XIL™ Programmer's Guide

SunSoft

A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

The XIL™ Imaging Library provides a set of key functions for the fields of image processing and digital video. This book explains how to use these functions in developing application programming interfaces (APIs) and applications.

Who Should Use This Book

The XIL library was designed to meet the needs of developers creating APIs and applications for a number of markets. These markets include:

- Markets that require digital video technology
- Commercial document imaging
- Technical document imaging
- Desktop publishing
- Color prepress
- Graphics arts
- Technical imaging

How This Book Is Organized

This book presents XIL in three parts. Chapters 1 through 8 explain how to create the framework for an XIL program and discuss such topics as input, output, and error handling. Chapters 9 through 11 discuss the XIL library's

image processing functions—functions that would be used inside the framework. Chapters 12 through 20 explain how the XIL library enables you to compress and decompress sequences of digital images.

A chapter-by-chapter description of the book follows:

Chapter 1, "Introduction to the XIL Library," provides an overview of the XIL library functions. In addition, the chapter discusses how the XIL library relates to other Sun Microsystems™, and third-party, libraries.

Chapter 2, "Basic XIL Program," introduces programming with the XIL library by looking at a simple XIL program that reads an 8-bit grayscale image from a file and displays it in an X window.

Chapter 3, "XIL Images," discusses the different types of XIL images, what attributes XIL images have, and how images are stored in memory.

Chapter 4, "XIL Storage," describes each of the XIL image storage formats and the functions for accessing information in a storage object. In addition, the chapter discusses how to enforce storage organization.

Chapter 5, "XIL Program That Uses Tiling," describes how to write an XIL program that uses tiled memory.

Chapter 6, "Handling Input and Output," takes a systematic look at how you handle the reading and writing of images in an XIL program, including input/output (I/O) with files, displays, and other devices.

Chapter 7, "Reading Kodak PhotoCD Images," discusses the library's device handler for reading and decoding the Eastman Kodak Company's Photo CD format.

Chapter 8, "Preparing Images for Display," discusses some of the issues that arise when you need to display different types of XIL images on different types of displays.

Chapter 9, "Presentation Functions," discusses a group of functions that are useful in preparing images for display. Topics covered include dithering and color space conversion.

Chapter 10, "Error Handling," discusses the XIL library's default error handler and explains how to write and install a custom error handler.

Chapter 11, "Arithmetic, Relational, and Logical Functions," covers the XIL library's arithmetic and logical functions. These functions enable you to add two images, take the logical AND of two images, multiply an image by a constant, and so on.

Chapter 12, "Geometric Functions," discusses the XIL library's geometric functions. Included among these are functions enabling you to scale images, rotate images, and transpose images.

Chapter 13, "Miscellaneous Image Processing Functions," presents the remaining image-processing functions in the XIL library. Among other uses, these functions give you the ability to filter images, pass images through lookup tables, and dilate or erode images.

Chapter 14, "Compressed Image Sequences," discusses the compressed image sequence data structure (called a CIS) in which you store compressed image data. The chapter goes over basic CIS management operations such as creating a CIS and writing data to it. In addition the chapter discusses CIS attributes and how to recover from datastream errors.

Chapter 15, "Compressing and Decompressing Sequences of Images," illustrates compressing and decompressing image sequences by presenting two example programs: a movie maker and a movie player.

Chapter 16, "JPEG Baseline Sequential Codec," discusses how the library's JPEG baseline sequential compressor and decompressor (codec) works and the special attributes of a JPEG CIS.

Chapter 17, "JPEG Lossless Codec," explains how the JPEG lossless codec performs its job and presents information about attributes that are specific to a CIS associated with a JPEG lossless codec.

Chapter 18, "H.261 Codec," covers the XIL interface to the codec specified by the CCITT in Recommendation H.261. The chapter discusses how such a codec works and lists the CIS attributes that apply specifically to a CIS associated with an H.261 compressor or decompressor.

Chapter 19, "MPEG-1 Codec," discusses the XIL interface to the MPEG-1 codec specified by the Moving Pictures Expert Group. The chapter discusses how an MPEG-1 codec works and lists the CIS attributes that apply specifically to a CIS associated with an MPEG-1 codec.

Chapter 20, "CCITT Group 3 and Group 4 Codecs," discusses how these document image codecs work and lists the CIS attributes specific to these codecs.

Chapter 21, "Cell Codec," deals with the Cell codec. The chapter briefly explains how the Cell codec works, discusses attributes that are specific to a Cell codec, and explains how to call optimized routines to play back Cell-encoded movies.

Chapter 22, "CellB Codec," discusses the library's CellB codec, which was derived from the Cell codec for use in videoconferencing applications. The chapter explains how the CellB codec works and the special attributes of a CellB CIS.

Chapter 23, "Acceleration in XIL Programs," explains how the XIL runtime system defers the execution of functions called in XIL programs as long as possible so that it can replace certain sequences of functions with optimized routines.

Appendix A, "Optimizations and Molecules For XIL Version 1.3," identifies the optimizations provided with the current release of the library. Molecules are optimized routines that the library can execute in lieu of executing a predefined sequence of functions from the API. The appendix includes additional optimizations for some of the molecules.

Appendix B, "XIL Error Messages," provides a list of XIL error messages. For each message, the appendix specifies an error ID and a list of functions that can generate the error.

Appendix C, "XIL-XGL Interoperability," explains how a single program can use both XIL and XGL™ functions to process an image.

Appendix D, "Cell and CellB Bytestream Definitions," presents the information about Cell and CellB bytestreams that you would need to implement a Cell or CellB codec.

Appendix E, "Bibliography," lists some books and articles to consult for further information on such subjects as image processing operations, JPEG compression, dithering, and color models.

The book also contains a glossary of terms from the fields of image processing and digital video.

Related Books

The primary companion to this book is the *XIL Reference Manual*. The reference manual contains man pages for all the functions in the XIL library.

The *XIL Programmer's Guide* and the *XIL Reference Manual* are included in the AnswerBook2™ on-line documentation. See the XIL 1.3 AnswerBook located at the following web site:

`http://docs.sun.com`

Because programming with the XIL library can be closely tied to programming with the X library, you may also find it useful to consult the following books:

- *Xlib Programming Manual*

- *Xlib Reference Manual*

- *Developing Visual Applications XIL: An Imaging Foundation Library* (ISBN 0-13-461948-X)

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Note – The term "x86" refers to the Intel 8086 family of microprocessor chips, including Pentium and Pentium Pro processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture, whereas "*Intel Platform Edition*" appears in the product name.

What Typographic Changes Mean

The following table describes the typefaces used in this book.

Table 0-1   Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, functions, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
AaBbCc123	What you type, contrasted with on-screen computer output	`system%` **`su`** `password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-1   Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

XIL 1.3 New Features

This document lists the new features in XIL 1.3.

Backward Compatibility

The XIL 1.3 library is fully backward compatible. This means that existing applications should run with XIL 1.3 *without* being modified or recompiled.

MT-Hot

The XIL 1.3 library is MT-hot. You can write multithreaded applications without putting locks around XIL functions. Multiple threads from the API can execute together as long as they don't require data from the same image. The library itself also is MT-hot. It takes advantage of multiprocessor systems without applications having to be rewritten.

Tiled Storage

XIL 1.3 provides the option to store very large images in separate buffers of contiguous memory called tiles. If a region of an image within a tile boundary is needed, only that tile is loaded into memory. The XIL library maximizes the sequence of operations done on a tile while it is loaded into memory, thereby increasing performance.

While using tiled storage is backward compatible with existing XIL applications, in some cases it can cause performance degradations. By default, use of tiling is turned off.

Optionally, you can manipulate XIL images stored as tiles by using the tiling functions listed below. Keep in mind, however, that when tiling is turned on, XIL uses tiling behind the scenes to enhance performance whether or not an application makes explicit calls to tiling functions.

The new tiling functions are:

- `xil_get_tile_storage()`
- `xil_get_tilesize()`
- `xil_set_tile_storage()`
- `xil_set_tilesize()`
- `xil_state_get_default_tilesize()`
- `xil_state_get_default_tiling_mode()`
- `xil_state_set_default_tilesize()`
- `xil_state_set_default_tiling_mode()`

XIL 1.3 includes a new `XilStorage` storage object. `XilStorage` provides access to an image's data storage directly. Unlike the backward-compatible `XilMemoryStorage` structure, the `XilStorage` object supports tiled as well as contiguous storage.

The new storage functions are:

- `xil_get_storage_movement()`
- `xil_get_storage_with_copy()`
- `xil_set_data_supply_routine()`
- `xil_set_storage_movement()`
- `xil_set_storage_with_copy()`
- `xil_storage_create()`
- `xil_storage_destroy()`
- `xil_storage_get_band_stride()`
- `xil_storage_get_by_name()`
- `xil_storage_get_coordinates()`
- `xil_storage_get_data()`
- `xil_storage_get_image()`
- `xil_storage_get_name()`
- `xil_storage_get_offset()`
- `xil_storage_get_pixel_stride()`
- `xil_storage_get_scanline_stride()`

- `xil_storage_is_type()`
- `xil_storage_set_band_stride()`
- `xil_storage_set_coordinates()`
- `xil_storage_set_data()`
- `xil_storage_set_data_release()`
- `xil_storage_set_name()`
- `xil_storage_set_offset()`
- `xil_storage_set_pixel_stride()`
- `xil_storage_set_scanline_stride()`

New Data Type

XIL 1.3 now supports the IEEE 754, 32-bit single-precision floating point data type. Using this data type allows you to develop highly sophisticated scientific imaging applications.

Floating Point Warp Tables

XIL 1.3 supports the use of floating point warp tables in `xil_tablewarp()`. This is in addition to the previous support of signed, short integer fixed-point warp tables, which still may be used. Floating point warp tables provide more fractional precision than fixed-point tables but require more memory.

Temporary Images

XIL 1.3 supports the concept of temporary images. Temporary images are images used as an intermediate step in creating a subsequent image. They may only be written to, and read from, once. Temporary images are particularly advantageous for large images because XIL can release the storage associated with them when it is no longer needed.

The new temporary image functions are:

- `xil_create_temporary()`
- `xil_create_temporary_from_type()`

Storage Formats

XIL_GENERAL

XIL 1.3 supports a new `XIL_GENERAL` storage format. This format allows you the flexibility of specifying each band of a multiband image as a separate memory buffer. Furthermore, each band can have its own scanline and pixel stride.

XIL_BAND_SEQUENTIAL

XIL 1.3 now supports the `XIL_BAND_SEQUENTIAL` format for all data types, not just `XIL_BIT` images.

KCMS Integration

The XIL 1.3 library includes Kodak Color Management System (KCMS™) support. With KCMS integration, you can achieve as close as possible color matching between a display image and the actual stored image.

The new color space functions are:

- `xil_color_correct()`
- `xil_colorspace_create()`
- `xil_colorspace_destroy()`
- `xil_colorspace_get_by_name()`
- `xil_colorspace_get_name()`
- `xil_colorspace_get_type()`
- `xil_colorspace_set_name()`
- `xil_colorspacelist_create()`
- `xil_colorspacelist_destroy()`
- `xil_colorspacelist_get_by_name()`
- `xil_colorspacelist_get_name()`
- `xil_colorspacelist_set_name()`

Additional XIL Functions

In addition, the XIL 1.3 library includes the following new functions.

