielding of the cursor during decompression operations.

Hardware Cursors

The Image Compression Manager supports hardware cursors introduced in PCI-based Macintosh computers, which eliminates cursor flicker. For all software codecs, this support requires no changes.

For codecs that manage the cursor themselves, QuickTime has a flag, `codecCompletionDontUnshield`, for use when calling the `ICMDecompressComplete` function. Use this flag to prevent the Image Compression Manager from unshielding the cursor when `ICMDecompressComplete` is called.

Timecode Support

QuickTime provides timecode information to decompressor components when movies are played. To support timecodes, your codec must support the function `ImageCodecSetTimeCode`, which allows the Image Compression Manager to set the timecode value for the next frame to be decompressed.

Working With Video Fields

The functionality of the `ImageFieldSequenceExtractCombine` function is performed by individual image codecs. This is because the way in which fields are stored is different for every compression format. A codec component function, `ImageCodecExtractAndCombineFields`, is defined for this purpose. Apple encourages developers of codecs to incorporate this function, if their compressed data format is capable of separately storing both fields of a video frame.

Accelerated Video Support

QuickTime supports codecs that accelerate certain image decompression operations. These features are most likely used by developers of video hardware boards that provide special acceleration features, such as arbitrary scaling or color space conversion.

If a codec cannot decompress directly to the screen it has the option of specifying that it can decompress to one or more types of non-RGB pixel spaces, specified as an `OSType` (e.g., `'yuvs'`). The ICM then attempts to find a decompressor component of that type (a transfer codec) that can transfer the image to the screen. Since the ICM does not define non-RGB pixel types, the transfer codec must support additional calls to set up the offscreen. If a transfer codec cannot be found that supports the specified non-RGB pixel types, the ICM uses the None codec with an RGB offscreen buffer.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

The real speed benefit comes from the fact that since the transfer codec defines the offscreen buffer, it can place the buffer in on-board memory, or even point to an overlay plane so that the offscreen image really is on the screen. In this case, the additional step of transferring the bits from offscreen memory on to the screen is avoided.

For an image decompressor component to indicate that it can decompress to non-RGB pixel types, it should, in the `ImageCodecPreDecompress` call, fill in the `wantedDestinationPixelTypes` field with a handle to a zero-terminated list of pixel types that it can decompress to. The ICM immediately makes a copy of the handle. Cinepak, for example, returns a 12-byte handle containing `yuvs`, `yuvu`, and $00000000. Since `ImageCodecPreDecompress` can be called often, it is suggested that codecs allocate this handle when their component is opened and simply fill in the `wantedDestinationPixelTypes` field with this handle during `ImageCodecPreDecompress`. Components that use this method should be sure to dispose the handle at close.

Apple's Cinepak decompressor supports decompressing to `'yuvs'` and `'yuvu'` pixel types. Type `'yuvs'` is a YUV format with u and `v` components signed (center point at $00), while `'yuvu'` has the u and `v` component centered at $80.

As an example, suppose XYZ Co. had a video board that had a YUV overlay plane capable of doing arbitrary scaling. The overlay plane takes data in the same format as Cinepak's `'yuvs'` format. In this case, XYZ would make a component of type `'imdc'` and subtype `'yuvs'`.

The `ImageCodecPreDecompress` call would set the `codecCanScale`, `codecHasVolatileBuffer`, and `codecImageBufferIsOnScreen` bits in the `capabilities flags` field. The `codecImageBufferIsOnScreen` bit is necessary to inform the ICM that the codec is a direct screen transfer codec. A direct screen transfer codec is one that sets up an offscreen buffer that is actually onscreen (such as an overlay plane). Not setting this bit correctly can cause unpredictable results.

The real work of the codec takes place in the `ImageCodecNewImageBufferMemory` call. This is where the codec is instructed to prepare the non-RGB pixel buffer. The codec must fill in the `baseAddr` and `rowBytes` fields of the `dstPixMap` structure in `CodecDecompressParams`. The ICM then passes these values to the original codec (e.g., Cinepak) to decompress into.

The codec must also implement `ImageCodecDisposeMemory` to balance `ImageCodecNewImageBufferMemory`.

Since Cinepak then decompresses into the card's overlay plane, `ImageCodecBandDecompress` needs to do nothing aside from calling `ICMDecompressComplete`.

```
pascal ComponentResult
ImageCodecPreDecompress(Handle storage,
        CodecDecompressParams *p)
{
    CodecCapabilities   *capabilities = p->capabilities;
    // only allow 16 bpp source
    if ((**p->imageDescription).depth != 16)
        return codecConditionErr;
    /* we only support 16 bits per pixel dest */
    if (p->dstPixMap.pixelSize != 16)
        return codecConditionErr;

    capabilities->wantedPixelSize = p->dstPixMap.pixelSize;

    capabilities->bandInc = capabilities->bandMin =
                (*p->imageDescription)->height;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    capabilities->extendWidth = 0;
    capabilities->extendHeight = 0;

    capabilities->flags =
            codecCanScale | codecImageBufferIsOnScreen |
            codecHasVolatileBuffer;

    return noErr;
}

pascal ComponentResult

ImageCodecBandDecompress(Handle storage,
        CodecDecompressParams *p)
{
    ICMDecompressComplete(p->sequenceID, noErr,
                codecCompletionSource | codecCompletionDest,
                &p->completionProcRecord);

    return noErr;
}

pascal ComponentResult
ImageCodecNewImageBufferMemory(Handle storage,
        CodecDecompressParams *p, long flags,
        ICMMemoryDisposedUPP memoryGoneProc,
        void *refCon)
{
    OSErr err = noErr;
    long offsetH, offsetV;
    Ptr baseAddr;
    long rowBytes;

    // call predecompress to check to make sure we can handle
    // this destination
    err = ImageCodecPreDecompress(storage, p);
    if (err) goto bail;

    // set video board registers with the scale
    XYZVideoSetScale(p->matrix);

    // calculate a base address to write to
    offsetH = (p->dstRect.left - p->dstPixMap.bounds.left);
    offsetV = (p->dstRect.top - p->dstPixMap.bounds.top);
    XYZVideoGetBaseAddress(p->dstPixMap, offsetH, offsetV,
                &baseAddr, &rowBytes);

    p->dstPixMap.baseAddr = baseAddr;
    p->dstPixMap.rowBytes = rowBytes;
    p->capabilities->flags = codecImageBufferIsOnScreen;
bail:
    return err;
}

pascal ComponentResult
ImageCodecDisposeMemory(Handle storage, Ptr data)
{
    return noErr;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
}
```

Some video hardware boards that use an overlay plane require that the image area on screen be flooded with a particular RGB value or alpha-channel in order to have the overlay buffer "show through" at that location. Codecs that require this support should set the `screenFloodMethod` and `screenFloodValue` fields of the `CodecDecompressParams` record during `ImageCodecPreDecompress`. The ICM then manages the flooding of the screen buffer. This method is more reliable than having the codec attempt to flood the screen itself, and will ensure compatibility with future versions of QuickTime.

Packetization Information

QuickTime functions support packetizing compressed data streams, primarily for video conferencing applications. For this purpose, the field `preferredPacketSizeInBytes` was added to the compression parameters structure. Codec developers need only use this field.

Packet information is appended, word-aligned, to the end of video data. It is a variable-length array of 4-byte integers, each representing the offset in bits of the end of a packet, followed by another integer containing the number of packet hints, and finally a four-byte identifier indicating the type of appended data:

```
[boundary #1][boundary #2]...[boundary #N][N]['pkts']
```

Packets are given in bits, because some types of compressed image data (such as H.261) are cut up on bit-boundaries rather than byte-boundaries.

```
// given:  image data, length, and a packet number
// returns: a pointer to the start of the packet and a packet size, plus
// information about leading and trailing bits

char* GetNextPacket(char* data, int len, int packet, long* packet_size,
    char* leading_bits, char* trailing_bits)
{
    long *lp, packets;
    lp = (long*) data;  // 'data' must be word-aligned
    lp += len/4 - 1;
    if (*lp != 'pkts')
        return nil;

    packets = *lp[ -1 ];           // negative indexing is good for you
    if (packet >= packets)
        return nil;             // out of bounds
    lp -= packets;      // now 0-indexing into the packet array will work
    if (packet == 0)
    {
        *packet_size = (lp[0] + 7)/8;   // count the bits
        *leading_bits = 0;
        *trailing_bits = lp[0] % 8;
        return data;                    // in case of 0-length packet
    }
    else
    {
        *packet_size = ( lp[pktnum] - lp[pktnum-1] + 7) / 8;
        *leading_bits = lp[packet-1] % 8 ? 8 - lp[packet-1] % 8 : 0;
        *trailing_bits = lp[packet] % 8;
        return data + lp[packet-1] / 8;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    }
}
```

Note that this technique can be used for further extensions by the addition of further appended formats. The last two words are always the number of words and an extension identifier.

DV Image Compressor Component

The DV image compressor component makes it possible to compress QuickTime video data into DV format. It is invoked automatically by the Image Compression Manager when an application requests output of type `kDVCNTSCCodecType` for NTSC DV data or `kDVCPALCodecType` for PAL DV data.

When creating NTSC video, the DV image compressor component generates 720 X 480 frames. When creating PAL video, it generates 720 X 576 frames.

> **Note:** Many DV devices use IEEE 1394 (FireWire) serial connections for input/output operations. QuickTime supports compression and decompression of DV data, but it does not include support for FireWire communication. You need additional software to communicate with DV devices.

DV Image Decompressor Component

The DV image decompressor component makes it possible to decompress DV video data. It is invoked automatically by the Image Compression Manager when an application specifies input of type `kDVCNTSCCodecType` for NTSC DV data or `kDVCPALCodecType` for PAL DV data.

There are two quality modes for DV decompression:

- In the low-quality mode, which is the default, the DV image decompressor component generates a 1/4-screen image. When operating in this mode, the component uses approximately 25% of the video data in the DV stream and correspondingly fewer system resources.
- In the high-quality mode, the component processes all of the video data in the DV stream. Applications can specify the high-quality modem by calling the `SetMediaPlayHints` function with the `hintsHighQuality` flag set.

When a computer includes a video display adapter that performs YUV decompression in hardware, the DV image decompressor can use a YUV decompressor component written to use the hardware decompression capabilities in place of the software YUV decompressor in QuickTime, resulting in even higher performance.