General Object Functions

- `xil_cis_get_state()`
- `xil_colorspace_get_state()`
- `xil_colorspacelist_get_state()`
- `xil_dithermask_get_state()`
- `xil_get_state()`
- `xil_histogram_get_state()`
- `xil_imagetype_get_state()`
- `xil_kernel_get_state()`
- `xil_lookup_get_state()`
- `xil_roi_get_state()`
- `xil_sel_get_state()`
- `xil_storage_get_state()`

Devices

- `xil_device_set_attribute()`

Double Buffering Support

- `xil_create_double_buffered_window()`
- `xil_get_active_buffer()`
- `xil_set_active_buffer()`
- `xil_swap_buffers()`

Miscellaneous Functions

- `xil_dithermask_get_values()`
- `xil_get_writable()`
- `xil_histogram_create_copy()`
- `xil_interpolation_table_create_copy()`
- `xil_interpolation_table_get_values()`
- `xil_kernel_create_separable()`
- `xil_kernel_get_values()`
- `xil_sel_get_values()`
- `xil_set_data_supply_routine()`

Performance Optimizations

See Appendix A, "Optimizations and Molecules For XIL Version 1.3," for the XIL library performance optimizations in this release.

Introduction to the XIL Library *1*

This introduction discusses these subjects:

- What functionality the XIL library provides
- How the library works with other Sun and third-party libraries

The chapter is divided into the following main sections:

Functions in the XIL Library	*page 1*
XIL Naming Conventions	*page 6*
The XIL Library: A Foundation Library	*page 8*
XIL Library is Multithread Hot	*page 10*
Installation	*page 11*
Examples	*page 11*
Documentation	*page 12*

Functions in the XIL Library

The functions in the XIL library, which you call as C subroutines, fall into two main categories: image processing and image compression. The next two sections introduce these groups of functions, and a third section summarizes the remaining library functions.

≡ *1*

Image Processing Functions

The image processing functions in the XIL library can be broadly grouped under the following headings: arithmetic, relational, and logical functions; geometric functions; and some miscellaneous functions.

Arithmetic, Relational, and Logical Functions

The arithmetic, relational, and logical functions include functions that enable you to:

- Add two images
- Take the maximum pixelwise values of two images
- Take the logical AND of two images
- Multiply an image by a constant

For detailed information about arithmetic, relational and logical functions, see Chapter 11, "Arithmetic, Relational, and Logical Functions."

Geometric Functions

The geometric functions include, among others, routines to:

- Scale images
- Rotate images
- Translate images
- Transpose images
- Warp images

For detailed information about geometric functions, see Chapter 12, "Geometric Functions."

Miscellaneous Functions

The miscellaneous functions enable you to:

- Find the minimum and maximum values in an image
- Produce a histogram for an image
- Threshold an image
- Fill an area in an image
- Filter an image
- Detect image edges

- Dilate or erode an image
- Pass an image through a lookup table to convert between data types
- Perform an interband linear combination
- Blend images
- Paint on an image
- Set and get the values of pixels in an image
- Copy a pattern to an image

For detailed information about miscellaneous functions, see Chapter 13, "Miscellaneous Image Processing Functions."

Image Compression Functions

The image compression functions in the library enable you to compress and decompress images and sequences of images. Several compression and decompression (codec) formats are supported:

- JPEG baseline sequential
- JPEG lossless
- MPEG-1
- H.261
- CCITT Group 3 and Group 4
- Cell
- CellB

JPEG

The JPEG compression standards were developed by the Joint Photographic Experts Group to support the compression of still images, both grayscale and color. Although not specifically designed for the compression of sequences of images, or movies, JPEG compressors are also used frequently for that purpose. The JPEG baseline sequential compressor is a *lossy* compressor, which means that it compresses an image in such a way that when the compressed data is decompressed, the decompressed image and the original image may not match exactly. On the other hand, with a *lossless* JPEG compressor, the decompressed image matches the original image pixel for pixel.

For more information about JPEG image compression, see Chapter 16, "JPEG Baseline Sequential Codec." For more information about lossless JPEG compression and decompression, see Chapter 17, "JPEG Lossless Codec."

≡ *1*

MPEG-1

The MPEG-1 video compression standard was developed by the Moving Picture Experts Group. The goal of MPEG-1 was to compress full-motion video and the associated audio at the rate of about 1.5 Mbits/s. This is approximately the rate at which data can be read from a CD-ROM, so MPEG-1 compressed video is a good choice for use in interactive multimedia applications.

Note – The current release of the XIL library includes an MPEG-1 *decompressor only.* A compressor can be obtained from a third party.

For more information about the MPEG-1 image compression, see Chapter 19, "MPEG-1 Codec."

H.261

The H.261 compression-decompression scheme was developed by the International Telegraph and Telephone Consultative Committee (CCITT). An H.261 video codec is intended to be used to compress and decompress video data sent over Integrated Services Digital Network (ISDN) lines. Thus, the codec is suitable for use in video telephony and videoconferencing applications.

Note – The current release of the XIL library includes an H.261 *decompressor only.* A compressor can be obtained from a third party.

For more information about the H.261 image compression-decompression scheme, see Chapter 18, "H.261 Codec."

CCITT Group 3 and Group 4

The CCITT Group 3 and Group 4 compression standards were developed by the International Telegraph and Telephone Consultative Committee to enable facsimile machines to compress and decompress digitized documents. Now, Group 3 and Group 4 compressors and decompressors are also used for general document storage and retrieval.

For more information about the CCITT Group 3 and Group 4 image compression, see Chapter 20, "CCITT Group 3 and Group 4 Codecs."

Cell

The Cell image compression technology, which was developed by Sun, has been optimized for the rapid decompression and display of images on simple hardware. Therefore, the Cell codec is able to achieve reasonable display quality on indexed-color frame buffers. The initial focus of the Cell technology is on Sun-to-Sun communications, where the benefits of fast decoding outweigh the benefits of standards. Possible areas of application include media distributions on CD-ROM and multimedia mail.

For more information about Cell image compression, see Chapter 21, "Cell Codec."

CellB

The CellB codec, which derives from its Cell counterpart, is intended for use primarily in videoconferencing applications. It features greater balance between the time spent compressing and decompressing images than the Cell codec. The CellB codec's strengths include:

- Software compression at interactive rates
- Very fast decoding and display, especially on indexed-color frame buffers
- Low rates of CPU use
- Good quality output

For more information about CellB image technology, see Chapter 22, "CellB Codec."

Additional XIL Functions

The image processing and image compression functions mentioned above are the heart of the XIL library. However, the library also contains functions that perform the following tasks.

Copying an Image to the Display	*page 153*
Rescaling an Image	*page 158*
Casting an Image From One Data Type to Another	*page 160*
Dithering an Image	*page 161*

≡ 1

Color Space Manipulation	page 178
Black Generation	page 186
Error Handling	page 189

As you can see, the XIL library contains a broad range of functions. The next section explains why the XIL library was designed this way and how it relates to other Sun, and to third-party, libraries.

XIL Naming Conventions

To make it easier to understand what each XIL library function does, XIL uses a special self describing convention for naming library functions.

Format

The format of a library function is:

`xil_`*<object_name>*`_`*<method>*`()`

Each function name component has the meaning shown in Table 1-1.

Table 1-1   Function Name Components

Name Component	Meaning
`xil`	Is the prefix string of a function name. All XIL functions have this same prefix.

object_name	Identifies which XIL object is being acted upon by this function. The XIL objects are:
	`cis` (compressed image sequence)
	`colorspace`
	`colorspacelist`
	`device`
	`dithermask`
	`error`
	`histogram`
	`image`
	`imagetype`
	`interpolation_table`
	`kernel`
	`lookup`
	`roi` (region of interest)
	`sel` (structuring element)
	`storage`
	`state`

method	Is the action the function takes on the object.

Example Function Names

Table lists some examples of XIL functions and describes what they do.

Table 1-2   Example Function Names

Function	Description
`xil_roi_add_rect()`	Adds a rectangle to an XIL region of interest (ROI) object
`xil_storage_get_name()`	Gets the name of the XIL storage object
`xil_roi_create()`	Creates an ROI object

Functions Common to All XIL Objects

There is a basic set of functions that an application can perform on objects. These function names will become familiar to you as you are introduced to XIL objects in this guide:

- `xil_<`*object_name*`>_create()`
- `xil_<`*object_name*`>_destroy()`
- `xil_<`*object_name*`>_get_by_name()`
- `xil_<`*object_name*`>_get_name()`
- `xil_<`*object_name*`>_set_name()`
- `xil_<`*object_name*`>_get_state()`

Naming Convention Unique to XIL Image Objects

The XIL image object is special in that the object component of the function name is omitted: the name consists of the prefix and method only. For example, the `xil_create()` function creates an XIL image object.

Understanding the image object naming convention results in functions that appear similar but, in fact, function differently. Take the following functions as examples:

- `xil_device_set_attribute()`
- `xil_set_device_attribute()`

The former function sets an attribute of the device object; the latter sets the device attribute of an image object.

The XIL Library: A Foundation Library

The XIL library is what Sun calls a foundation library. Such a library is an application programming interface (API), but has special characteristics that distinguish it from other APIs. For example, a foundation library must deal with hardware dependencies, while most APIs are hardware independent. In addition, a foundation library is geared toward a broad area of application, such as imaging and video (the XIL library) or graphics (the XGL library),

while most APIs have a narrower scope. For example, APIs based on the XIL library might address such areas as document imaging, medical image processing, and image compression (see Figure 1-1).

Figure 1-1   Foundation Libraries and Application Programming Interfaces

This extra layer of software is desirable for several reasons. Of note are the following:

- It minimizes code duplication.

  A number of APIs—developed either by Sun or third parties—may require the same functionality. For instance, an API directed at the commercial document imaging market and another directed at the facsimile market both must be able to compress and decompress image data according to CCITT standards. If the code that handles this compression is part of a foundation library, the two APIs don't have to create two functions that do the same thing.

- It provides a single interface to devices.

A foundation library provides a single interface to the hardware with which the library interacts. For example, the XIL library provides an interface to video cards, scanners, imaging accelerators, frame buffers, and printers. Any applications or APIs written on top of the XIL library access these devices using XIL routines.

- It permits object sharing.

  A foundation library allows multiple APIs to share objects in a simple way. For instance, because XIL defines memory formats for different types of images, all APIs whose imaging functionality is based on XIL routines can operate on these images without having to convert the format of any image data.

- It enables extensibility.

  Because an API also may need to provide capabilities that the XIL library does not support, it is possible to *export* data from the XIL domain into *application space*. There, an application or API can do whatever processing of the data it needs to do. When this processing is complete, the data can be *imported* so that the program can process the data using only XIL functions.

XIL Library is Multithread Hot

In this release, the XIL library is MT-hot. You can write multithreaded applications without putting locks around XIL functions, because XIL protects the objects of its own data structures. If, for example, two threads call `xil_set_origin()` at the same time, XIL guarantees that each call occurs atomically. Multithreaded applications, however, are still responsible for managing their own variables and data to prevent race conditions.

Note – The error handler of a multithreaded application may only make XIL calls on the object that was returned to the error handler; doing otherwise could lead to a deadlock.

Installation

With the release of XIL 1.3, the `XILHOME` environment variable no longer exists. The various components of the XIL product are now installed into standard Solaris locations that applications or makefiles access by default. The user or application developer should *not* set `XILHOME`. The user does not need to set any other environment variables to use XIL 1.3.

For the purposes of information only, the XIL runtime libraries are now installed in `/usr/lib`, the pipelines that XIL picks up directly are installed in `/usr/openwin/lib/xil`, and the header files are installed in `/usr/openwin/include/xil` but a link exists as `/usr/include/xil`.

When installed, the manual pages are installed in `/usr/openwin/man`.

Examples

The `/usr/openwin/demo/xil/` directory contains several example programs that are discussed in this book.

Table 1-3   Example Programs Provided With the XIL Developer's Release

Example Program	Description
`example1`	Reads a grayscale image from a file and displays the image in an Xlib™ window. See Chapter 2, "Basic XIL Program."
`example2`	Performs the same functions as example1 except this program copies its source image tile-by-tile into XIL memory. See Chapter 5, "XIL Program That Uses Tiling."
`display`	Demonstrates how to display images of different data types and with different numbers of bands in an Xlib window. See Chapter 8, "Preparing Images for Display."
`encode`	Demonstrates how to create movies using XIL functions. See Chapter 15, "Compressing and Decompressing Sequences of Images."
`xilcis_example`	Demonstrates how to play back movies using XIL functions. See Chapter 15, "Compressing and Decompressing Sequences of Images."
`xil_to_xgl`	Processes an image using both XIL and XGL 3.*x* calls in the same program. See Appendix C, "XIL-XGL Interoperability."

☰ *1*

Documentation

The *XIL Programmer's Guide* is included in the AnswerBook2™ on-line documentation. To view the on-line version of this guide, see the Solaris XIL 1.3 AnswerBook located at the following web site:

`http://docs.sun.com`

Basic XIL Program 2≡

This chapter introduces programming with the XIL library by looking at a simple XIL program called `example1`. The program illustrates functions you typically will find in most XIL programs.

After briefly discussing how to build and run `example1`, this chapter walks you through the program's code. The chapter concludes with a brief discussion on building XIL programs.

This chapter has the following main sections:

☰ *2*

Program Overview

The `example1` program reads an 8-bit grayscale image from a file and displays it in an X window. The program also enables you to take the one's complement of the image displayed in the window, which produces a negative of the image.

All XIL applications process images in some way. As you read this book, you will learn that images can come from a variety of sources. The `example1` program gets its image input from two disk files: a header file, `brainscan.header`, and an associated data file, `brainscan.data`. The header file provides XIL with the image attributes it needs. The data file stores the image pixel values.

Running the `Example1` *Program*

The source files for the `example1` program can be found in the `/usr/openwin/demo/xil/example1` directory. Table 2-1 lists these source files and indicates what the code in each file does.

Table 2-1   Source Files for `Example1`

Source File	Description
`example1.c`	Contains `main()`, which prepares different types of source images to be copied to various types of display images
`fileio.c`	Reads an image from a file and loads the image data into an XIL image
`window.c`	Creates the program's X colormap and manages that colormap

The `example1` directory also contains an image file that the program should use as input and a `Makefile` you can use to build the program.

To run `example1`, follow these steps:

1. **Change your working directory to** `/usr/openwin/demo/xil/example1`**.**

2. **To build the program, type** `make`**.**

3. **To execute the program from the directory in which you built it, type:**

```
% example1 brainscan.header
```

An X-ray of a brain is displayed in an X window.

4. **To take the one's complement of the image in the window, move your pointer into the program's window and press any key on your keyboard.** A second keypress will restore the image to its original state (by causing a second bitwise negation to be performed).

5. **To exit the program, move your pointer into the program's window and click any mouse button.**

Header Files and Initializing

All XIL programs must include the approproiate XIL header files and XIL library initialization, as explained in the sections that follow.

Including Header Files

The first XIL-related item in `example1.c` is a preprocessor directive that includes the `xil.h` header file.

All XIL programs must include this header file and, therefore, must include the line:

```
#include <xil/xil.h>
```

This header file defines a number of symbolic constants and enumeration constants and contains function prototypes for all of the API functions in the XIL library.

Note – All other header files are private and may be renamed or removed in subsequent releases.

Initializing the Library

If you skip down to the `main` routine in `example1.c`, you'll note that after reading command-line arguments, the example calls the routine `xil_open()`.

```
XilSystemState state;
...
state = xil_open();
if (state == NULL) {
    /* XIL's error handler will print an error msg on stderr */
    exit(1);
}
...
```

Note – All XIL programs must call the `xil_open()` function to initialize the library before using other functions from the XIL library.

As part of the initialization process, `xil_open()` returns an object of type `XilSystemState` called `state`. The library uses this data structure to keep track of information such as what XIL images (and other XIL data structures) have been created during the current session and what the program's error handling routine is. You'll notice later in the program that `state` must be passed to any XIL function that creates an image.

Note – It is possible to have multiple `XilSystemState` objects, which allows you to use XIL routines and routines from an API built on top of the XIL library in the same program.

Program Tasks

The `example1` program performs three main tasks, namely

- It acquires a disk-based input image. From this input, the program creates an XIL memory-based image.

- It creates an output image. Because the program displays its output, it creates a display image, which is written to the screen frame buffer and displayed in an X window.

- It performs some XIL functions using the memory-based image as the source and the display image as the destination. When the user presses a key, the program takes the one's complement to display the negative of the brain scan image.

Each of these tasks is described in detail in the sections that follow.

Acquiring an Input Image

As mentioned previously, images processed by XIL can come from a variety of sources, and how you bring an image into an application depends on its source. The `example1` program considers the single case in which a program gets its input image from a disk file. To read such a file, the program must perform the sequence of steps outlined below. It must

1. **Read the image header file (or take whatever steps are appropriate) to determine the image's width, height, number of bands, and data type.**

2. **Create an XIL image of the correct dimensions and data type.**
   This image will serve as an empty container in which you place the pixel values from your image data file.

3. ***Export* the XIL image to obtain information about how the image's pixel values will be stored in memory.**
   At all times, an image is either in *XIL space* or *application space.* While an image is in XIL space (is *imported*), you can operate on the image only by using XIL functions. In this state, the image is in under XIL control, enabling it to use acceleration. You do not know the address of the image or its pixel values.

   While an image is in application space (is *exported*), you can access the storage allocated to hold the image's pixel values and use XIL functions to write data in that memory (or to modify existing data). You can also change the pointer to the storage.

   Note – You are not required to export an XIL image. You can operate on an image solely by using XIL library functions. Typically, however, developers want to access the image pixel values to perform tasks such as those listed above.

   ---

4. **Obtain a pointer to the disk-based image data.**
   The `example1` program uses the system call `mmap(2)` to map the contents of `brainscan.data` to application address space. For details on `mmap(2)`, see the manual page.

5. **Create an XIL storage object.**
   An XIL storage object is similar to an XIL image in that it is initially an empty container that will eventually store the image's data format and data pointer. When you create a storage object, your application receives a handle to it for future reference.

6. **Fill in the XIL storage object.**
   Fill in the XIL storage object describing the format and location of the disk-based image data.

7. **Copy the disk-based image data to your XIL image.**

8. **Import the XIL image.**
   This passes control of the image over to XIL, giving XIL the flexibility to move the image to a hardware accelerator.

The sections below discuss each of these steps in detail and show how the steps are performed in `example1`. The code fragments shown are taken from the `fileio.c` source file.

Step 1: Reading the Header File

You need to read the header of your image file to determine the following attributes of the image:

- Width in pixels
- Height in pixels
- Number of bands, or channels
- Data type

This information is required to create an XIL image in which to store the image file's pixel values. How you get this information from your image file's header depends on the format of that header.

The contents of the image header file, `brainscan.header`, are shown here.

```
512 512 8 1
brainscan.data
```

This code fragment from the `load_example_image_file()` function shows
how the example program handles getting the necessary image attributes from
the header.

```
XilImage
load_example_image_file(XilSystemState state, char* pathname)
{
    /*
     *  XIL Image information and dimensions
     */
    XilImage     image;
    unsigned int width;
    unsigned int height;
    unsigned int nbands;
    unsigned int depth;
    XilDataType  datatype;
...

    /*
     *  Read the header information
     */
    fscanf(header_file, "%d%d%d%d", &width, &height, &depth, &nbands);
    fscanf(header_file, "%s", datafile_pathname);

    /*
     *  Convert depth to XIL datatype
     */
    switch(depth) {
      case 1:     datatype = XIL_BIT;   break;
      case 8:     datatype = XIL_BYTE;  break;
      case 16:    datatype = XIL_SHORT; break;
      case 32:    datatype = XIL_FLOAT; break;
      default:
        fprintf(stderr, "unknown data depth %d\n", depth);
        return NULL;
    }
```

The `load_example_image_file()` routine is passed the name of the image file, opens the file, and reads the four integers contained in the first line of the header. These integers represent the width of the image, its height, its *depth* (number of bits used to represent one band of one pixel), and the number of bands in the image.

You can pass the width, height, and number-of-bands arguments directly when you call the routine that creates your XIL image (see "Step 2: Creating an XIL Image"). However, before making that call, you must use the depth value to set a variable of the enumeration type `XilDataType`. The program switches on the value of `depth` to set this variable to one of the following constants: `XIL_BIT`, `XIL_BYTE`, `XIL_SHORT`, or `XIL_FLOAT`. These enumeration constants correspond to the four types of data the XIL library supports: 1-bit data, 8-bit unsigned data, 16-bit signed data, and 32-bit, single-precision IEEE floating point data.

Because the program is provided a single-band, 8-bit image as input, the variable `datatype` is set to `XIL_BYTE`.

Note – The other item read from the header file (by the second `fscanf` statement) is the name of the disk-based image data file that contains the actual pixel values. The data file is accessed in "Step 4: Obtaining a Pointer to the Disk-Based Image Data" on page 22.

Step 2: Creating an XIL Image

Once you have read the appropriate values from your image file's header, you can create your XIL image by calling `xil_create()`.

```
...
XilImage image;
...
/* Create the image to read the data into */
image = xil_create(state, width, height, nbands, datatype);
...
```

With the exception of `state`, the parameters being passed to `xil_create()` are the variables set in "Step 1: Reading the Header File" on page 18. The `state` parameter is the data structure returned by the call to `xil_open()` earlier in the program.

The return value of `xil_create()` is a handle to an XIL image (an object of type `XilImage`). This is the image you export.

Step 3: Exporting the Image

You now need to export the image you just created to get some information about how the image's pixel values will be stored in memory. The `example1` program uses the following code fragment to accomplish these tasks.

```
...
XilImage image;
...
    /*
     *  Export image for access and control of the image data
     */
    if(xil_export(image) == XIL_FAILURE) {
        /*
         * error msg printed to stderr
         */
...
```

The call to `xil_export()`, whose only parameter is a handle to the image, marks the image as exported. By exporting the image, the application gains access to information about the image that enables it to modify the image directly.

For example, the application can get a pointer to the location at which the image's pixel values will be stored. For further information about exporting images, see "Types of XIL Images" on page 44.

Note – Any time you plan to use an image's data, you must export the image by calling `xil_export()` and then get pointers to the data. The pointers are valid only until the image is again imported. For more information, see "Types of XIL Images" on page 44.

Step 4: Obtaining a Pointer to the Disk-Based Image Data

The `example1` program uses the system call `mmap(2)` to map the contents of `brainscan.data` to application address space. The `mmap(2)` call returns a data pointer to the starting address of the disk-based image file. The code fragments related to memory mapping are shown below.

```
...
    /*
     *  Data loading variables
     */
    int         data_fd;
    void*       mmap_ptr;
    struct stat statbuf;
...
    /*
     *  Get the size of the data file.
     */
    if(fstat(data_fd, &statbuf) == -1) {
        perror("stat of datafile:  ");
        close(data_fd);
        return NULL;
    }

    /* Memory map the data file */
  mmap_ptr = mmap(NULL, statbuf.st_size, PROT_READ, MAP_SHARED, data_fd, 0);
...
```

Step 5: Creating an XIL Storage Object

XIL storage is similar to an XIL image in that it is initially an empty container that will store information about the image's data format. You need to create a storage object for your XIL image by making the XIL function call shown in this code fragment. The `xil_storage_create()` function returns a storage handle to your application for reference in subsequent XIL calls.

```
...
    /*
     *  Construct storage object for setting storage on the image
     */
    storage = xil_storage_create(state, image);
...
```

Note that when the storage object is no longer needed, the program destroys it with the `xil_storage_destroy()` function. This function is passed the storage handle as its only parameter.