Specifying the Size of an Image Buffer

You can specify the size of the image buffer used by your image compressor or decompressor component. When your component calls the `ImageCodecPreDecompress` or `ImageCodecPreCompress` function, you can specify the size of the buffer as follows:

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- In the `CodecDecompressParams` or `CodecCompressParams` record, set the `codecWantsSpecialScaling` flag in the `flags` field of the `CodecCapabilities` record.
- Provide values for the `requestedBufferWidth` and `requestedBufferHeight` fields in the `CodecDecompressParams` or `CodecCompressParams` record.

This is illustrated in Listing 9-5.

Listing 9-5 Specifying the size of an image buffer for a codec

```
p->capabilities->flags |= codecWantsSpecialScaling;
p->requestedBufferWidth = 720;
p->requestedBufferHeight = 480;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Codec Components API

This chapter lists the data structures and functions that support codec components in QuickTime.

Data Structures

The following data structures are defined in the *QuickTime API Reference:*

- The `ICMFrameTimeRecord` data type defines the frame time structure, which contains a frame's time information for scheduled asynchronous decompression operations.
- The `CDSequenceDataSource` data type defines the decompression data source structure. This contains a linked list of all data sources for a decompression sequence. Because each data source contains a link to the next data source, a codec can access all data sources from this structure.
- The `CodecCapabilities` data type defines the compressor capability structure. Image compressor components use the compressor capability structure to report their capabilities to the Image Compression Manager. Before compressing or decompressing an image, the Image Compression Manager requests this capability information from the component that will be handling the operation by calling the `ImageCodecPreCompress` or `ImageCodecPreDecompress` function provided by that component. The compressor component examines the compression or decompression parameters and indicates any restrictions on its ability to satisfy the request in a formatted compressor capability structure. The Image Compression Manager then manages the operation according to the capabilities of the component.
- The `CodecDecompressParams` data type defines the decompression parameters structure. Decompressors accept the parameters that govern a decompression operation in this data structure. It is used by the `ImageCodecBandDecompress` and `ImageCodecPreDecompress` functions.
- The `CodecCompressParams` data type defines the compression pParameters structure. Compressor components accept the parameters that govern a compression operation in this data structure. This structure is used by the `ImageCodecBandCompress` and `ImageCodecPreCompress` functions.

Functions

This section lists the functions that image compressor components must support. It also lists the utility functions that the Image Compression Manager provides for use by compressors and decompressors. For details of these functions, see the *QuickTime API Reference*.

The function list is divided into two parts. Direct Functions (page 106) lists image compressor component functions that are called by the Image Compression Manager in response to application requests. Indirect Functions (page 106) lists image compressor component functions that may be called by the Image Compression Manager at any time. The next section, Image Compression Manager Utility Functions (page 107) lists Image Compression Manager utility functions that are available to image compressor components.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

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

```
#define kImageCodecGetCodecInfoSelect           0x00
#define kImageCodecGetCompressionTimeSelect     0x01
#define kImageCodecGetMaxCompressionSizeSelect  0x02
#define kImageCodecPreCompressSelect            0x03
#define kImageCodecBandCompressSelect           0x04
#define kImageCodecPreDecompressSelect          0x05
#define kImageCodecBandDecompressSelect         0x06
#define kImageCodecBusySelect                   0x07
#define kImageCodecGetCompressedImageSizeSelect 0x08
#define kImageCodecGetSimilaritySelect          0x09
#define kImageCodecTrimImageSelect              0x0A
```

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

Direct Functions

These functions are invoked by the Image Compression Manager in direct response to application functions:

- ImageCodecGetCodecInfo
- ImageCodecGetMaxCompressionSize
- ImageCodecGetCompressionTime
- ImageCodecGetSimilarity
- ImageCodecGetCompressedImageSize
- ImageCodecTrimImage
- ImageCodecBusy

Indirect Functions

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

- ImageCodecPreCompress
- ImageCodecBandCompress
- ImageCodecPreDecompress
- ImageCodecBandDecompress

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Image Compression Manager Utility Functions

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

- `SetImageDescriptionExtension`
- `GetImageDescriptionExtension`
- `RemoveImageDescriptionExtension`
- `CountImageDescriptionExtensionType`
- `GetNextImageDescriptionExtensionType`
- `ICMShieldSequenceCursor`
- `ICMDecompressComplete`

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

About the Base Image Decompressor

This chapter describes the features of the base image decompressor. The base image decompressor is an Apple-supplied component that makes it easier for developers to create new decompressors. The base image decompressor does most of the housekeeping and interface functions required for a QuickTime decompressor component, including scheduling for asynchronous decompression.

Whenever possible, an image decompressor component should handle asynchronous requests for decompression, as described in Asynchronous Decompression (page 98). If you are implementing an image decompressor component, you can include this capability with a minimum of additional programming by using the services of the base image decompressor. The base image decompressor handles the necessary scheduling, which frees you to concentrate on the details of decompression.

When you use the base image decompressor with an image decompressor component, your component must support functions that are called by the base image decompressor when necessary. Your component can then delegate a number of other function calls to the base image decompressor, which greatly simplifies the implementation of your component.

Using the Base Image Decompressor

To use the services of the base image decompressor, your image decompressor component must support functions that the base image decompressor calls when necessary. The following sections explain when the base image decompressor calls these functions and how your image decompressor component must respond. These sections also list standard image decompressor calls that your image decompressor must handle itself rather than delegate.

The base image decompressor and image decompressor components are managed by the Component Manager.

Connecting to the Base Image Decompressor

To use the services of the base image decompressor, your image decompressor component must open a connection to the base image decompressor component. Listing 11-1 illustrates how to make the connection.

Listing 11-1 Connecting to the base image decompressor component

```
ComponentInstance baseCodec;
OSErr err;
err = OpenADefaultComponent (decompressorComponentType,
                             kBaseCodecType,
                             &baseCodec);
err = ComponentSetTarget (baseCodec,
                          self);
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Providing Storage for Frame Decompression

Your image decompressor component uses an `ImageSubCodecDecompressRecord` structure to store information needed to decompress a single frame. The structure is created by the base decompressor component when your component is initialized, as described in Initializing Your Decompressor Component (page 110).

Initializing Your Decompressor Component

The first function call that your image decompressor component receives from the base image decompressor is always a call to `ImageCodecInitialize`. In response to this call, your image decompressor component returns an `ImageSubCodecDecompressCapabilities` structure that specifies its capabilities. This structure contains the following fields:

- `canAsync` specifies whether the component can support asynchronous decompression operations, as described in Asynchronous Decompression (page 98).
- `decompressRecordSize` specifies the size of the `ImageSubCodecDecompressRecord` structure that the base decompressor component creates for your image decompressor component.

With the help of the base image decompressor, any image decompressor that uses only interrupt-safe calls for decompression operations can support asynchronous decompression.

Listing 11-2 shows how to specify that a decompressor supports asynchronous decompression operations.

Listing 11-2 Specifying the capabilities of a decompressor component.

```
ImageSubCodecDecompressCapabilities deccap;
ImageSubCodecDecompressRecord decrec;
deccap->decompressRecordSize = sizeof(decrec);
deccap->canAsync = true;
```

Specifying Other Capabilities of Your Component

The base image decompressor gets additional information about the capabilities of your image decompressor component by calling your component's `ImageCodecPreflight` function. The base image decompressor uses this information when responding to a call to the `ImageCodecPredecompress` function, which the Image Compression Manager makes before decompressing an image.

Your image decompressor component returns information about its capabilities by filling in the `capabilities` structure. Listing 11-3 illustrates how to fill in this structure. In this example, the decompressor component specifies that it supports the ARGB, ABGR, BGRA, and RGBA pixel formats used by Microsoft Windows.

Listing 11-3 Sample implementation of ImageCodecPreflight

```
pascal ComponentResult ImageCodecPreflight (
                        ComponentInstance ci,
                        CodecDecompressParams *p)
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
{
    register CodecCapabilities*capabilities = p->capabilities;
    /*  Decide which depth compressed data we can deal with. */

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

    /*  We can deal only 32 bit pixels. */
    capabilities->wantedPixelSize = 32;

    /*  The smallest possible band we can do is 2 scan lines. */

    capabilities->bandMin = 2;
    /*  We can deal with 2 scan line high bands. */
    capabilities->bandInc = 2;

    /*  If we needed our pixels to be aligned on some integer
     *  multiple we would set these to
     *  the number of pixels we need the dest extended by.
     *  If we dont care, or we take care of
     *  it ourselves we set them to zero.
     */
    capabilities->extendWidth = p->srcRect.right & 1;
    capabilities->extendHeight = p->srcRect.bottom & 1;
    {
        OSType *pf = *glob->wantedDestinationPixelTypeH;
        p->wantedDestinationPixelTypes =
            glob->wantedDestinationPixelTypeH;
        // set up default order
        pf[0] = k32BGRAPixelFormat;
        pf[1] = k32ARGBPixelFormat;
        pf[2] = k32ABGRPixelFormat;
        pf[3] = k32RGBAPixelFormat;
        switch (p->dstPixMap.pixelFormat) {
        case k32BGRAPixelFormat: // we know how to do these pixel formats
            break;
        case k32ABGRPixelFormat:
            pf[0] = k32ABGRPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ARGBPixelFormat;
            pf[3] = k32RGBAPixelFormat;
            break;
        case k32ARGBPixelFormat:
            pf[0] = k32ARGBPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ABGRPixelFormat;
            pf[3] = k32RGBAPixelFormat;
            break;
        case k32RGBAPixelFormat:
            pf[0] = k32RGBAPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ARGBPixelFormat;
            pf[3] = k32ABGRPixelFormat;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
            break;
        default:            // we don't know how to do these, so return
                            // the default
            break;
        }
    }
    return(noErr);
}
```

Implementing Functions for Queues

If the image decompressor component supports asynchronous scheduled decompression, it receives a `ImageCodecQueueStarting` call from the base image decompressor when processing of the queue begins and the `ImageCodecQueueStopping` function when processing of the queue is finished. It is not necessary for your image decompressor component to implement these functions. Implement them only if there are tasks that your image decompressor component must perform after being notified, such as locking structures in memory before `ImageCodecDrawBand` is called.

Calls to `ImageCodecQueueStarting` and `ImageCodecQueueStopping` are never made during interrupt time.

Decompressing Bands

Your image decompressor component must implement the `ImageCodecBeginBand` and `ImageCodecDrawBand` functions for decompressing bands. It can also implement the `ImageCodecEndBand` function to be information that decompression of a band is complete. It receives these calls from the base image decompressor when decompression of either a complete frame or an individual band needs to be performed.