Step 6: Describing the Storage to XIL

You must use XIL functions to set values in, and to get values from, the storage object. In this case, you are setting values about the disk-based image's data layout. The disk-based image data file that `example1` reads as input is a single-band (grayscale) image. This particular file is stored in pixel-sequential format. You can read all the details of XIL data types and supported storage formats in Chapter 3, "XIL Images." For now, you need to know that XIL supports this image format. To set up this format for the XIL image you created (see "Step 2: Creating an XIL Image"), you must provide XIL with three pieces of information:

- The *pixel stride* (distance in bytes to the next pixel on the same scanline) of the image

- The *scanline stride* (distance to the same pixel on the next horizontal scanline) of the image

- The pointer to the disk-based image data

This code fragment shows the functions you need to call to describe the storage layout to XIL.

```
...
    /*
     *  Describe the storage to XIL.
     */
    xil_storage_set_pixel_stride(storage, 0, nbands);
    xil_storage_set_scanline_stride(storage, 0, nbands*width);
    xil_storage_set_data(storage, 0, mmap_ptr, NULL);
...
```

Using XIL Functions For Pixel-Sequential Format

In each of the XIL functions shown above, the storage argument is a handle to the XIL storage you created. The second argument is always 0 for pixel-sequential format.

The final arguments of `xil_storage_set_pixel_stride()` and `xil_storage_set_scanline_stride()` are the pixel stride (`nbands`) and scanline stride (`nbands*width`), respectively. These values were obtained when `brainscan.header` was read in step 1 (see "Step 1: Reading the Header File"). Pixel stride is equivalent to the number of bands in the image. Likewise, scanline stride is the number of bands times the width. On some hardware, a scanline can include extra bytes of padding for boundary alignment. By specifying `nbands*width`, XIL knows exactly how many bytes of the scanline contain actual image data.

The third argument of the `xil_storage_set_data()` function is the pointer to the image data that was returned when the image data file was memory mapped in step 4 (see Step 4: Obtaining a Pointer to the Disk-Based Image Data"). The final argument is not used but provides for an optional callback routine, which XIL could use to tell the user when it is done with the data.

Using XIL Functions For Other Formats

The storage setting functions described above work for setting up the pixel-sequential format. However, you need to determine which storage setting functions to use for other formats XIL supports. Say, for instance, the `example1` program read an RGB image from a file and the 8-bit (`XIL_BYTE`) RGB values in the file were stored in band-sequential format. In this case, you would need to provide XIL with different storage layout information. In

addition to setting the pointer to the data file with `xil_storage_set_data()` and the scanline stride with `xil_storage_set_scanline_stride()` as described earlier, you would need to set the *band stride* (distance in bytes to the same pixel in the next band). You can do this with the `xil_storage_set_band_stride()` function. See the manual pages on these functions for details. Also see Chapter 4, "XIL Storage," for more information on storage formats.

Note – You do not explicitly set the storage type. XIL determines the type from the values you set in the storage object.

Step 7: Copying Disk_Based Image Data to Your XIL Image

Your XIL program can use a number of methods to copy the data from the image data file to the XIL image. It can use a simple copy function or more advanced techniques that require you to create routines that write the disk-based data into memory tile by tile.

The `example1` program uses a simple method: calling the "convenience" function `xil_set_storage_with_copy()`. Chapter 5, "XIL Program That Uses Tiling," describes one of the more advanced methods that gets tile storage on an image with a function called `xil_get_tile_storage()`.

Note – Historically, XIL applications used the `xil_get_memory_storage()` and `xil_set_memory_storage()` functions to get and set an `XilMemoryStorage` structure to access or change an exported image's formatting information. `xil_get_memory_storage()`filled in an `XilMemoryStorage` structure from which your application could access the image's pixel values; `xil_set_memory_storage()` allowed your application to set an image's memory storage. These functions are still supported. However, they are not compatible with functions in the current version of the

---

XIL library that are used with images stored as tiles. Table 2-2 lists the incompatible functions. It is recommended that you use the new functions in new applications that you write.

Table 2-2 Incompatible XIL functions

Old XIL Functions	New XIL Functions
xil_get_memory_storage()	xil_get_storage_with_copy()
xil_set_memory_storage()	xil_set_storage_with_copy()
	xil_get_tile_storage()
	xil_set_tile_storage()
	xil_release_storage()

In example1, the xil_set_storage_with_copy() function uses the information in the storage object previously set in step 6 (see "Step 6: Describing the Storage to XIL") to copy the disk-based image to the memory-based XIL image in the specified layout. The function knows where to write each pixel value it reads from the disk file—even if there are unused bytes between pixels in the XIL image—because it knows the pixel stride and scanline stride.

With this function, the file data are conveniently copied to the XIL image as shown in this code fragment.

```
...
XilStorage storage;
...
if(xil_set_storage_with_copy(image, storage) == XIL_FAILURE) {
        /*
         * Print an error msg to stderr
         */
...
```

Step 8: Importing the Image

Once you have read image data from a file into an XIL image, you should import the image so you can accelerate imaging operations where possible. You import the image with a call to `xil_import()`.

```
...
XilImage image;
...
xil_import(image, TRUE);
...
```

Note – Importing an image is the opposite of exporting it. When you import an image, any pointer to the image's pixel values you had while the image was exported becomes invalid, so you can no longer modify the image directly. You must make all modifications using XIL functions. To access the data again, you must export the image again and get new pointers to the data. For more information, see "Types of XIL Images" on page 44.

The argument `image` is a handle to the image being imported, and TRUE is a change flag. The flag indicates that the image was modified while it was exported. In this case, it was filled with pixel values.

Creating an Output Image

Because `example1` displays its output, its output image must be a special type of image called a *display image*. This type of image is written to a screen's frame buffer and is displayed in an X window.

To create this display image, the example program:

- Creates an X window the size of the image it wants to display.

  See "Creating an X Window," for detailed information.

- Uses the `xil_create_from_window()` function to create a display image from the X window just created. This routine turns the X window into an XIL image that can be used as a destination for an XIL operation.

  See "Creating a Display Image" on page 29 for detailed information.

Creating an X Window

Most of the Xlib code in this example appears in the `open_window()` function defined in the `window.c` source file. This function:

- Establishes a connection with the X server

- Finds the best X visual to use when displaying a grayscale image on a particular display

- Creates the X colormap that will be used when the image is displayed

- Creates the X window in which the image will be displayed

- Maps the X window

To create the X window, the `open_window()` function calls `XCreateWindow()`.

```
XilImage
open_window(XilSystemState state,
            unsigned int   width,
            unsigned int   height,
            unsigned int   nbands,
            XilDataType    datatype,
            XVisualInfo*   vinfo,
            Xil_boolean    prefer_depth_24)
...
Display*             xdisplay;
Window               xwindow;
...
xwindow = XCreateWindow(xdisplay,DefaultRootWindow(xdisplay),
                        0, 0, width, height, 0,
                        vinfo->depth, InputOutput, vinfo->visual,
                        attribmask, &winattr);
...
```

The return value of `XCreateWindow()` is an X window ID, which is used in the call to `XMapWindow()`, as shown below.

```
...
/*
 *  Make the window visible -- XIL can only create from mapped
 *  and visible windows.
 */
XMapWindow(xdisplay, xwindow);
...
```

Once the X window is mapped, the XIL display image can be created. See "Creating a Display Image."

Creating a Display Image

Before you can use an XIL function to display an image, you must create a display image with the `xil_create_from_window()` function. This code fragment from `window.c` turns an X window into a special type of XIL image so the window can be used as the source or (as in this case) the destination image for an operation.

```
...
XilImage
open_window(XilSystemState state,
...
    Display*            xdisplay;
    Window              xwindow;
...
    /*
     *  Create the XIL display image
     */
  display_image = xil_create_from_window(state, xdisplay, xwindow);
```

`xil_create_from_window()` returns a handle to the newly created display image. The parameters you pass to the routine are a handle to the system state data structure returned by `xil_open()`, a pointer to a structure describing your display, and the ID of your X window.

At this point, the program is ready to do some image processing of the source image it read earlier.

Processing an Image

With handles to a source and destination image, `example1` is now ready to use XIL functions to process the source image and eventually write the processed image to the display. The functions used in this part of the program can be thought of as serving two purposes.

One purpose is to alter the source image so that it can be written to the destination image. For example, in the sample program, the source image contains 8-bit data, but the display image may be just 1-bit deep (if the screen on which the image is to be shown is monochrome). In this case, the source image must be converted to 1-bit data before it can be copied to the display image. This kind of conversion may require that you create an intermediate image.

The second purpose is to change the source image to alter its appearance in some way. For instance, you might darken the image or enhance the edges in the image. The `example1` program enables you to take the one's complement of its source image.

Making Source and Destination Images Compatible

The `example1` program handles only one type of source image: an 8-bit grayscale image. However, it handles several types of display images. It may write its output to:

- A 3-band `XIL_BYTE` display image whose X colormap is read only.

- A 1-band `XIL_BYTE` display image. There are actually two cases here because the image's X colormap (grayscale) may be read-write or read only.

- A 1-band `XIL_BIT` display image whose X colormap is read only.

In only one of these cases can the source image be copied to the display image as is. The following sections briefly describe how each case must be handled.

Three-Band, Eight-Bit Destination—Read-Only Colormap

In this case, the source image contains one band, and the display image contains three. To solve this problem, The following code fragment from `example1.c` creates a temporary 3-band `XIL_BYTE` image, called `retained_image`, and copies the source image to each band of the temporary image.

```
...
if (vinfo.class == TrueColor) {
    /* Copy the grayscale image into each of the bands */
    retained_image = xil_create(state, width, height, 3,
        XIL_BYTE);
    band0 = xil_create_child(retained_image, 0, 0, width,
        height, 0, 1);
    band1 = xil_create_child(retained_image, 0, 0, width,
        height, 1, 1);
    band2 = xil_create_child(retained_image, 0, 0, width,
        height, 2, 1);
    xil_copy(image, band0);
    xil_copy(image, band1);
    xil_copy(image, band2);
}
...
```

Note that the example program actually creates three band children of the temporary image and copies the source image to those children. Once this has been done, `retained_image` can be copied to the display image.

One-Band, Eight-Bit Destination—Read-Write Colormap

In this case, the program could write the source image directly to the display image. However, doing that would require that the program create and use a colormap with 256 entries. If such a colormap were installed, it might cause other X clients, including the window manager, to display their output in false colors.

To avoid this problem, the example program creates an X colormap of 256 entries, but writes its grayscale ramp in the last 216 entries. The program then uses a call to `xil_rescale()` to ensure that all the values in the image to be displayed fall between 40 and 255.

One-Band, Eight-Bit Destination—Read-Only Colormap

In this case, the example program can write the source image directly to the display image because the program's X colormap is a read-only grayscale ramp.

One-Band, One-Bit Destination—Read-Only Colormap

In this case, the example program creates a single-bit XIL image (using `xil_create()`) and then uses the function `xil_error_diffusion()` to dither the 8-bit image to a 1-bit image.

Additional Image Processing

Besides processing the source image so that it can be written to the display, `example1` enables you to take the one's complement of the brain scan by pressing any key while your pointer is in the program's window. The following code fragment from `example1.c` performs this operation.

```
...
    Display*        xdisplay;
...
    XEvent          xevent;
...
    /*
     * Refresh window as necessary till program terminates
     */
    while(1) {
        XNextEvent(xdisplay, &xevent);

        if(xevent.xany.type == KeyPress) {
            xil_not(retained_image, retained_image);
...
        if(vinfo.class == GrayScale) {
         xil_add_const(retained_image, &offset, retained_image);
        }
        xil_copy(retained_image, display_image);
        } else if(xevent.xany.type == Expose) {
            xil_copy(retained_image, display_image);
        } else if(xevent.xany.type == ButtonPress) {
            break;
...
```

If the example gets a keypress event, it calls `xil_not()` to take the one's complement of the source image. This single statement works for all cases except the case where the program is using a GrayScale X visual. In that case, the values in `retained_image` before the call to `xil_not()` fall in the range 40 to 255. After the call to `xil_not()`, they fall in the range 0 to 215. Therefore, before the one's complement is displayed, the example adds 40 to each value in the image using the function `xil_add_const()`.

You may want to experiment with removing from the example the code used to take the one's complement of the brain scan and replacing it with code that performs another image-processing operation. You might try any of the

arithmetic and logical operations that work with one source image and a constant. These operations are discussed in Chapter 11, "Arithmetic, Relational, and Logical Functions." Or you might try replacing the logical negation with a geometric operation. The XIL geometric operations are the subject of Chapter 12, "Geometric Functions."

Closing the Library

Before exiting, all XIL programs should deallocate the memory associated with any XIL data structures that haven't already been destroyed.

```
...
XilSystemState state;
...
    /*
     * We are responsible for destroying all of the XIL
     * objects we created. Like free(), XIL supports
     * destroying NULL objects.
     */
    xil_destroy(image);
    xil_destroy(display_image);
    xil_destroy(bit_image);
    xil_destroy(band0);
    xil_destroy(band1);
    xil_destroy(band2);

    if(retained_image != image) {
        xil_destroy(retained_image);
    }

    xil_lookup_destroy(colorcube);

    /*
     * We do not destroy the XilKernel because we didn't
     * create it. We aquired it using the
     * xil_kernel_get_by_name() call.
     */

    xil_close(state);

...
```

Note – If your program creates a display image, you must destroy it with a call to `xil_destroy()` before you break your connection with the X server and display (`XCloseDisplay()`).

Building an XIL Program

The `example1` directory, which contains the source files for `example1`, also contains the `Makefile` you use to build the program. The library line is shown below.

```
LD_LIBS    += -lxil -lX11
```

When you link your XIL program, you must include the XIL library (`-lxil`). The X window library (`-X11`) can be omitted if your program doesn't display output in an X window.

Note – XIL applications should not compile using the `-fnonstd` flag. XIL's geometric code depends on IEEE standard handling of floating point exceptions. When divide by zero is encountered, a valid value is generated, which is processed accordingly. If, however, an application is compiled with the `-fnonstd` flag, it may crash in XIL's code with an exception.

Conditionally Compiling Code for Different XIL Versions

Beginning with release 1.2, the XIL library lets you conditionally compile code so you can take advantage of new interfaces in a current release while still supporting earlier releases of the library. To permit compile-time decisions, the library uses preprocessor directives to define symbols that identify the current XIL major and minor version numbers; these symbols—`XIL_API_MAJOR_VERSION` and `XIL_API_MINOR_VERSION`—are available to your application when you include the `xil.h` header file (see "Including Header Files" on page 15).

By convention, the major version number is the numeral preceding the decimal point in the XIL version number, and the minor version number is the numeral following the decimal point. Table 2-3 shows what the major and minor version numbers would be for various XIL releases.

Table 2-3 Examples of Major and Minor Version Numbers

XIL Version Number	Major Version Number	Minor Version Number
1.2	1	2
1.3	1	3
2.0	2	0
2.1	2	1

By using these symbols, you can maintain a single source-code file that can be compiled against different versions of the XIL library. For example, version 1.2 of the XIL library introduced `XilDevice` objects, which can be used to initialize device attributes before creating the corresponding device image; earlier versions of the library can't use this object. The following code fragment shows how you might use the `XIL_API_MAJOR_VERSION` and `XIL_API_MINOR_VERSION` symbols to write code that uses an `XilDevice` object when you compile your application with the 1.2 version of the library, but code that omits the object when you compile with earlier versions:

```
#if (XIL_API_MAJOR_VERSION == 1 && XIL_API_MINOR_VERSION >= 2) ||
    (XIL_API_MAJOR_VERSION > 1)
{
  /*create a device object */
  XilDevice device;
  device = xil_device_create(state, "SUNWrtvc");
  /* initialize the DEVICE_NAME attribute */
  xil_device_set_value(device, "DEVICE_NAME", (void *)devname);
  /* create a device image with the initialized attribute */
  rtvc_image = xil_create_from_device(state, "SUNWrtvc", device);
  xil_device_destroy(device);
}
#else
  /* create a device image without initialized attributes */
  rtvc_image = xil_create_from_device(state, "SUNWrtvc",
                                      (void *) devname);
#endif
```

In this example, a device image with an initialized attribute is created only for XIL versions 1.2 and later; otherwise, the device image is created without initialized attributes.

See "XIL 1.3 New Features" on page xxxv for a discussion of the new features introduced in the current release of the library.

2

XIL Images 3≣

Because all XIL programs work with XIL images, you should not only know how to create an XIL image, but understand exactly what an XIL image is and what its characteristics are. This chapter explains key information to understand about XIL images.

This chapter has the following main sections:

Digitized Image

When you first think about a computer image, you probably think of a picture or scene that has been digitized and is represented as a two-dimensional array of numbers or vectors. For example, the upper-left corner of an **8**-bit, single-band image might look like this:

255	255	255	255	255	255	255	255	255	255	255	255
255	254	254	254	254	254	254	254	254	254	254	254
255	254	253	253	253	253	253	253	253	253	253	253
255	254	253	252	252	252	252	252	252	252	252	252
255	254	253	252	251	251	251	251	251	251	251	251
255	254	253	252	251	250	250	250	250	250	250	250
255	254	253	252	251	250	249	249	249	249	249	249
255	254	253	252	251	250	249	248	248	248	248	248

Figure 3-1   Digitized Image

One component of an XIL image *is* a set of pixel values like those shown in Figure 3-1. However, an XIL image is more than that. It is an object containing many members, one of which is a pointer to a set of pixel values.

Basic XIL Image Attributes

When you use the `xil_create()` function to create an XIL image, you *must* specify four attributes of the image being created: its width, height, number of bands, and data type.

Width, Height, and Number of Bands

These three attributes define the dimensions of an image. An image's width and height are given in pixels. Its number of bands is the number of values needed to describe a single pixel; for example, in a grayscale image, one value describes the gray level of each pixel, so the image is said to have one band. In

an RGB image, three values (red, green, and blue) are required to describe a single pixel, so such an image has three bands. Each band in a multiband image must have the same width and height.

Each of these image attributes is stored as an unsigned 32-bit integer. In this release of the XIL library, the image is only limited by the system's resources.

Note – The three attributes (width, height, and number of bands) are set when you create an XIL image and cannot be changed afterwards.

The functions you use to read the values of these attributes are shown in Table 3-1.

Table 3-1 Functions for Reading an Image's Width, Height, or Number of Bands

Function	Description
xil_get_width()	Gets the width of an image in pixels
xil_get_height()	Gets the height of an image in pixels
xil_get_info()	Gets the width, height, number of bands, and datatype of an image
xil_get_size()	Gets the width and height of an image in pixels
xil_get_nbands()	Gets the number of bands in an image

Data Type

This image attribute specifies the type of data used to represent the value of one band of one pixel. The XIL library supports four data types: 1-bit data, 8-bit unsigned data, 16-bit signed data, and single-precision IEEE floating point data. You establish an image's data type when you create the image by passing to the `xil_create()` function one of the following enumeration constants: `XIL_BIT`, `XIL_BYTE`, `XIL_SHORT`, or `XIL_FLOAT`. These enumerators are of type `XilDataType`.

Note – The single-precision IEEE floating point data type is only available through the new XIL storage access functions, namely: `xil_get_storage_with_copy()`, `xil_set_storage_with_copy()`,

`xil_get_tile_storage()`, and `xil_set_tile_strorage()`. It is not available through the `xil_set_memory_storage()` or `xil_get_memory_storage()` functions.

The library enables you to convert an image from one supported data type to another by providing the `xil_cast()` function.

To read an image's data type, you use the `xil_get_datatype()` function. The XIL library also includes the `xil_get_info()` function, which reads an image's width, height, number of bands, and data type.

Exporting XIL Images

When you first create an XIL image using `xil_create()`, you receive a handle to the image, and the image is in a state called *imported*. While the image is in this state, you can process it only with XIL functions because you don't know, and can't find out, the address at which the image's pixel values are stored. Nor do you know the exact layout of those values in memory.

However, you do have the option of *exporting* the image, using the `xil_export()` function. Once an image is exported, you can obtain a complete description of how its pixel values are stored in memory using XIL functions for getting image storage attributes from the image's associated storage object. (See "Storage Functions" on page 76 for details.) You can also continue to process the image using XIL functions while it is exported; however, you may lose some acceleration.

Note – The `xil_export()` function has very little overhead. It simply sets a flag.

While an image is exported, you can also set its memory storage using the functions for setting attributes on the image's associated storage object. (See "Storage Functions" on page 76 for details.) Defining an image's memory storage in this way would be appropriate, for example, if you had exported an empty image and wanted to fill it with pixel values by setting a pointer to some pixel values in system memory.

After you've finished the work you needed to perform with the image exported, you should import the image using the `xil_import()` function. While imported, the image must be processed using XIL functions.

Note – XIL still supports the `xil_get_memory_storage()` and `xil_set_memory_storage()`functions for getting and setting an image's pixel values in memory. However these functions are not compatible with the functions in the current version of the XIL library that are used for images stored as tiles. See Table 2-1 in Chapter 2, "Basic XIL Program," for a list of the incompatible XIL functions.

Table 3-2 lists and describes the XIL functions related to importing and exporting XIL images.

Table 3-2   Functions for Importing and Exporting Images

Function	Description
`xil_import()`	Imports an image so that XIL functions may be accelerated
`xil_export()`	Exports an image so that you can get or set the image's pointer to its pixel values
`xil_get_exported()`	Determines whether an image is currently exported

Caution – The description of memory storage that you read is valid only as long as your image remains exported. (This is provided that you do not enforce storage organization. For details, see "Enforcing XIL Image Storage Organization" on page 79.) Once you import the image, both the address at which the image's pixel values are located and their layout in memory may change. Trying to access pixel values using an invalid pointer or invalid information about their layout can cause serious problems in your application.

If you plan to export an image to set and get storage attributes, you need to understand XIL image storage formats. See "Storage Formats" on page 68 for details.

Providing Data to XIL Images

In Chapter 2, "Basic XIL Storage," you learned of one way of providing data to an XIL image using the `xil_set_storage_with_copy()` function.

In a number of cases, however, it's not possible to provide a description of the entire image's storage. A tile, for example, may be located across the network or in a file whose data cannot be memory mapped (that is, a compressed image file). In these cases, it is more efficient to provide the tile data to XIL on demand so that only one tile is loaded into memory at a time.

Your application can specify a data supply routine for an image or tile using the `xil_set_data_supply_routine()` function. When an XIL operation needs data for a tile of the image, XIL calls the routine to obtain the memory. The routine is only called when there are no data associated with a tile. So, the first time a tile from the image is needed, your application's routine is called to provide data. From that point on, the data remains under XIL control as if the storage had been set.