Implementing ImageCodecBeginBand

The `ImageCodecBeginBand` function allows your image decompressor component to save information about a band before decompressing it. For example, your image decompressor component can change the value of the `codecData` pointer if not all of the data for the band needs to be decompressed. The base image decompressor preserves any changes your image decompressor component makes to any of the fields in the `ImageSubCodecDecompressRecord` or `CodecDecompressParams` structures.

The `ImageCodecBeginBand` function is never called at interrupt time. If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

A sample implementation of `ImageCodecBeginBand` is shown in Listing 11-4.

Listing 11-4 Sample implementation of ImageCodecBeginBand

```
pascal ComponentResult ImageCodecBeginBand (
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
                        ComponentInstance ci,
                        CodecDecompressParams *p,
                        ImageSubCodecDecompressRecord *drp,
                        long flags)
{
    ExampleDecompressRecord *mydrp = drp->userDecompressRecord;
    long            numLines,numStrips;
    long            stripBytes;
    short           width;
    short           y;
    OSErr           result = noErr;
    Ptr             cDataPtr;

    /* initialize some local variables */

    width = (*p->imageDescription)->width;
    numLines = p->stopLine - p->startLine;  /* number of scanlines in */
                                            /* this band */
    numStrips = (numLines+1)>>1;    /* number of strips in this band */
    stripBytes = ((width+1)>>1) * 5;    /* number of bytes in one */
                                        /* strip of blocks */
    cDataPtr = drp->codecData;

    /*
     *  If skipping some data, just skip it here. We can tell because
     *  firstBandInFrame says this is the first band for a new frame, and
     *  if startLine is not zero, then that many lines were clipped out.
     */
    if ( (p->conditionFlags & codecConditionFirstBand) && p->startLine != 0 )
{
        if ( p->dataProcRecord.dataProc ) {
            for ( y=0; y < p->startLine>>1; y++ ) {
                if (
(result=CallICMDataProc(p->dataProcRecord.dataProc,&cDataPtr,stripBytes,
                        drp->dataProcRecord.dataRefCon)) != noErr ) {
                    result = codecSpoolErr;
                    goto bail;
                }
                cDataPtr += stripBytes;
            }
        } else
            cDataPtr += (p->startLine>>1) * stripBytes;
    }

    drp->codecData = cDataPtr;
    mydrp->width = width;
    mydrp->numStrips = numStrips;
    mydrp->srcDataIncrement = stripBytes;
    mydrp->baseAddrIncrement = drp->rowBytes<<1;
    mydrp->glob = (void *)storage;
    /* figure out our dest pixel format and select the
       correct DecompressStripProc */
    switch(p->dstPixMap.pixelFormat) {
        case 0:     // old case where planebytes
                    // is not set by codecmanager
        case k32ARGBPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32ARGB;
            break;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
        case k32ABGRPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32ABGR;
            break;
        case k32BGRAPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32BGRA;
            break;
        case k32RGBAPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32RGBA;
            break;
        default:
            bail;
            break;
    }
bail:
    return result;
}
```

Implementing ImageCodecDrawBand

When the base image decompressor calls your image decompressor component's `ImageCodecDrawBand` function, your component must perform the decompression specified by the fields of the `ImageSubCodecDecompressRecord` structure. The structure includes any changes your component made to it when performing the `ImageCodecBeginBand` function.

If the `ImageSubCodecDecompressRecord` structure specifies either a progress function or a data-loading function, the base image decompressor never calls the `ImageCodecDrawBand` function at interrupt time. If not, the base image decompressor may call the `ImageCodecDrawBand` function at interrupt time.

If the `ImageSubCodecDecompressRecord` structure specifies a progress function, the base image decompressor handles `codecProgressOpen` and `codecProgressClose` calls, and your image decompressor component must not implement these functions.

If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

A sample implementation of `ImageCodecDrawBand` is shown in Listing 11-5.

Listing 11-5 Sample implementation of ImageCodecDrawBand

```
pascal ComponentResult ImageCodecDrawBand (
                          ComponentInstance ci,
                          ImageSubCodecDecompressRecord *drp)
{
    ExampleDecompressRecord *mydrp = drp->userDecompressRecord;
    short y;
    Ptr cDataPtr = drp->codecData;  // compressed data pointer;
    Ptr baseAddr = drp->baseAddr;   // base address of dest PixMap;
    SInt8 mmuMode = true32b;        // we want to be in 32-bit mode
    OSErr err = noErr;
    for (y = 0; y < mydrp->numStrips; y++) {
        if (drp->dataProcRecord.dataProc) {
            if ( (err =
CallICMDataProc(drp->dataProcRecord.dataProc,&cDataPtr,
                        mydrp->srcDataIncrement,
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
                drp->dataProcRecord.dataRefCon)) != noErr ) {
                err = codecSpoolErr;
                goto bail;
            }
        }
        SwapMMUMode(&mmuMode);      // put us in 32-bit mode
(mydrp->decompressStripProc)(cDataPtr,baseAddr,(short)drp->rowBytes,
                            (short)mydrp->width,glob->sharedGlob);
        SwapMMUMode(&mmuMode);      // put us back
        baseAddr += mydrp->baseAddrIncrement;
        cDataPtr += mydrp->srcDataIncrement;
        if (drp->progressProcRecord.progressProc) {
            if ( (err =
CallICMProgressProc(drp->progressProcRecord.progressProc,
                codecProgressUpdatePercent,
                FixDiv ( y, mydrp->numStrips),
                drp->progressProcRecord.progressRefCon)) != noErr ) {
                err = codecAbortErr;
                goto bail;
            }
        }
    }
bail:
    return err;
}
```

Implementing ImageCodecEndBand

Your image decompressor component is not required to implement the `ImageCodecEndBand` function. If it does, the base image decompressor calls the function when the decompression of a band is complete or is terminated by the Image Compression Manager. The call simply notifies your component that decompression is finished. After your component handles the call, it can perform any tasks that are necessary when decompression is finished, such as disposing of data structures that are no longer used, after receiving notification. Note that because the `ImageCodecEndBand` function can be called at interrupt time, your image decompressor component cannot use this function to dispose of data structures; this must occur after handling the function.

Providing Information About the Decompressor

Your image decompressor component must also implement the `ImageCodecGetCodecInfo`. This performs the same task as the `CDGetCodecInfo` function. The Image Compression Manager calls your image decompressor component's `ImageCodecGetCodecInfo` function when it receives a `GetCodecInfo` call.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Providing Progress Information

If the `ImageSubCodecDecompressRecord` structure does not specify a progress function, your image decompressor component can implement `codecProgressOpen` and `codecProgressClose` functions to provide progress information. If the `ImageSubCodecDecompressRecord` structure does specify a progress function, your image decompressor component can implement the `codecProgressUpdatePercent` function to provide progress information during lengthy decompression operations. Implementing this function is optional.

Handling and Delegating Other Calls

Your image decompressor component must delegate the following image decompressor component calls to the base image decompressor:

- `ImageCodecPreDecompress`
- `ImageCodecBandDecompress`
- `ImageCodecBusy`
- `ImageCodecFlush`

If the `ImageSubCodecDecompressRecord` structure specifies a progress function, your image decompressor component must also delegate these decompressor component calls to the base image decompressor:

- `codecProgressOpen`
- `codecProgressClose`

Your image decompressor component can implement any other image decompressor component functions itself or delegate any of the calls to the base image decompressor. To delegate calls, it uses the `DelegateComponentCall` function.

Closing the Component

When your image decompressor component closes, it must close its connection to the base image decompressor by calling the `CloseComponent` function.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Using Data Codec Components

This chapter discusses how to use data codec components to compress or decompress data. A list of functions provided by data codec components is also included.

Data codecs enable you to compress and decompress data that is not automatically handled by QuickTime media operations. For example, QuickTime automatically uses image and sound codecs to compress and decompress video and sound tracks, but does not automatically compress or decompress sprites.

Data codecs are useful for compressing and decompressing sprites, 3D models, or other data types whose media handlers do not inherently support compression.

Data codecs also enable you to compress or decompress arbitrary blocks of data from other sources. The data does not necessarily need to come from, or go to, a QuickTime movie.

Data codecs are divided into compressor and decompressor components. All data compressors have a component type of `DataCompressorComponentType` and all data decompressors have a component type of `DataDecompressorComponentType`. The compression algorithm is indicated by the component subtype.

Once you have selected a data codec component, you can compress or decompress from one buffer to another. With care, this can be done at interrupt time.

Component Types

Data compressor and decompressor components have component types of `DataCompressorComponentType` and `DataDecompressorComponentType`. Each component has a unique component subtype, indicating the type of compression algorithm it supports.

Select an appropriate data codec component for your data using the Component Manager functions, such as the `OpenADefaultComponent` or `FindNextComponent` functions.

Prior to compressing or decompressing data, you need to create a buffer containing the source data and allocate a buffer to receive the destination data. You can use the `DataCodecGetCompressBufferSize` function to determine the size of destination buffer you need to allocate before doing a compression or decompression.

The `DataCodecCompress` function enables you to compress data using a specified compressor component. Similarly, the `DataCodecDecompress` function enables you decompress data using a specified decompressor component.

If a compressor or decompressor component implements the `DataCodecBeginInterruptSafe` and `DataCodecEndInterruptSafe` functions, your application or other software can perform compression or decompression operations during interrupt time. You do this as follows:

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

1. Before performing the compression or decompression operation, call the `DataCodecBeginInterruptSafe` function. In the call, pass the maximum size of a data block to be compressed or decompressed in the `maxSrcSize` parameter.

2. If the call fails, do not perform compression or decompression operations during interrupt time. Otherwise, you may proceed.

3. When the compression or decompression operation is complete, call `DataCodecEndInterruptSafe` to release resources used to make the operation safe at interrupt time.

Functions

The following functions are provided by data codec components:

- `DataCodecBeginInterruptSafe` allocates any temporary buffers needed to perform compression or decompression during interrupt. Returns an error if operations are not interrupt-safe.
- `DataCodecCompress` compresses data.
- `DataCodecDecompress` decompresses data.
- `DataCodecEndInterruptSafe` releases resources used to conduct compression or decompression operations during interrupt.
- `DataCodecGetCompressBufferSize` returns the maximum possible size of the compressed data that will be returned using the specified compressor component.
- `DataCodecCompressPartial` allows you to feed a large block of uncompressed data into the compressor in chunks.
- `DataCodecDeompressPartial` allows you to decompress a large block of data in a series of smaller pieces.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Standard Image Compression Dialog Components

This chapter introduces the standard image compression dialog component and illustrates the two standard dialog boxes.