Types of XIL Images

There are two basic types of XIL images:

- Memory images
- Device images

Memory Images

The memory image is the most common type of XIL image. When you create this type of image, using the function `xil_create()`, system memory is allocated to store both the image structure that describes the image and the image's pixel values. All operations that do not read or write a device use memory images exclusively.

Device Images

XIL device images represent devices such as displays, cameras, frame grabbers, scanners, and printers. For each device image, the structure describing the image is stored in system memory, but the image's pixel values are read from or written to the device. Device images can be subdivided into images that represent displays and images that represent other devices. Images that represent displays are referred to as *display* images. The main difference between these two subtypes of device image is the way in which they are

created. To create a device image, you pass the name of a device to the `xil_create_from_device()` function. To create a display image, you call the `xil_create_from_window()` function.

You can use a device image as either a source image or a destination image in an XIL operation. When you use a device image as a source image, data is read from the device; similarly, when you use a device image as a destination image, data is written to the device.

The handler for each device is responsible for moving XIL images to or from the device. The current release of the XIL library includes a group of these handlers that support standard devices.

Display Images

Display images are a special subclass of device images. They are very valuable because they eliminate the need for the scenario shown in Figure 3-2.

1. Process a source image and write   2. Call a display routine that copies the
   your output to a temporary image.   temporary image to your display.

Figure 3-2   Copying Data from a Backing Image to the Display (Not Necessary)

Display images enable the XIL library to process a source image and write the results of the processing directly to an X window by using a display image as the destination image for the operation. In other words, functions can write directly to frame-buffer memory, so the need for the backing image is eliminated (see Figure 3-3).

1. Process a source image and write your output
   directly to a display image.

Figure 3-3   Writing the Output of an Operation to a Display Image

To create a display image, you first create an X window. Then, you call the `xil_create_from_window()` function, passing information to it about which display and which window to associate with the display image.

A display image can also serve as a source image in most cases (if the display is readable). Any pixels that are obscured by another window when a read takes place are undefined.

There are some restrictions on the types of images that can be display images. The only possibilities are:

- 1-band images containing 1-bit data elements

- 1-band images containing 8-bit data elements

- 1-band images containing 16-bit data elements

- 3-band images containing 8-bit data elements

Additional XIL Image Attributes

This section discusses the following additional image attributes:

- Origin
- Region of interest
- Color space
- Parent
- Image type
- Synchronization flag
- Readable and writable flags
- Name

Note – It is possible for applications to create additional image attributes. To create and set an attribute, you use the `xil_set_attribute()` function. To retrieve the value of an application-defined attribute, you use the `xil_get_attribute()` function.

Origin

An image's *origin* is a pair of floating-point numbers that define a point in the image's coordinate plane. By default, an image's origin is in its upper-left corner (0.0, 0.0), but you can change the origin using the `xil_set_origin()` function. When an operation is performed, the origins of the source image or images and the destination image are aligned. (For nongeometric operations, the floating-point origin values are rounded to integers for this purpose.)

For all nongeometric operations, the intersection of the source and destination images determines which pixels in the destination image will be modified. This is illustrated in Figure 3-4.

Figure 3-4   Image Origins

Only the shaded area shown in Figure 3-4 is modified in the destination image, and only the shaded area in the source is used by the operator.

Table 3-3 lists and describes the XIL functions you use to read or set an image's origin.

Table 3-3   Functions for Getting and Setting an Image's Origin

Function	Description
xil_get_origin()	Gets both coordinates of an image's origin
xil_get_origin_x()	Gets the *x* coordinate of an image's origin
xil_get_origin_y()	Gets the *y* coordinate of an image's origin
xil_set_origin()	Sets an image's origin

Region of Interest

An XIL region of interest (ROI) is an object of type `XilRoi` that describes a single-bit mask for an image. If the ROI is an attribute of a destination image, it determines which pixels in the destination may be written. Where there are 0's in the ROI, the destination image may not be modified, and where there are 1's, it may be modified. If the ROI is an attribute of a source image, it determines which pixels may be used as input to an operation.

Since an ROI is an attribute of an image, as opposed to a parameter for an operation, two or three ROIs may be involved in a single operation. For example, the operation's source image may have one ROI, and the operation's destination image, another. In this case, the ROI used for the operation is the intersection of the ROIs associated with the source and destination images, as shown in Figure 3-5.

Source ROI   Destination ROI   ROI used for operation

Figure 3-5   Regions of Interest

The dark gray area in the image on the right represents the ROI used for the operation.

If the origin of the source or destination has been set to something other than 0.0, the origins are first aligned. Then, the ROI is determined. This is illustrated in Figure 3-6.

Figure 3-6   Regions of Interest and Origins

The dark gray area represents the ROI used for the operation.

Creating and Destroying an ROI

You can create an ROI in several ways:

- You can create a new ROI object using the `xil_roi_create()` function, which returns a handle to the ROI. When first created, the ROI is empty.

  For information on how to set values in the ROI, see "Building an ROI."

- You can create a copy of an existing ROI using the `xil_roi_create_copy()` function.

- You can get a copy of the ROI associated with an image using the `xil_get_roi()` function.

- You can create an ROI by taking the union or intersection of two existing ROIs. The functions you use for these operations are `xil_roi_unite()` and `xil_roi_intersect()`.

When you are finished with an ROI, use the `xil_roi_destroy()` function to destroy it.

Building an ROI

After creating an empty ROI, you can build a region of interest using the functions listed and described in Table 3-4.

Table 3-4 Functions Used to Build a ROI

Function	Description
xil_roi_add_rect()	Adds a rectangle to the ROI (you specify the width and height of the rectangle and its coordinates)
xil_roi_subtract_rect()	Subtracts a rectangle from an ROI
xil_roi_add_image()	Adds an XIL_BIT image to the ROI (bits that are set in the image are added to the ROI)
xil_roi_add_region()	Adds an X region to an ROI

Note – It's also possible to convert an ROI to a single-bit XIL image or an X region. To perform these tasks, use the xil_roi_get_as_image() and xil_roi_get_as_region() functions, respectively.

Setting an Image's ROI Attribute

Once you have created an ROI and established its contents, you make it an attribute of an image using the xil_set_roi() function. You can get a copy of the ROI associated with a particular image by using the xil_get_roi() function.

Translating an ROI

The library includes a xil_roi_translate() function that moves all the pixels in an ROI that have been set left or right and/or up or down. The prototype for this function is shown below.

```
XilRoi xil_roi_translate(XilRoi roi, int xoffset, int yoffset);
```

The `roi` parameter is the ROI whose pixels you want to translate. The `xoffset` and `yoffset` parameters are integers that represent the number of pixels the ROI should be moved horizontally and vertically. If `xoffset` is positive, the ROI is moved to the right, and if it is negative, the ROI is moved to the left. If `yoffset` is positive, the ROI is moved down, and if it is negative, the ROI is moved up.

The function returns a handle to the new translated ROI.

Naming an ROI

As with all objects, the XIL library enables you to specify a string that serves as the name for an ROI. This naming is useful because it enables you to later get a handle to an ROI by using its name. Table 3-5 lists and describes the ROI naming functions.

Table 3-5   ROI Naming Functions

Function	Description
`xil_roi_set_name()`	Sets the name of an ROI
`xil_roi_get_name()`	Returns a copy of an ROI's name
`xil_roi_get_by_name()`	Returns a handle to an existing ROI that has the name you specify

Note – When you call `xil_roi_get_by_name()`, a pointer to the named object is returned, not a copy of it. Therefore, you should not destroy an ROI obtained in this way.

Color Space

You can manipulate the color spaces associated with an image using either of two methods, depending on your requirements. You can convert an image with a particular associated color space to another image associated with a different color space. This is the default method. Alternately, you can correct the color of a destination image to compensate for individual device characteristics.

Each of these methods is briefly described below. For more information on these methods, see "Color Space Manipulation" on page 178.

Color Conversion

A newly created image does not have a color space associated with it, but you can assign one to it using the `xil_set_colorspace()` function. The possible color spaces are:

- CCIR Rec. 709 RGB
- A linear version of CCIR Rec. 709 RGB
- CCIR Rec. 709 YC_bC_r
- A luminance-only space derived from CCIR Rec. 709 YC_bC_r
- A linear version of the luminance-only space derived from CCIR Rec. 709 YC_bC_r
- CCIR Rec. 601 YC_bC_r
- A luminance-only space derived from CCIR Rec. 601 YC_bC_r
- A YC_bC_r color space defined by Kodak for PhotoCD
- A linear CMY
- A linear CMYK

You associate color spaces with images to convert an image from one color space to another (using the `xil_color_convert()` function). Say, for example, you have an RGB image and want to convert it to the YC_bC_r color space. To do this, you would follow the procedure outlined below:

1. Set the color-space attribute of your RGB image to one of the RGB color spaces listed above.

2. Create an image whose width, height, number of bands, and data type match those of the RGB image.

3. Set the new image's color-space attribute to one of the YC_bC_r color spaces listed above.

4. Use the `xil_color_convert()` function to convert the RGB data from the first image to its YC_bC_r counterpart, and write the converted data to the newly created image.

Color Correction

Each display device has an associated profile containing characteristics about that device. If a color image is created on one device, it may not display the colors with the exact same intensities, hues, and so forth on other devices. To compensate for differences in device characteristics, you can use the `xil_color_correct()` function.

To perform the color correction, `xil_color_correct()` uses Kodak Color Management System (KCMS) functions integrated into the XIL library. For details on KCMS, you should read the *KCMS Application Developer's Guide.*

Say, for example, you create an RGB image on one color monitor. Then you send the image over the network to another machine where you want to print that image such that the printout reflects the same color characteristics as the original image. To accomplish this, you would follow the procedure outlined below:

1. Create a destination image whose width, height, number of bands, and data type match those of your RGB image.

2. Create a color space (`color_space0`) for your source image based on a profile type of your choice. You use the `xil_colorspace_create()` function, which accepts any of three types of profiles based on the `XilColorspaceType` enumeration.

3. Create a color space (`color_space1`) from the profile of the machine in the network that will send your image to the printer.

4. Create a color space (`color_space2`) from the printer's profile for the destination image.

5. Use the `xil_colorspacelist_create()` function to create a color space list from the color spaces you created in steps 2 through 4.

6. Use the `xil_color_correct()` function to color correct the images.

Child Image

The XIL library enables you to create a *child* image (subimage) from an XIL image (the child's *parent*). Once created, a child image is a discrete `XilImage` object. It does not inherit the attributes (ROI, origin, and so forth) of its parent. You can operate on a child image (set ROIs on it, reset its origin, export it, copy it, and so forth) as you would any XIL image (with a few exceptions noted in "Relationship Between a Child and its Parent," below).

Relationship Between a Child and its Parent

The main difference between a child image and an ordinary XIL image (that is, one that is *not* a child) is that a child shares the storage of its parent. A child image does not get a copy of part of the parent's pixel data, but a pointer to that data. See Figure Figure 3-7.

Figure 3-7   Child Image Sharing Storage of Parent

Sharing storage has certain implications:

- If you modify a child image, you modify the contents of the parent.

- Although you can access the data of a child image through an `XilStorage` object using either `xil_get_storage_with_copy()` or `xil_get_tile_storage()`, you cannot set storage on a child using `xil_set_storage_with_copy()` or `xil_set_tile_storage`. (For details on these functions, see Chapter 6, "Handling Input and Output.")

- You can use the `xil_set_tilesize()` function to set the tile size on the parent image but not on the child.

Creating a Child Image

To create the child image, use the `xil_create_child()` function, whose prototype is shown below.

```
XilImage xil_create_child(XilImage src, unsigned int xstart,
    unsigned int ystart, unsigned int width, unsigned int height,
    unsigned int startband, unsigned int numbands);
```

This function returns a handle to the child image. The parameters to the function are defined in Table 3-6.

Table 3-6   Parameters to `xil_create_child()`

Parameter	Description
`src`	A handle to the parent image
`xstart`	The horizontal offset in pixels from the upper-left corner of the source image to the upper-left corner of the child
`ystart`	The vertical offset in pixels from the upper-left corner of the source image to the upper-left corner of the child
`width`	The width of the child image in pixels
`height`	The height of the child image in pixels
`startband`	The offset in bands from the first band in the parent to the first band in the child
`numbands`	The number of bands in the child image (these bands must match contiguous bands in the parent)

Identifying a Parent

A child image has an attribute that identifies its parent image. To retrieve this attribute, a handle to the parent image, use the `xil_get_parent()` function.

Using a Child Image

The following are a few ways you can use child images:

- Take a larger image and operate from one section to another.

  For example, you might copy the contents of one section of an image to another. To do this create two child images from a parent: one representing the source (`child1`) and the other, the destination (`child2`). Then perform an `xil_copy()`. See Figure 3-**8**.

Figure 3-8   Copying Parent Image Contents

- Convert a single-band source image to a multiband destination image.

  Say, for example, you want to display a 1-band `XIL_BYTE` source image as a 3-band `XIL_BYTE` image. To do this, you can create a child image the full width and height of each band of the 3-band `XIL_BYTE` image. Then copy the child image into each band of the 3-band image, as shown here.

```
temp_image = xil_create(state, width, height, 3, XIL_BYTE);
band0 = xil_create_child(temp_image, 0, 0, width, height, 0,
     1);
band1 = xil_create_child(temp_image, 0, 0, width, height, 1,
     1);
band2 = xil_create_child(temp_image, 0, 0, width, height, 2,
     1);
xil_copy(image, band0);
xil_copy(image, band1);
xil_copy(image, band2);
```

  The `display` program, which is located in the `/usr/openwin/demo/xil/display/` directory, uses this technique to display a grayscale image. See "Replicating the Source Image in the Bands of the Destination" on page 144 for more information.

- Operate on a subset of an image (a spatial subset and/or subset of contiguous bands.

  If, for example, you want to do geometric operations on only the center 100x100 pixels of an image, establishing this region as a child image allows you to select origins and ROIs on it.

- Set a ROI on a different region of a multiband image.

  As you recall, an ROI is single-bit mask for an image. See "Region of Interest" on page 49. As such, an ROI can only be set on all bands of an XIL image. To set an ROI on a different region of each band, first create a child of each band.

If you specify overlapping but not coincident sibling images (children of the same parent) as the source and destination for an operation, the operation is performed. However, the XIL library generates a warning message and the results of the operation are undefined.

> **Note** – An exception to this behavior is `xil_copy()`, which detects the overlap and correctly generates the destination image.

In general, you read the attributes of a child image using the same functions you use to read the attributes of a parent. However, the library does contain a `xil_get_child_offsets()` function that is designed for use only with child images. This function returns the values of `xstart`, `ystart`, and `startband` used in creating the child image.

```
void xil_get_child_offsets(XilImage child,
    unsigned int *offset_x, unsigned int *offset_y,
    unsigned int *offset_band);
```

Image Type

For very XIL image, there is a subset of attributes that constitutes its image type. These attributes are the image's width, height, number of bands, data type, and color space. If you call the `xil_get_imagetype()` function and pass it a handle to an image, the routine returns an object of type `XilImageType` that contains only these data elements. You can use this object to create an image of the same type as the one whose image type you just ascertained. To create an image in this way, use the `xil_create_from_type()` function.

The XIL library also contains a number of utility functions that affect image types. These are listed and described in Table 3-7.

Table 3-7   Image Type Utility Functions

Function	Description
`xil_imagetype_get_width()`	Gets the width of an image type in pixels
`xil_imagetype_get_height()`	Gets the height of an image type in pixels
`xil_imagetype_get_size()`	Gets the width and height of an image type in pixels
`xil_imagetype_get_nbands()`	Gets the number of bands in an image type
`xil_imagetype_get_datatype()`	Gets the data type of an image type

Table 3-7   Image Type Utility Functions

Function	Description
`xil_imagetype_get_info()`	Gets the width, height, number of bands, and data type of an image type
`xil_imagetype_set_name()`	Sets the name of an image type
`xil_imagetype_get_name()`	Returns a copy of an image type's name
`xil_imagetype_get_by_name()`	Returns a handle to an image type that has the name you specify

Synchronization Flag

XIL operations that affect images may be deferred in a way similar to the way in which Xlib client requests are buffered before being sent to the X server. However, if an image's synchronization flag is set, operations on that image are not deferred. You set this flag using the `xil_set_synchronize()` function. You can determine whether the flag is currently set by calling the `xil_get_synchronize()` function.

Readable and Writable Flags

Memory images can always be read and written to; however, certain device images may be read-only or write-only. To determine whether a device image is readable, use the `xil_get_readable()` function.

```
Xil_boolean xil_get_readable(XilImage device_image);
```

This function returns TRUE if the device image is readable.

To determine whether a device image is writable, use the `xil_get_writable()` function.

```
Xil_boolean xil_get_writable(XilImage device_image);
```

This function returns TRUE if the device image is writable.

Naming an Image

The XIL library enables you to assign a name (char *) to an image. This type of naming is useful because it enables you to get a handle to an image later in your program by specifying the name of the image. The functions that allow for the naming of images are listed and described in Table 3-8.

Table 3-8   Image Naming Functions

Function	Description
xil_set_name()	Sets the name of an image
xil_get_name()	Returns a copy of an image's name
xil_get_by_name()	Returns a handle to the image that has the name you specify

Temporary Images

The XIL library supports temporary images. You can create temporary images for interim images resulting from performing a sequence of XIL functions on a source image to produce a particular destination image you're interested in using.

Temporary images provide significant benefits with tiling. In addition, they help the deferred execution mechanism recognize when images are not needed. See Chapter 23, "Acceleration in XIL Programs," for details about deferred execution. It is strongly recommended that you create a temporary image for each interim image that you know you won't be accessing again.

Simple Example of Using Temporary Images

Say, for example, your application reads in an image into memory from a file. It performs a series of function calls such as an xil_lookup() followed by xil_convolve() and xil_ordered_dither()ultimately to display the image. (For now, accept that each function includes source and destination image parameters.) The output from xil_lookup() becomes the input (source image) to xil_convolve() and so forth. An xil_ordered_dither() results in an image that takes up less memory. See the example shown in Figure 3-9.

If your original image is very large, storage for the interim images can be a significant amount of memory. In Figure 3-9, the total image storage is 52 Mb.

Figure 3-9   Temporary Images

By using temporary images, only the memory for one interim image is required at a time. Once the output image from the `xil_lookup()` function is read as the input image to `xil_convolve()`, XIL invalidates (discards) the output image. In Figure 3-9, the maximum amount of memory needed to store images would be 36Mb.

Note – If your application uses tiling functions, it can significantly reduce the storage requirements. (For details on tiling, see Chapter 5, "XIL Program That Uses Tiling.")

Even when using tiling functions, the same rule applies. Once the tile is read by the next function in the series, the memory for that interim tile is discarded.

Functions For Creating Temporary Images

The XIL library provides two functions to create temporary images:
`xil_create_temporary()` and `xil_create_temporary_from_type()`.
Their function prototypes are shown here.

```
XilImage
xil_create_temporary(XilSystemState system_state,
                     unsigned int   width,
                     unsigned int   height,
                     unsigned int   nbands,
                     XilDataType    datatype);

XilImage
xil_create_temporary_from_type(XilSystemState system_state,
                               XilImageType   imagetype);
```

These functions have the same parameters as the respective image creating
functions `xil_create()`, which is described in "Step 2: Creating an XIL
Image" on page 20, and `xil_create_from_type()`, which is described in
"Image Type" on page 59. The only difference is that the temporary functions
return a temporary rather than real images.

Temporary Image Properties

From the application point of view, a temporary image has the following
properties:

- It is a write-once, read-once image.

  Your application can write to a temporary image once and read from it once.
  Then XIL destroys the image.

- It is implicitly destroyed once read.

  If your application creates a temporary image and writes into it but then
  does *not* to read from it, your application should call `xil_destroy()` to
  clean up the image object. If, however, if the image is read from, XIL
  destroys the image.

- Only the regions of the temporary image operated upon by a function are
  populated.

Like regular images, only the region acted upon by a function is populated. If, for example, an image is 10K pixels by 10K pixels but a function operates on a 2K-pixel by 2K-pixel region of it, only that region is populated.

- Storage is transient.

  Storage is not persistent. Once one function is done reading a temporary image, that temporary image's storage is released.

- Its characteristics can be changed dynamically.

  As examples, an application can change the origin, set an ROI, set attributes, and color spaces up until the point at which the application writes into the temporary image. Once a temporary image is written into (is used as a destination), it can no longer be altered. It becomes a static image.

Temporary Image Pseudocode Example

The pseudocode below demonstrates how to use temporary images.

```
filesrc = xil_create(state, width, height, nbands, datatype);
display = xil_create_from_window(state, xdisplay, xwindow);
tmp1 = xil_create_temporary(state, width, height, nbands,
    datatype);
xil_lookup(filesrc, tmp1);
tmp2 = xil_create_temporary(state, width, height, nbands,
    datatype);
xil_convolve(tmp1, tmp2);
xil_ordered_dither (tmp2, display);
// wait //
xil_destroy(filesrc);
xil_destroy(display);
```

The following pseudocode shows how your application can change an ROI of a temporary image (`tmp1` in this case). However, an error is returned if your application tries to change `tmp1`'s ROI again after it is used as a destination in the call to `xil_lookup()`.

```
fileSrc = xil_create();
display = xil_create_from_window();
tmp1 = xil_create_temporary();
xil_set_roi(tmp1, roi);
xil_lookup(fileSrc, tmp1);
// The following would generate an error.
xil_set_roi(tmp1,NULL);
```

≡ 3

XIL Storage *4*▤

This chapter describes XIL image storage. If you plan to export an image, you need to understand XIL image storage.

This chapter has the following main sections:

Storage Object

You use an `XilStorage` object to describe an image's storage attributes, such as its scanline and pixel strides, and to store the address of the image's data. When you first create an `XilStorage` object with the `xil_storage_create()` function, the storage object contains 0 values for the image's storage attributes and a `NULL` pointer to image data. You must use XIL functions to fill in the storage object with this information. The particular attributes you need depend on the image storage format.

If you are going to access data already associated with an XIL image, you need to ask XIL to fill in the storage object attributes to learn how the data is formatted. If you are going to provide an XIL image with data that you control, you need to set the attributes of the storage object to tell XIL how to process the new data.

Storage Formats

XIL supports three image storage formats:

* `XIL_PIXEL_SEQUENTIAL`
* `XIL_BAND_SEQUENTIAL`
* `XIL_GENERAL`

These enumerators are of type `XilStorageType`. The sections that follow describe each of the storage formats and explain which attributes must be set for each format in the XilStorage object.

Note – You may choose to have XIL store portions of very large images in equal-sized buffers called *tiles.* In this case, XIL uses tiling behind the scenes to enhance image processing. Using explicit tiling functions is optional. For simplicity, and because you may never need to know about tiling to write an XIL application, storage is described here without introducing this extra layer of detail. If you want to write applications that explicitly manage tiled memory, a chapter is dedicated to this topic. See Chapter 5, "XIL Program That Uses Tiling."