Standard image compression dialog components provide a consistent user interface for specifying the parameters that control the compression of an image or image sequence. Your application specifies a test image for the dialog box and then calls the standard-image compression component. The component then presents a dialog box to the user, manages the dialog box, validates the user's settings, and stores those settings for your application. The standard dialog component also provides numerous facilities for determining reasonable default settings for a given image or sequence. Finally, this component manages the process of compressing the image or image sequence, using the parameter settings provided by the user or your application.

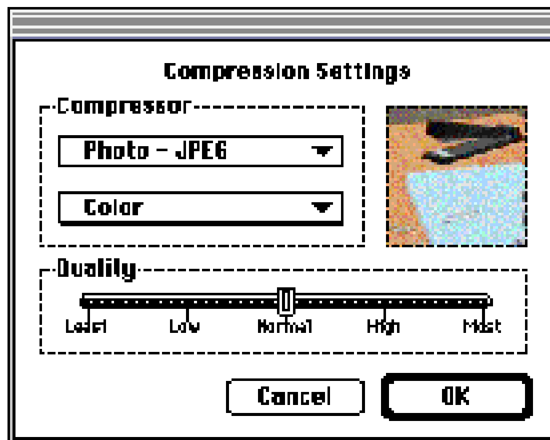
By using a standard image compression dialog component, you can reduce the amount of work you need to do in your application in order to compress an image or an image sequence. For example, you can eliminate the need to manage interactions with the user and to validate the image compression parameters specified by the user. Furthermore, the standard dialog component simplifies the process of compressing images or sequences. This, in turn, allows you to focus on the problem at hand, rather than on the details of image compression parameters. In addition, the standard image compression dialog component supplied by Apple supports many features that are helpful to the user, including Balloon Help and a test image. Finally, Apple's component will be localized by Apple, so that you need not worry about international issues relating to this dialog box.

Types of Dialog Boxes

Standard image compression dialog components support two basic dialog boxes. One dialog box provides a minimal interface and is suitable for compressing single images. Figure 13-1 shows an example of this dialog box. Using this dialog box, the user can select a compressor component, the pixel depth for the operation, and the desired spatial quality.

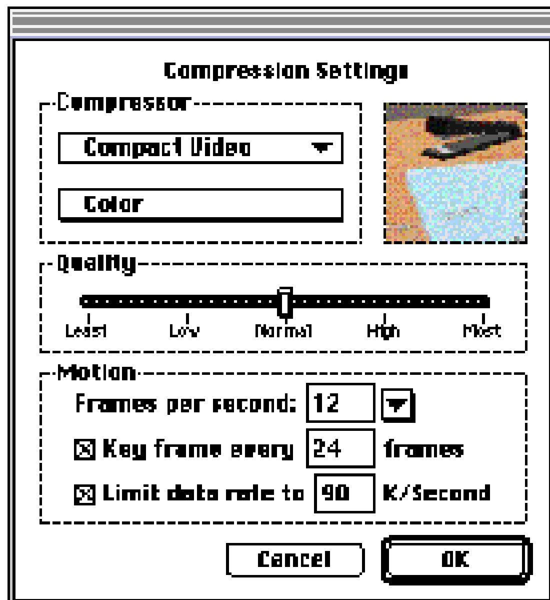
2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 13-1 Dialog box for single-frame compression



The other dialog box allows the user to set compression parameters for image sequences. In addition to the parameters supported by the single-frame dialog box, this dialog box supports frame rate, key frame rate, spatial and temporal quality settings, and data rate settings. Figure 13-2 shows an example of this dialog box.

Figure 13-2 Dialog box for image-sequence compression



Your application can control which dialog box is presented to the user.

By using standard dialog components, you can avoid many of the details of obtaining, validating, and using image compression parameters. The process of validating image compression parameters can be very involved, depending upon the capabilities of the selected compressor component. Apple's standard image compression dialog component verifies that the user's settings are valid for the selected compressor. In addition, this component uses a test image to demonstrate the effects of the user's compression settings.

 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Working With Standard Image Compression Dialog Components

This chapter describes in detail how you can use the standard image compression dialog component.

You can use the standard image compression dialog component to obtain image or image sequence compression parameters from the user and to manage the process of compressing the image or sequence. This component presents a consistent interface to the user and eliminates the need for you to worry about the details of managing this dialog box. Once you have collected the parameter information from the user, you can use the component to instruct the Image Compression Manager to perform the image or sequence compression. Again, the component manages the details for you.

Because the standard image compression dialog component is a component, you use the Component Manager to open and close your connection.

Before you can open a connection to a standard image compression dialog component, be sure that the Component Manager, Image Compression Manager, and 32-bit Color QuickDraw are available. You can use the Gestalt Manager to determine if these facilities are available.

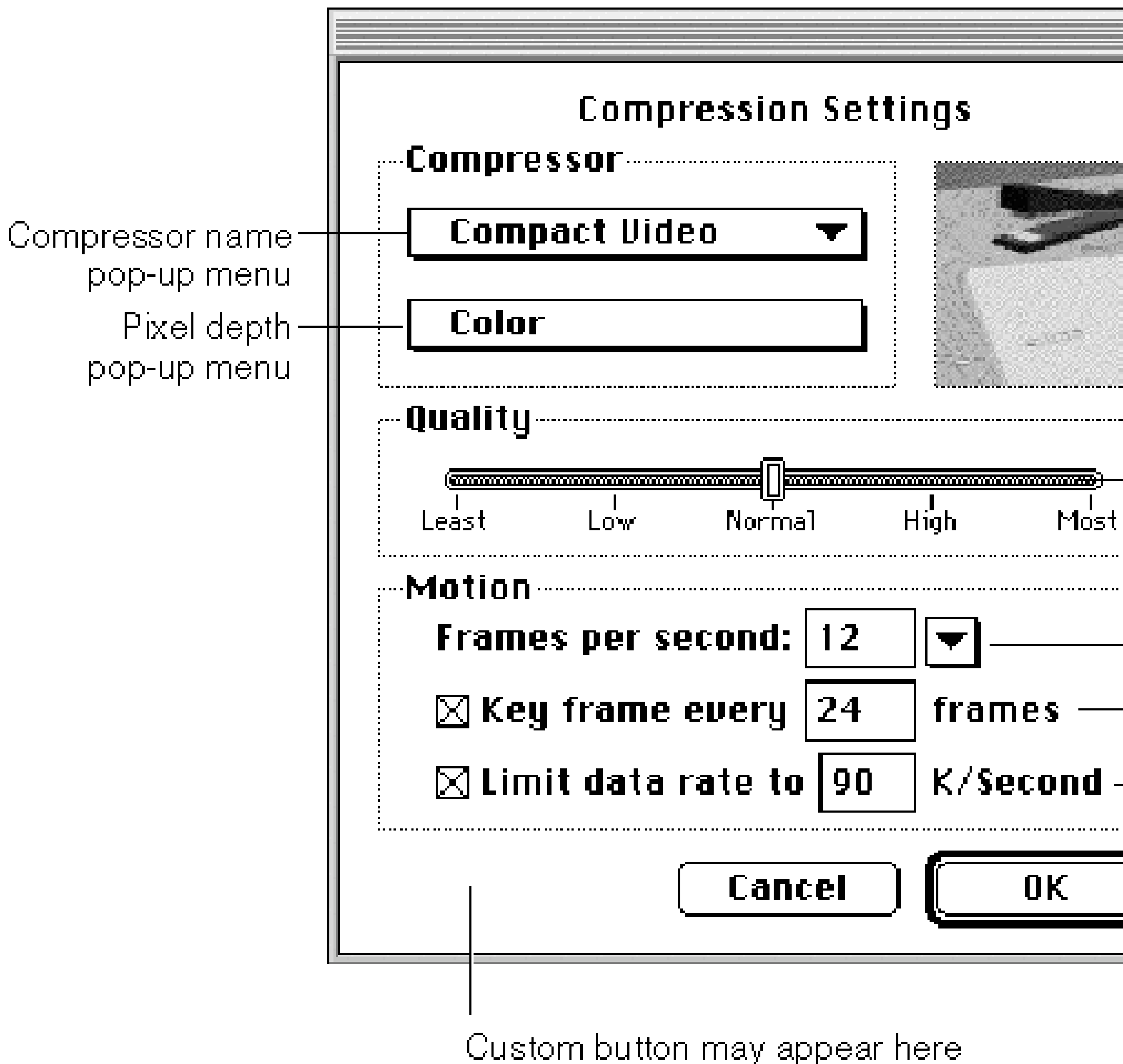
Once you have established a connection to a standard image compression dialog component, your application can present the dialog box to the user. The user selects the desired compression parameters and clicks the OK button. The component then stores these parameters for your application, using them, when appropriate, to work with the Image Compression Manager to compress the image or sequence.

Every standard image compression dialog box has its own set of parameter information. This information identifies the compressor component to be used, determines which dialog box is used, and specifies the parameters to be used during the compression operation. This information is stored by the component. You can use functions provided by the component to examine or modify these parameters.

The standard image compression dialog component provided by Apple allows you to augment or extend the interface provided by its dialog boxes. This component supports a single custom button. Your application enables this button when it instructs the component to display the dialog box to the user. You provide the code that supports this button in a hook function in your application. In addition, this component allows you to define a filter function; you can use this function to process dialog box events before the component. Figure 14-1 identifies the parts of the dialog box supported by Apple's standard dialog component.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Figure 14-1 Elements of the standard image compression dialog box



Opening a Connection to a Standard Image Compression Dialog Component

As is the case with all components, your application must establish a connection to a standard image compression dialog component before you can use its services. As with other components, you use the Component Manager's `OpenDefaultComponent` functions to connect to a component. You must use the Component Manager's `CloseComponent` function to close your application's connection when you are done.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Apple provides constants that define the component type and subtype values for standard image compression dialog components. All of these components have a type value of 'scdi'; you can use the StandardCompressionType constant to specify this value. These components have a subtype value of 'imag'; the StandardCompressionSubType constant defines this value.

Displaying the Dialog Box to the User

Once you have opened a connection to a standard image compression dialog component, you can proceed to display the dialog box to the user. In preparation, you might establish default parameter settings and specify a test image. Your application may then instruct the component to display the dialog box to the user. The following sections discuss each of these steps in more detail.

Setting Default Parameters

The standard dialog component stores and manages a set of compression parameters for your application. Before presenting the dialog box to the user, you may want to set default values for these parameters. The standard dialog component provides a number of options for establishing these default values:

- You may supply an image to the component from which it can derive default settings. The component examines the characteristics of the image and sets appropriate default values. The SCDefaultPictHandleSettings function works with images stored in picture handles; the SCDefaultPictFileSettings function works with images stored in picture files; and the SCDefaultPixMapSettings function works with pixel maps. These functions are discussed in Getting Default Settings for an Image or a Sequence (page 125).
- If you have not set any defaults, but you do supply a test image for the dialog, the component examines the test image and derives appropriate default values based upon its characteristics. The next section discusses how to assign a test image to the user dialog box.
- If you have not set any defaults and do not supply a test image, the component uses its own default values.
- You may modify the settings by using the SCSetInfo function. This function gives you a great deal of freedom; you can use it to modify any of the parameters stored by the component.

If you supply either a test or a default image, the standard dialog component extracts default compression settings from that image, including color table, grayscale information (if appropriate), and compression defaults (if the source image is already compressed). If any of these default values differ from your needs, use the SCSetInfo function to modify the value.

Designating a Test Image

The standard image compression dialog component provided by Apple supports a test image in its dialog box. The component uses this test image to show the user the effect of the current set of compression parameters. Whenever the user changes the dialog box settings, the component applies those parameters

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

to the test image and displays the results in its dialog box. In addition, the standard dialog component may sometimes use the test image to obtain hints about the type of compression operation you expect to perform. In some cases, the component may derive default parameter values by examining the test image.

The component provides three functions that allow you to specify a dialog box's test image. Each of these functions uses a different image source: a handle, a picture file, or a pixel map. Your application is responsible for obtaining the image and for disposing of it after you are done.

The test image portion of the dialog box supported by Apple's standard image compression dialog component is a square measuring 80 pixels by 80 pixels. In order to deal with test images that are larger than this area, Apple's component allows you to specify a part of the image to display. You can specify an **area of interest,** which indicates a portion of the test image that is to be displayed in the dialog box. If the area of interest is still larger than the display area in the dialog box, the component may shrink the image or crop it (or both) until the image fits.

Listing 14-1 shows one way to specify a test image. This code fragment uses an image that is stored in a picture file. The program asks the user to specify the file, using the `SFGetFilePreview` function. The program then opens the image file and instructs the standard image compression dialog component to use the picture that is stored in the file.

Listing 14-1 Specifying a test image

```
Point                   where;
ComponentInstance       ci;
SFTypeList              typeList;
SFReply                 inReply;
short                   srcPictFRef;
where.h = where.v = -2;                     /* center dialog box on the
                                               best screen */
typeList[0] = 'PICT';                       /* set file type */
SFGetFilePreview (where, "\p", nil, 1, typeList, nil,
                    &inReply);
if (!inReply.good) {     /* handle error */
}
result = FSOpen (inReply.fName, inReply.vRefNum, &srcPictFRef);
if (result) {               /* handle error */
}

result = SCSetTestImagePictFile
            (ci,                            /* component connection */
            srcPictFRef,                    /* source picture file */
            nil,                            /* use the entire image */
            scPreferScalingAndCropping);
                                            /* shrink image and crop it */
if (result) {                               /* handle error */
}
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Displaying the Dialog Box and Retrieving Parameters

Standard image compression dialog components provide two functions that display the dialog box to the user and retrieve the user's compression settings: `SCRequestImageSettings` and `SCRequestSequenceSettings`. Both of these functions start with your default parameter settings. Any changes made by the user are stored by the component. You may use the `SCGetInfo` function to examine these settings.

The `SCRequestImageSettings` function obtains image compression parameters from the user and displays a dialog box. The `SCRequestSequenceSettings` function works with sequence-compression parameters, using a dialog box. Both of these functions allow you to augment or extend the interface in the dialog box; see Extending the Basic Dialog Box (page 126) for more information about extending the basic dialog boxes.

Listing 14-2 shows how to use the `SCRequestImageSettings` function to display the dialog box to the user and obtain the resulting image compression settings. This code fragment obtains the compression parameters from the user and then uses those parameters to compress the image that is stored in the file the user selected in Listing 14-1. The program then stores the compressed image in a different file. This fragment assumes that the destination file has already been selected.

Listing 14-2 Displaying the dialog box to the user and compressing an image

```
ComponentInstance           ci;                 /* component connection */
short                       srcPictFRef;        /* source file */
short                       dstPictFRef;        /* destination file */
result = SCRequestImageSettings(ci);
if (result < 0) {                               /* handle error */
}
if (result == scUserCancelled) {                /* user clicked Cancel
                                                    button */
}
result = SCCompressPictureFile
            (ci,                                /* component connection */
            srcPictFRef,                        /* source picture file */
            dstPictFRef);                       /* dest picture file */
if (result < 0) {                               /* handle error */
}
```

Note that, because the standard dialog component stores the compression parameters for you, the new user settings become the default values the next time your application interacts with the user. If this is inappropriate, use one of the mechanisms discussed in Setting Default Parameters (page 123) to modify those defaults.

Getting Default Settings for an Image or a Sequence

This section describes the functions that allow you to derive sensible default compression settings for an image or a sequence. The standard dialog component examines an image you provide and selects appropriate default settings based on the image's characteristics. The component stores those settings for you and uses them with other functions, including not only functions governing image or sequence compression, but also utility functions such as `SCNewGWorld`. If you choose to display a dialog box to the user, the component uses these settings as the default dialog box settings.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Any of these functions may be used with a single image or an image that is part of a sequence. You tell the standard dialog component whether the image is part of a sequence when you call the function.

If there is a custom color table associated with the image or the sequence, these functions retrieve and store it. You can use the color table settings request to retrieve the custom color table and obtain as much color and depth information as possible from the image or sequence of images.

You can retrieve these settings using the `SCGetInfo` function, or modify them using the `SCSetInfo` function.

There are three functions available: `SCDefaultPictHandleSettings` works with pictures, `SCDefaultPictFileSettings` works with picture files, and `SCDefaultPixMapSettings` works with pixel maps.

Working With Image or Sequence Settings

The standard dialog component provides two functions that allow you to work with the current compression settings for an image or a sequence of images. You can establish these settings in a number of ways: see Setting Default Parameters (page 123) for more information about your options.

You use the `SCGetInfo` function to retrieve settings information. The `SCSetInfo` function enables you to modify the settings.

These functions can work with a number of different types of settings information. When you call either function, you specify the type of data you want to work with. Each of these request types requires different parameter data. See Request Types (page $@) for a description of each of these request types and their data requirements.

Extending the Basic Dialog Box

Apple's standard image compression dialog component allows you to customize the operation of the user dialog box in a number of ways. First, you can define a filter function. This function, which is a modal-dialog filter function, can process dialog box events before the component does. Your filter function can then perform custom processing that is appropriate to your application. Because the compression dialog box is a movable modal dialog box, you must provide a filter to process update events for your application windows.

Second, you can define a hook function. This function receives item hits before the standard image compression dialog component does, and can therefore augment the basic dialog box. For example, your hook function can provide additional validation of the user's selections.

Finally, you can define a custom button in the dialog box. You can then use your hook function to detect when the user clicks this button. Your hook function can then extend the dialog box interface by displaying additional dialog boxes, for example.

You use the `scExtendedProcsType` request type with the `SCSetInfo` function to take advantage of these mechanisms for customizing the user dialog box. Listing 14-3 contains code that uses this function to define a custom button in the dialog box.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Listing 14-3 Defining a custom button in the dialog box

```
SCExtendedProcs ep;
ep.filterProc = MyFilter;              /* custom filter function */
ep.hookProc = MyHook;                  /* custom hook function */
ep.refcon = 0;                         /* reference constant for filter
                                          and hook functions */
BlockMove("\pDefaults",ep.customName,32);  /* custom button name */
SCSetInfo(ci,scExtendedProcsType,&ep);  /* set new extended functions */
```

Listing 14-4 shows a hook function that returns the dialog box to its default settings whenever the user clicks the custom button. The standard dialog component calls this function each time the user selects an item in the dialog box. On entry, the hook function receives information about the current dialog box, a pointer to the appropriate standard image compression dialog parameter block, and a reference constant that is supplied by your application.

This hook function first checks to see whether the user clicked the custom button. If so, the function changes the current compression settings.

Listing 14-4 A sample hook function

```
pascal short MyHook(DialogPtr theDialog,short itemHit,
                            void *params,long refcon)
{
    SCSpatialSettings ss;

    if (itemHit == scCustomItem)     {          /* check for custom item */
        ss.codecType = 'jpeg';                  /* create new settings */
        ss.codec = anyCodec;
        ss.depth = 32;
        ss.spatialQuality = codecNormalQuality;
        SCSetInfo(params,                       /* component connection */
            scSpatialSettingsType,              /* set spatial settings */
            &ss);                               /* new spatial settings */
    }
    return (itemHit);
}
```

In your hook function, you may want to display additional user dialog boxes. Apple's standard image compression dialog component provides two functions that help you position your dialog box on the screen. The `SCPositionDialog` function places a dialog box in a specified location; the `SCPositionRect` function positions a rectangle. By using these functions you can position your dialog boxes near the standard dialog box.

Listing 14-5 contains code that uses the `SCPositionDialog` function to place a Standard File Package dialog box onto the same screen as the standard image compression dialog box.

Listing 14-5 Positioning related dialog boxes

```
Point       where;                          /* positions dialog boxes */
ComponentInstance       ci;                 /* component connection */
where.h = where.v = -2;                     /* center dialog box on the
                                                best screen */
result = SCPositionDialog (ci,              /* component connection */
            -3999,                  /* resource number of dialog box */
            &where);                /* returns upper-left point */
SFPutFile (where,                           /* positions the dialog box */
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
"\pSave compressed picture as":,
"\pUntitled",
nil,
&outReply);
```

Creating a Standard Image Compression Dialog Component

Apple's standard image compression dialog component fully implements the functional interface for components of this type. As a result, this component allows you to customize the dialog box by enabling the custom button or by defining a filter function. In most cases your application should be able to use the component that is supplied by Apple. However, if you want to create your own standard image compression dialog component, you should read this section.

Apple has defined a component type value for standard image compression dialog components. All components of this type have the same type and subtype values. You can use the following constants to specify the type and subtype.

```
#define  StandardCompressionType      'scdi'
#define  StandardCompressionSubType   'imag'
```

Apple has defined a functional interface for standard image compression dialog components. For information about the functions your component must support, see Types and Functions (page $@). You can use the following constants to refer to the request codes for each of the functions your component must support.

```
#define  scPositionRect               2   /* SCPositionRect */
#define  scPositionDialog             3   /* SCPositionDialog */
#define  scSetTestImagePictHandle     4   /* SCSetTestImagePictHandle */
#define  scSetTestImagePictFile       5   /* SCSetTestImagePictFile */
#define  scSetTestImagePixMap         6   /* SCSetTestImagePixMap */
#define  scGetBestDeviceRect          7   /* SCGetBestDeviceRect */
#define  scRequestImageSettings       10  /* SCRequestImageSettings */
#define  scCompressImage              11  /* SCCompressImage */
#define  scCompressPicture            12  /* SCCompressPicture */
#define  scCompressPictureFile        13  /* SCCompressPictureFile */
#define  scRequestSequenceSettings    14  /* SCRequestSequenceSettings */
#define  scCompressSequenceBegin      15  /* SCCompressSequenceBegin */
#define  scCompressSequenceFrame      16  /* SCCompressSequenceFrame */
#define  scCompressSequenceEnd        17  /* SCCompressSequenceEnd */
#define  scDefaultPictHandleSettings  18  /* SCDefaultPictHandleSettings */
#define  scDefaultPictFileSettings    19  /* SCDefaultPictFileSettings */
#define  scDefaultPixMapSettings      20  /* SCDefaultPixMapSettings */
#define  scGetInfo                    21  /* SCGetInfo */
#define  scSetInfo                    22  /* SCSetInfo */
#define  scNewGWorld                  23  /* SCNewGWorld */
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Image Compression Dialog Types and Functions

This chapter describes the request types and functions associated with the standard image compression dialog components and an application-defined function.

Request Types

This section describes the request types used by two standard dialog component functions that allow you to work with the current compression settings for an image or a sequence of images. (You can establish these settings in a number of ways; see Setting Default Parameters (page 123) for more information about your options.)

You use the `SCGetInfo` function to retrieve settings information. The `SCSetInfo` function enables you to modify the settings.

These functions can work with a number of different types of settings information. When you call either function, you specify the type of data you want to work with. The following request types are defined:

```
#define  scSpatialSettingsType    'sptl'      /* spatial options */
#define  scTemporalSettingsType   'tprl'      /* temporal options */
#define  scDataRateSettingsType   'drat'      /* data rate */
#define  scColorTableType         'clut'      /* color table */
#define  scProgressProcType       'prog'      /* progress function */
#define  scExtendedProcsType      'xprc'      /* extended dialog */
#define  scPreferenceFlagsType    'pref'      /* preferences */
#define  scSettingsStateType      'ssta'      /* all settings */
#define  scSequenceIDType         'sequ'      /* sequence ID */
#define  scWindowPositionType     'wndw'      /* window position */
#define  scCodecFlagsType         'cflg'      /* compression flags */
```

Each of these request types requires different parameter data. The following sections discuss each of these request types and their data requirements.

Spatial Settings Request Type

Use the spatial settings request to retrieve or modify the current spatial compression parameters. These parameters control how each image is compressed.

You supply a pointer to a spatial settings structure. If you are retrieving these settings, the standard dialog component places the current settings into the specified structure; if you are changing the settings, place the new values into the structure. The component uses those values to update its settings.

The `SCSpatialSettings` data type defines the format and content of the spatial settings structure:

```
typedef struct {
    CodecType           codecType;          /* compressor type */
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
    CodecComponent      codec;              /* compressor */
    short               depth;              /* pixel depth */
    CodecQ              spatialQuality;     /* desired quality */
} SCSpatialSettings;
```

Field	Description
`codecType`	Specifies the default compressor type that is displayed in the pop-up menu of compressors in the dialog box. The standard image compression dialog component uses this field to return the compressor type that was selected by the user. You must set this parameter to one of the compressor types supported by the Image Compression Manager, or to `nil`. If you set the field to `nil`, the standard image compression dialog component uses as the default value the first compressor or compressor type that it retrieves from the Image Compression Manager.
`codec`	Provides additional information about the default compressor that is displayed in the pop-up menu of compressors in the dialog box. If the user selects a specific compressor component, the standard image compression dialog component returns the appropriate compressor identifier in this field. Other options for this field are discussed below.
`depth`	Specifies the default value of the pixel depth pop-up menu in the dialog box. This menu allows the user to select the color or gray scale resolution value to be used when compressing the image or image sequence. If you set this field to 0, the component chooses an appropriate depth for the default compressor you specified with the `theCodec` field. When the user clicks OK, the standard image compression dialog component sets this field to the pixel depth value selected by the user. Note that the standard image compression dialog component may adjust the depth value so that it corresponds to a value that is supported by the compressor that has been selected by the user. The depth returned could be 0 if the `scShowBestDepth` flag is set.
`spatialQuality`	Specifies the default setting of the quality slider in the dialog box. This slider controls the spatial quality of the compressed image sequence, which influences the amount of spatial compression that can be achieved. Spatial compression eliminates redundant information within each frame in a sequence. When the user clicks OK, the standard image compression dialog component sets this field to the spatial quality value selected by the user. Note that the standard image compression dialog component may adjust the quality value so that it corresponds to a value that is supported by the compressor that has been selected by the user.

The `scListEveryCodec` bit in the flag in the `scPreferenceFlagsType` request influences the operation of the compressor list in the dialog box and, therefore, the way the component uses the `codec` field. Set the flag to 1 to have the list contain an entry for each compressor component in the system. If the flag is set to 1, the standard image compression dialog component uses this field along with the `codecType` field to select the default compressor that appears in the dialog box. To specify a default image compressor component, set this field to the appropriate compressor identifier. When the user clicks OK in the dialog box, the standard image compression dialog component returns the compressor identifier that corresponds to the selected image compressor component.

If you set the `codec` field to `nil`, the standard image compression dialog component uses as the default value the first compressor of the specified type that it retrieves from the Image Compression Manager. If you have set the flag to 0, the list contains only one entry for each type of compressor in the system. The standard

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

image compression dialog component ignores this field when creating the list of compressor types. In this case, the standard image compression dialog component does not change the value of this field when the user clicks OK.

However, you may use the `codec` field to specify additional selection criteria by setting this field to one of the special compressor identifiers supported by the Image Compression Manager. The standard image compression dialog component may use this value when it validates the compression parameters selected by the user.

Temporal Settings Request Type

Use the temporal settings request to retrieve or modify the current temporal compression parameters. These parameters govern sequence-compression operations.

You supply a pointer to a temporal settings structure. If you are retrieving these settings, the standard dialog component places the current settings into the specified structure; if you are changing the settings, place the new values into the structure. The component uses those values to update its settings.

The `SCTemporalSettings` data type defines the format and content of the temporal settings structure:

```
typedef struct {
    CodecQ      temporalQuality;    /* desired quality */
    Fixed       frameRate;          /* frame rate */
    long        keyFrameRate;       /* key frame rate */
} SCTemporalSettings;
```

Field	Description
`temporalQuality`	Specifies the default setting of the motion quality slider in the dialog box. This slider controls the temporal quality of the compressed image, which influences the amount of temporal compression that can be achieved (note that Apple's component uses the same slider for both spatial and temporal quality). Temporal compression eliminates redundant information between frames in an image sequence. When the user clicks OK, the standard image compression dialog component sets this field to the temporal quality value selected by the user. Note that the standard image compression dialog component may adjust the quality value so that it corresponds to a value that is supported by the compressor that has been selected by the user.
`frameRate`	Specifies the default value of the text-edit box that controls the number of frames per second in the image sequence to be compressed. This dialog item allows the user to select the frame rate to be used when compressing the image sequence. Note that this field is stored as a fixed-point number, allowing the user to specify fractional frame rates. When the user clicks OK, the standard image compression dialog component sets this field to the frame rate value specified by the user. If you have set the `scAllowZeroFrameRate` flag to 1 in the `scPreferenceFlagsType` request, and the user specifies nothing or 0, the component sets this field to 0. This dialog item can be useful in cases where your application cannot determine the frame rate of the source movie. For example, movies stored in PICT files do not include frame rate information. Therefore, the user must specify a frame rate for you. Alternatively, some users may want to create movies with different frame rates. This item allows the user to specify a rate for the compressed sequence.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
keyFrameRate	Specifies the default value of the text-edit box that controls the frequency with which key frames are inserted into the compressed image sequence. Key frames provide points from which a temporally compressed sequence may be decompressed. When the user clicks OK, the standard image compression dialog component sets this field to the key frame rate value specified by the user. If you have set the scAllowZeroKeyFrameRate flag to 1 in the scPreferenceFlagsType request, and the user specifies nothing or 0, the component sets this field to 0.

Data-Rate Settings Request Type

Use the data-rate settings request to retrieve or modify the current temporal compression parameters that govern the data rate. These parameters affect sequence-compression operations.

You supply a pointer to a data-rate settings structure. If you are retrieving these settings, the standard dialog component places the current settings into the specified structure; if you are changing the settings, place the new values into the structure; the component uses those values to update its settings.

The SCDataRateSettings data type defines the format and content of the data-rate settings structure:

```
typedef struct {
    long        dataRate;              /* desired data rate */
    long        frameDuration;         /* frame duration */
    CodecQ      minSpatialQuality;     /* minimum value */
    CodecQ      minTemporalQuality;    /* minimum value */
} SCDataRateSettings;
```

Field	Description
dataRate	Specifies the maximum number of bytes of compressed data your application wants to receive per second. Use this parameter to modulate the rate at which the component passes compressed data to your application. This can be useful to account for hardware limitations during sequence playback.
frameDuration	Indicates the duration of each frame, in milliseconds. Set this parameter to 0 to allow the standard dialog component to calculate the duration based upon the frame rate you specify in an scTemporalSettingsType request. However, if you allow the user to specify a 0 frame rate (that is, you set thescAllowZeroFrameRate flag to 1 in your scPreferenceFlagsType request), you must set the frame duration each time you compress a frame, because the component does not have sufficient information to determine an appropriate rate.
minSpatialQuality	Specifies the minimum acceptable spatial quality. In order to meet your specified data rate, the standard dialog component may have to adjust the spatial quality setting. Use this parameter to set a minimum level, which the component may not exceed.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
minTemporalQuality	Specifies the minimum acceptable temporal quality. As with spatial quality, in order to meet your specified data rate, the standard dialog component may have to adjust the temporal quality setting. Use this parameter to set a minimum level, which the component may not exceed.

Color Table Settings Request Type

Use the color table settings request to retrieve or modify the color table that the standard dialog component uses with all compression operations. Unless you specify otherwise, the component extracts the color table from the source image or sequence.

You supply a pointer to a color table handle (CTabHandle data type). Your application is responsible for disposing of this handle when you are done with it. Set the pointer to nil to clear the current color table; this may be useful if the current color table is inappropriate for the image or sequence you are working with.

Progress Function Request Type

Use the progress function request to assign a progress function for use by the standard dialog component. The progress function is a part of your application. The standard dialog component calls this function during time-consuming operations, and reports its progress. Your progress function can use the information it receives from the standard dialog component to keep the user informed about the progress of the operation.

You supply a pointer to an Image Compression Manager progress function structure. Set the pointer to nil to clear the current progress function; in this case, the standard dialog component does not report its progress to the user. Set the pointer to -1 to use the component's default progress function.

Extended Functions Request Type

Use the extended functions request to extend the interface provided in the standard image or sequence dialog boxes. You may specify a filter function, a hook function, and a custom button; you may retrieve the current settings for these options using the SCGetInfo function.

You supply a pointer to an extended functions structure. If you are retrieving these settings, the standard dialog component places the current settings into the specified structure; if you are changing the settings, place the new values into the structure; the component uses those values to update its settings. Set this pointer to nil to remove the current functions.

By default, none of these extended interface elements are used.

The SCExtendedProcs data type defines the format and content of the extended functions structure:

```
typedef struct {
    SCModalFilterProcPtr    filterProc;     /* filter function */
    SCModalHookProcPtr      hookProc;       /* hook function */
    long                    refcon;         /* reference constant */
    Str31                   customName;     /* custom button name */
} SCExtendedProcs;
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Field	Description
filterProc	Contains a pointer to a modal-dialog filter function in your application. Because the compression dialog box is a movable modal dialog box, you must provide a filter to process update events for your application windows. The standard component calls your filter function before it processes the event. You can use this function to control events in the dialog box. For example, you might use the filter function to release processing time to other windows displayed by your application while the standard image compression dialog box is being displayed. If you do not want to specify a filter function, set this parameter to nil.
hookProc	Contains a pointer to a dialog hook function in your application. The standard component calls your hook function whenever the user selects an item in the dialog box. You can use this function to customize the operation of the standard image compression dialog box. For example, you might want to support a custom button that activates a secondary dialog box. Another possibility would be to provide additional validation support when the user clicks OK. If you do not want to specify a hook function, set this parameter to nil.
refcon	Specifies a reference constant that is to be passed to the dialog hook function and the modal-dialog filter function.
customName	Specifies the string to be displayed in the custom button in the dialog box. If you are not using a custom button, set this parameter to nil .

This is how to declare a filter function named MyFilter:

```
pascal Boolean MyFilter (DialogPtr theDialog,
    EventRecord *theEvent, short *itemHit, long refcon);
```

This is how to declare a hook function named MyHook:

```
pascal short MyHook (DialogPtr theDialog,
    short *itemHit, SCParams *params, long refcon);
```

In both cases the refcon parameter accepts the reference constant that you supply in the refcon field of the extended functions structure.

Preference Flags Request Type

Use the preference flags request to specify or retrieve the standard dialog component's preference flags. These flags govern some of the details of the dialog box that are presented to the user.

You supply a pointer to a long integer. If you are retrieving these flags, the standard dialog component places the current settings into the specified field; if you are changing the flags, set the field with your desired flag values; the component uses those values to update its settings.

By default, the SCRequestImageSettings function operates with the scShowBestDepth and scUseMovableModal flags set to 1. The SCRequestSequenceSettings function operates with the scUseMovableModal flag set to 1. You should never need to change the values of the scListEveryCodec or scUseMovableModal flags.

The following flags are defined:

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
#define  scListEveryCodec          (1L<<1)  /* list every component */
#define  scAllowZeroFrameRate      (1L<<2)  /* allow 0 frame rate */
#define  scAllowZeroKeyFrameRate   (1L<<3)  /* 0 key frame rate OK */
#define  scShowBestDepth           (1L<<4)  /* use best image depth */
#define  scUseMovableModal         (1L<<5)  /* use movable dialog */
```

Flag	Description
scListEveryCodec	Controls the contents of the pop-up menu of compressors. If you set this flag to 1, the standard image compression dialog component lists every compressor component that is present in the system. Each entry in the list contains the name of a compressor component. The user may then select a specific component from the list. If you set this flag to 0, the list contains one entry for each type of compressor component that is present in the system. Each list entry contains the name of a compressor type (for example, a list entry might contain "Animation" for the animation compressor). The user may then choose a type of compressor and it is your application's responsibility to select an appropriate compressor of that type.
scAllowZeroFrameRate	Determines whether the component allows the user to specify a value of 0 for the frame rate. If you set this flag to 1, the component allows the user to specify either 0 or nothing for the frame rate. The component then includes a "best rate" entry in the pop-up menu. If the user specifies 0, the component sets the frameRate field in the SCTemporalSettings structure to 0. Your application must then determine the best frame rate for the movie. If you set this flag to 0, the component does not allow the user to enter 0 for the frame rate. In this case, the user must select a specific frame rate.
scAllowZeroKey-FrameRate	Similar to the scAllowZeroFrameRate flag, this flag determines whether the component allows the user to specify a value of 0 for the key frame rate. If you set this flag to 1, the component allows the user to specify 0 for the frame rate. If the user specifies 0, the component sets the keyFrameRate field in the SCTemporalSettings structure to 0. Your application must then determine the best key frame rate for the movie. If you set this flag to 0, the component does not allow the user to specify 0 for the frame rate. In this case, if the user has enabled temporal compression by checking the key frame checkbox, the user must also select a specific key frame rate.
scShowBestDepth	Determines whether the component includes a "best depth" entry in the pop-up menu for pixel depth. If you set this flag to 1, the component includes a "best depth" entry in the pop-up menu. If the user selects "best depth", the component sets the depth to 0. Your application must then determine the best pixel depth for the movie. If you set this flag to 0, the component does not include a "best depth" entry in the pop-up menu. The user must select a depth from among the other available choices.
scUseMovableModal	Determines whether the standard compression dialog is a movable or a stationary dialog. Set this flag to 1 to create a movable dialog. In this case, you should provide an event filter function to handle update events (use the scExtendedProcsType request).

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Settings State Request Type

Use the settings state request to set or retrieve the configuration of the standard dialog component. You may use this request to retrieve the configuration information so that you can save it for later use, or to reconfigure the component based on a saved configuration.

Your application is not concerned with the content of the configuration information that is returned. The standard dialog component saves its configuration in a format that it understands. This request affects only those settings that are valid across system restarts, such as the spatial and temporal compression parameters and the data-rate settings.

You supply a pointer to a handle. When you retrieve the settings, the standard dialog component creates an appropriately-sized handle and places its current configuration information into the handle. Your application is responsible for disposing of the handle when you are done with it.

When you modify the settings, you supply the configuration information in the handle. The component copies the data out of this handle. Your application is responsible for disposing of the handle when you are done with it. Set the pointer to `nil` to reset the component to its default configuration.

Sequence ID Request Type

Use the sequence ID request type to retrieve the sequence identifier being used by the component's `SCCompressSequenceFrame` function. You may not use this request to set the sequence identifier.

You supply a pointer to a field of type `ImageSequence` (this is an Image Compression Manager data type). The standard dialog component returns the current sequence identifier in that field.

Window Position Request Type

Use the window position request to position the user's dialog box.

You supply a pointer to a point. If you are retrieving this information, the standard dialog component places the coordinates of the upper-left corner of the dialog box into this point; if you are changing the dialog box's position, place the new coordinates into the point structure; the component uses those coordinates to position the dialog box.

Normally you should not need to use this request. By default, the standard dialog component centers the dialog box on the screen that is best-suited to display your test image. The component also saves the last window position for movable modal dialogs.

Control Flags Request Type

Use the control flags request to retrieve or modify the control flags used by the standard dialog component. The standard dialog component passes these flags through to the image compressor it uses to compress your image or sequence.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

You supply a pointer to a flags field of data type `CodecFlags` (this is an Image Compression Manager data type). If you are retrieving the flags, the standard dialog component places the current flags into this field. If you are setting new flag values, place your desired settings into the field; the component uses these new flag settings.

By default, the standard dialog component sets all flags to 0 when it compresses still images. When it is compressing sequences, the component sets the `codecFlagsPreviousUpdate` and `codecFlagsUpdatePreviousComp` flags to 1. Typically, you should not need to change these flag settings.

Standard image compression Dialog Component Functions

This section describes the functions that are supported by standard image compression dialog components.

Displaying the Standard image compression Dialog Box

Standard image compression dialog components provide two functions that allow you to display the standard dialog box to the user and retrieve the compression parameters specified by the user.

Use the `SCRequestImageSettings` function to retrieve the user's preferences for compressing a single image; use the `SCRequestSequenceSettings` functions when you are working with an image sequence.

Both of these functions manipulate the compression settings that the component stores for you. The component may derive the current settings from a number of different sources:

- You may supply an image to the component from which it can derive default settings. You do this by using one of the functions discussed in Getting Default Settings for an Image or a Sequence (page 125).
- If you have not set any defaults, but you do supply a test image for the dialog, the component examines the test image and derives appropriate default values based upon its characteristics.
- If you have not set any defaults and do not supply a test image, the component uses its own default values.
- You may modify the settings by using the `SCSetInfo` function.
- You may allow the user to modify those settings by calling one of the functions discussed in this section.

You may customize the dialog boxes by specifying a modal-dialog hook function or a custom button. You may use the custom button to invoke an ancillary dialog box that is specific to your application. See Request Types (page $@) for more information.

Compressing Still Images

The standard dialog component provides three functions you may use to compress a still image. These functions differ based on how the image is stored: `SCCompressImage` works with pixel maps; `SCCompressPicture` compresses a picture that is stored in a handle; and `SCCompressPictureFile` works with pictures stored in files.

All of these functions use the current compression settings. See Displaying the Standard image compression Dialog Box (page $@) for detailed information about establishing these current settings.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

If there are no default settings, each of these functions could potentially display a dialog box for single-frame compression operations.

Compressing Image Sequences

The standard dialog component provides three functions you may use to compress an image sequence. The `SCCompressSequenceBegin` function allows you to start a sequence-compression operation; use the `SCCompressSequenceFrame` function for each image in the sequence; you end the sequence by calling the `SCCompressSequenceEnd` function. The standard dialog component manages all of the compression details for you. Your application may have only one sequence-compression operation active on any given connection; naturally, you may have more than one connection active at a time.

All of these functions use the current compression settings. See Displaying the Standard image compression Dialog Box (page $@) for detailed information about establishing these current settings.

If there are no default settings, each of these functions could potentially display a dialog box for sequence-compression operations.

Specifying a Test Image

The standard image compression dialog component provided by Apple supports a test image. The dialog box contains a small image along with the other parts of the dialog box. The component uses this image to display the effect of the user's image compression settings. In this manner, the user can experiment with different settings and see the results of those settings immediately.

The component provides three functions that allow you to specify the test image. Use the `SCSetTestImagePictHandle` function if your test image is stored in a handle. Use the `SCSetTestImagePictFile` function if your test image is in a picture file. The `SCSetTestImagePixMap` function sets the test image from a pixel map.

Positioning Dialog Boxes and Rectangles

Standard image compression dialog components provide functions that allow you to position rectangles and dialog boxes. These functions are most useful in helping you to manage dialog boxes that are related to the standard image compression dialog. For example, your application might support a custom button that initiates a dialog box with the user to specify additional compression parameters. You can use these functions to position that dialog box in relation to the standard image compression dialog box.

There are two positioning functions: the `SCPositionRect` function positions a rectangle; the `SCPositionDialog` positions a dialog box. The `SCGetBestDeviceRect` function returns information about the best available display device.

 © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Using Image Transcoder Components

QuickTime's image transcoding support is part of the Image Compression Manager API. The Image Compression Manager uses an image sequence when compressing or decompressing data. An image sequence allows QuickTime to make certain optimizations because it knows that a similar operation will be repeated multiple times (that is, images will be repeatedly compressed to the same image data format).

A transcoder translates image data from one compressed format to another. This is done automatically, if an appropriate transcoder component is available, when QuickTime plays a movie that has been compressed in a format for which there is no decompressor on the playback machine. Transcoder components can also be used when an application provides an export function that saves data in a compressed format different from the compressed format of the source data.

Transcoding has two distinct advantages over the decompress-then-recompress approach to converting the format of compressed data. The first advantage is that the operation is usually substantially faster, since much of the data can be copied directly from the source image data format to the destination image data format. The second advantage is that the operation is usually more accurate because decompressing and recompressing provides two steps for introducing rounding and quantization errors. By directly transcoding, opportunities for small errors are substantially reduced.

QuickTime's image transcoding support is part of the Image Compression Manager API. Image transcoding can be invoked either explicitly, using transcoder-specific Image Compression Manager functions such as the `ImageTranscoderBeginSequence` function, or implicitly, using standard routines for decompressing images that in turn may use a transcoder when needed.

As with most other services in QuickTime, the details of image transcoding are handled by components. The Image Compression Manager uses image transcoder components to perform both implicit and explicit image transcoding. Application developers that perform image transcoding interact with the Image Compression Manager, not directly with the image transcoder components themselves. The Image Compression Manager takes care of the details of working with image transcoder components.

If you want to add new image transcoding capabilities to QuickTime, you can write an image transcoder component, as described in Creating Image Transcoder Components (page 143).

Invoking an Image Transcoding Process

Image transcoding can be invoked either explicitly, using transcoder-specific Image Compression Manager functions such as the `ImageTranscoderBeginSequence` function, or implicitly, using standard routines for decompressing images that in turn may use a transcoder when needed.

If a request is issued to decompress an image, but no image decompressor component is installed for that image format, QuickTime attempts to locate an image transcoder to convert the image data into a supported format. This automatic image transcoding is supported for both QuickTime movies and compressed image data. Implicit use of transcoders is transparent to the applications programmer and requires no special coding. For more information about decompressing images, see The Image Compression Manager (page 15).

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

QuickTime also provides a set of functions that applications can use explicitly to transcode images. These functions make it possible for any application to take compressed image data and transcode it into another format. For example, some applications create QuickTime movies by combining segments of other QuickTime movies that may be compressed in various formats. These applications often export the compressed image data by decompressing the images and then recompressing them to a common format. You can often use image transcoders to increase the speed and fidelity of these operations.

Transcoding Paths

The Image Compression Manager's support for image transcoding is based on an image transcoding sequence. The Image Compression Manager supports two paths for transcoding:

1. `ImageTranscodeSequenceBegin`, `ImageTranscodeFrame`, `ImageTranscodeDisposeFrameData`, `ImageTranscodeSequenceEnd`
2. `ImageTranscoderBeginSequence`, `ImageTranscoderConvert`, `ImageTranscoderDisposeData`, `ImageTranscoderEndSequence`

Use path #1 (the ...`codeSequenceBegin` series) if you want QuickTime to find a transcoder component for you, opening it for you and closing it when the sequence is complete. This is the simplest path.

Use path #2 (the ...`coderBeginSequence` series) if you want to use a particular transcoder component, or if you already have a transcoder component open and want to reuse it. One reason for doing this is if you are transcoding data with more than one image description (for example, the height or width of the source data changes at some point). This path is more efficient for multiple transcoding sequences, but requires you to find, open, and close the transcoder component yourself.

In either case, you begin by initiating a sequence and specifying the source and destination buffers and the source image description. You then make a series of calls to convert images, disposing of the transcoded data when you are done with it. Finally, you end the sequence, allowing QuickTime to release the resources associated with it.

If you are using path #2, you must find and open a transcoder component before beginning the first transcoding sequence, and close the component after you have completed the last transcoding sequence. Image transcoder components have component type `'imtc'`. The subtype and manufacturer fields specify the input and output data compression formats. For example, a transcoder that converts Motion JPEG to PICT would have type `'imtc'`, subtype `'mjpg'`, manufacturer code `'pict'`.

Here is a list of the image transcoder functions:

- Transcode sequence functions (QuickTime opens and closes component)
  - `ImageTranscoderBeginSequence` initiates an image transcoding sequence and specifies the input data format.
  - `ImageTranscoderConvert` performs image transcoding operations.
  - `ImageTranscoderDisposeData` disposes of transcoded data.
  - `ImageTranscoderEndSequence` ends an image transcoding sequence.
- Transcoder component sequence functions (your application opens and closes component)

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

- ❏ `ImageTranscodeSequenceBegin` initiates an image transcoder sequence operation.
- ❏ `ImageTranscodeFrame` transcodes a frame of image data.
- ❏ `ImageTranscodeDisposeFrameData` disposes transcoded image data.
- ❏ `ImageTranscodeSequenceEnd` ends an image transcoder sequence operation.

For further information about these functions, see the *QuickTime API Reference*.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Creating Image Transcoder Components

Image transcoder components are standard Component Manager components. An example component is provided in this chapter.

Image transcoder components have a type of 'imtc'.

The subtype field of the component defines the compressed image data format that the transcoder accepts as an input. The manufacturer field of the component defines the compressed image data format that the transcoder generates as output.

For example, a transcoder from Motion JPEG Format A to Motion JPEG Format B would have a subtype of 'mjpg' and a manufacturer code of 'mjpb'. No component-specific flags are currently defined for transcoders; they should be set to 0.

An Example

The example code in Listing 17-1 shows an image transcoder component. It converts an imaginary compressed data format 'bgr ' to uncompressed RGB pixels. The transcoding process simply copies the source data to the destination and inverts each byte in the process. This example shows the format of how an image transcoder might work without getting into the details of a particular image transcoding operation.

Listing 17-1 An image transcoder component that converts a compressed data format to uncompressed RGB pixels

```
#include <ImageCompression.h>
pascal ComponentResult main(ComponentParameters *params, Handle storage );
pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle
srcDesc, ImageDescriptionHandle *dstDesc, void *data, long dataSize);
pascal ComponentResult TestTranscoderConvert (Handle storage, void *srcData,
long srcDataSize,
void **dstData, long *dstDataSize);
pascal ComponentResult TestTranscoderDisposeData (Handle storage, void *dstData);
pascal ComponentResult TestTranscoderEndSequence (Handle storage);
pascal ComponentResult main(ComponentParameters *params, Handle storage )
{
    ComponentFunctionUPP proc = nil;
    ComponentResult err = noErr;
    switch (params->what) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
            break;
        case kImageTranscoderBeginSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderBeginSequence;
            break;
        case kImageTranscoderConvertSelect:
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

```
            proc = (ComponentFunctionUPP)TestTranscoderConvert;
            break;
        case kImageTranscoderDisposeDataSelect:
            proc = (ComponentFunctionUPP) TestTranscoderDisposeData;
            break;
        case kImageTranscoderEndSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderEndSequence;
            break;
        default:
            err = badComponentSelect;
            break;
    }
    if (proc)
        err = CallComponentFunctionWithStorage(storage,
                        params, proc);
    return err;
}
pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle
srcDesc, ImageDescriptionHandle *dstDesc, void *data, long dataSize)
{
    *dstDesc = srcDesc;
    HandToHand((Handle *)dstDesc);
    (***dstDesc).cType = 'raw ';
    return noErr;
}
pascal ComponentResult TestTranscoderConvert (Handle storage, void *srcData,
long srcDataSize,
void **dstData, long *dstDataSize)
{
    Ptr p;
    OSErr err;
    if (!srcDataSize)
        return paramErr;
    p = NewPtr(srcDataSize);
    err = MemError();
    if (err) return err;
    {
        Ptr p1 = srcData, p2 = p;
        long counter = srcDataSize;
        while (counter--)
            *p2++ = ~*p1++;
    }
    *dstData = p;
    *dstDataSize = srcDataSize;
    return noErr;
}
pascal ComponentResult TestTranscoderDisposeData (Handle storage, void *dstData)
{
    DisposePtr((Ptr)dstData);
    return noErr;
}
pascal ComponentResult TestTranscoderEndSequence (Handle storage)
{
    return noErr;
}
```


2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *QuickTime Compression and Decompression Guide*.

Date	Notes
2006-01-10	New document that describes the QuickTime data compression and decompression technologies.
	Replaces "Image Compression Manager," "Codec Components," "Data Codecs," "Image Compression Dialog," and "Image Transcoders."
2002-09-17	New document that explains how to compress and decompress image and video data.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.

2006-01-10 | © 2005, 2006 Apple Computer, Inc. All Rights Reserved.