`XIL_PIXEL_SEQUENTIAL`

`XIL_PIXEL_SEQUENTIAL` storage represents an image with a band stride of 1 and an arbitrary pixel stride. Storage for all the bands is in a single memory buffer.

Data Types

The data type of an image stored in `XIL_PIXEL_SEQUENTIAL` format can be any of the following:

* `XIL_BYTE`
* `XIL_SHORT`

- `XIL_FLOAT`

A Conceptual View of `XIL_PIXEL_SEQUENTIAL` *Format*

To illustrate the `XIL_PIXEL_SEQUENTIAL` storage format and to explain storage attributes, Figure 4-1 shows a conceptual view of a very small 3x3 RGB image.

Figure 4-1   RGB Image in `XIL_PIXEL_SEQUENTIAL` Format

To the left, Figure 4-1 depicts the red (R*x*), green (G*x*), and blue (B*x*) pixel components in each of the image's bands. To the right, the figure shows the image's scanlines.

The *band stride* is the number of data elements from one band to the next. Band stride is *always 1* for `XIL_PIXEL_SEQUENTIAL` format.

The *scanline stride* is the distance in data elements to the next pixel on the next horizontal scanline. For this image, scanline stride is 9.

The *pixel stride* is the distance in data elements to the next pixel of the same band (such as the distance from R0 to R1). The pixel stride in this case is 3.

From the data pointer (depicted by the arrow), you can access each subsequent band in the buffer.

Because the image's pixel data may be intermixed with non-pixel data, such as alpha channels or padding bytes, you must explicitly use the pixel stride and scanline stride to move around the data of an image in `XIL_PIXEL_SEQUENTIAL` format.

XIL_BAND_SEQUENTIAL

`XIL_BAND_SEQUENTIAL` storage represents an image with a pixel stride of 1 and a constant band stride. Storage for all the bands of data is in a single memory buffer.

Data Types

An image stored in `XIL_BAND_SEQUENTIAL` format can be any of the following data types:

- `XIL_BIT`
- `XIL_BYTE`
- `XIL_SHORT`
- `XIL_FLOAT`

A Conceptual View of `XIL_BAND_SEQUENTIAL` Format

Figure 4-2 uses a 3x3 RGB image to illustrate `XIL_BAND_SEQUENTIAL` format. You might compare Figure 4-2 to the 3x3 RGB image shown in Figure 4-1 on page 69.

Figure 4-2   RGB Image in `XIL_BAND_SEQUENTIAL` Format

To the left, Figure 4-2 depicts the red (R*x*), green (G*x*), and blue (B*x*) pixel components in each of the image's bands. To the right, the figure shows the image's scanlines.

The pixel stride for `XIL_BAND_SEQUENTIAL` format *is always* 1.

The band stride is constant, because one band always follows the next in the same buffer. In this case, the band stride is 9.

For this image, the scanline stride is 3.

From the data pointer (depicted by the arrow), you can access each subsequent band in the buffer.

Because the image's pixel data may be intermixed with non-pixel data, such as alpha channels or padding bytes, you must explicitly use the band stride and scanline stride to move around the data of an image in `XIL_BAND_SEQUENTIAL` format.

Note – For `XIL_BIT` images, each data element is represented by a single bit in memory. In this case, you need to understand more about the image than is described in Figure 4-2. See "XIL_BIT Images" for more information.

`XIL_BIT` *Images*

The memory format for `XIL_BAND_SEQUENTIAL` images of the `XIL_BIT` data type is different than it is for those of type `XIL_BYTE`, `XIL_SHORT`, or `XIL_FLOAT`. Bit data is packed into bytes, and scanlines are padded to the nearest byte boundary if necessary. For instance, Figure 4-3 shows how a 3-band, 3-by-3 image containing 1-bit data elements might look in memory.

Note – All strides are units of bytes.

To process a particular pixel, you need to be concerned about the band it is in and its offset within that band. To set and get the offset into the byte, the XIL library provides the respective functions, `xil_storage_set_offset()` and the `xil_storage_get_offset()`. These functions are only valid for `XIL_BIT` images.

The `display` program in the `/usr/openwin/demo/xil/display` directory handles `XIL_BIT` images.

Figure 4-3　　Memory Format for a 3-Band, 3-by-3 Image Containing 1-Bit Data Elements

XIL_GENERAL

`XIL_GENERAL` storage represents an image each band of which can be stored in a different location. There is no correlation between data pointers, nor do all bands have to be of the same format. Scanline stride and pixel stride can be different for each band.

Data Types

An image stored in `XIL_GENERAL` format can be any of the following data types:

- `XIL_BIT`
- `XIL_BYTE`
- `XIL_SHORT`
- `XIL_FLOAT`

Figure 4-4   RGB Image in `XIL_GENERAL` Format

Figure 4-4 illustrates an example of the `XIL_GENERAL` format. One rule to remember is that you *cannot assume* the band stride. In Figure 4-4, a separate data pointer points to the start of each of the image's bands. Note that each band can be of a different storage format. You must process each band through its own data pointer. To move around the data, you must explicitly use the appropriate attributes for that storage format.

Note – Unless you have special requirements, you should strive to use either `XIL_PIXEL_SEQUENTIAL` for non-bit images or `XIL_BAND_SEQUENTIAL` for bit images. They have constant pixel and band strides that allow use of more efficient pixel processing code.

Tiled Storage

The XIL library supports storage of data in tiles.

Up to this point, this chapter described the buffers pointing to the image data as single buffers (except for `XIL_GENERAL` storage, which allows a separate buffer for each band of image data). If your application has turned on tiling, (see "How Do You Set Up the Tiling Mode?" on page 86 for details), XIL may store a single image's data as a set of rectangular, uniformly sized tiles each of which can reside in one or more separate memory buffers.

Say, for example, that tiling is turned on and your XIL application is processing a very large image that XIL stores in four tiles for faster processing. See Figure 4-5.

Figure 4-5   Tiled Image

The dashed lines separate regions of this image that might be stored in separate tiles in memory. Each tile would contain the image data for all the bands in that particular image region. For details on processing images a tile at a time, see Chapter 5, "XIL Program That Uses Tiling."

Note – XIL still supports the `xil_get_memory_storage()` and `xil_set_memory_storage()`functions for getting and setting an image's pixel values in memory whether or not tiling is turned on.These functions, however, are not compatible with XIL functions for tiled images. See Table 2-2 on page 26 for a list of the incompatible XIL functions. New applications are encouraged to use tiled storage--either by using convenience functions that handle tiled storage or by writing routines that process image data a tile at a time.

Storage Functions

This section briefly describes the storage functions. Several of these functions are implemented in the example programs provided with the XIL product.

Creating and Destroying Storage Objects

You use the `xil_storage_create()` function to create a storage object. When you are done with a storage object, you must free the resource by calling the `xil_storage_destroy()` function. This doesn't destroy the storage associated with the storage object, just the object describing the data.

Setting Storage Attributes

You explicitly set information about data storage to pass existing data into XIL's control. By doing so, you are telling XIL how to process the data.

Table 4-1 lists and describes the functions for setting storage attributes.

Table 4-1   Setting Storage Attributes

Function	Description
`xil_storage_set_band_stride()`	Sets the band stride in the storage object associated with the image
`xil_storage_set_data()`	Sets the pointer to the start of the image data
`xil_storage_set_data_release()`	Notifies an application that XIL is done with the data; an optional function that can be called after calling `xil_storage_set_data()`
`xil_storage_set_offset()`	Sets the offset to the start of the bit data within a byte (`XIL_BIT` data only)
`xil_storage_set_pixel_stride()`	Sets the pixel stride in the storage object associated with the image
`xil_storage_set_scanline_stride()`	Sets the scanline stride in the storage object associated with the image

4

Getting Storage Attributes

You use the following functions to read or modify data in an XIL image. These functions cause XIL to fill in the storage object with the appropriate values for the image's current format.

Table 4-2 lists and describes the functions for getting storage attributes.

Table 4-2   Getting Storage Attributes

Function	Description
`xil_storage_get_band_stride()`	Gets the band stride from the storage object associated with the image
`xil_storage_get_data()`	Returns the pointer to the start of the image data
`xil_storage_get_image()`	Gets the image associated with the storage object
`xil_storage_get_offset()`	Returns the offset to the start of the bit data within a byte (`XIL_BIT` data only)
`xil_storage_get_pixel_stride()`	Gets the pixel stride from the storage object associated with the image
`xil_storage_get_scanline_stride()`	Gets the scanline stride from the storage object associated with the image

Naming Storage Objects

As with all objects, the XIL library enables you to specify a string that serves as the name for a storage object. This naming is useful because it enables you to later get a handle to a storage object by using its name. Table 4-3 lists and describes the storage object naming functions.

Table 4-3   Naming Storage Objects

Function	Description
xil_storage_set_name()	Sets the name of a storage object
xil_storage_get_name()	Returns a copy of a storage object's name
xil_storage_get_by_name()	Returns a handle to a previously existing XIL storage object that has the name you specify

Tile-Related Storage Functions

See Chapter 5, "XIL Program That Uses Tiling," for a list of the tile-related storage functions.

Effect of Color Space on Storage Format

Besides an image's data type, its color space can play a part in how the image is stored. Consequently, you need to be aware of the ordering of bands for multiband images of different color spaces, such as RGB, YC_bC_r, CMY, and CMYK. For an RGB image, blue values are stored in the first band, green values in the second, and red values in the third. For YC_bC_r images, luminance (Y) values are stored in the first band, C_b values in the second, and C_r values in the third. For CMY and CMYK images, cyan values are stored in the first band, magenta values in the second, and yellow values in the third. For CMYK images, black values are stored in a fourth band.

Enforcing XIL Image Storage Organization

Ordinarily, getting or setting storage information while the image is exported does not guarantee that the data format or location will not change when you import the data back into XIL.

To set the storage movement flag and to get the current flag settings, you use the respective functions, `xil_set_storage_movement()`and `xil_get_storage_movement()`.

The storage movement flag is the enumeration type `XilStorageMovement` shown here.

```
enum XilStorageMovement {
    XIL_ALLOW_MOVE,
    XIL_KEEP_STATIONARY,
    XIL_REPLACE
};
```

Table 4-4 describes each of the enumeration constants.

Table 4-4   `XilStorageMovement` Enumerators

Enumerator	Description
XIL_ALLOW_MOVE	Allows XIL to move the data to a different storage device or to reformat it. On the next call to `xil_export()`, the user has no guarantee that the storage is in the same place or format and must call XIL functions to get storage information. This is the default mode; it is strongly recommended for the best performance.
XIL_KEEP_STATIONARY	Instructs XIL to leave the storage in exactly the same place and in the same format when the `xil_import()` function is called. This mode would typically be used when the caller expects to export the image again after one or a very few operations, and wants to avoid the cost of *any* data copying or reformatting that might occur.
XIL_REPLACE	Instructs XIL to return the storage to the same location and format on subsequent image exports. This allows XIL to move the storage if an accelerator is available to speed subsequent processing operations, but ensures that the caller gets the data back in the same location and format on the next export.

Caution – Use the `XIL_KEEP_STATIONARY` and `XIL_REPLACE` modes with extreme care. They may have drastic negative effects on an application's performance by limiting the possibility of hardware acceleration or causing extra data reformatting.

Setting either `XIL_KEEP_STATIONARY` or `XIL_REPLACE` on an `XilImage` frees you from having to reacquire storage layout information after every call to `xil_export()`.

4

XIL Program That Uses Tiling 5≡

This chapter takes a look at an XIL program called `example2`. The program performs the same basic tasks as the XIL program called `example1` described in Chapter 2, "Basic XIL Program." It uses two of the same source files as the `example1` program (that is, `window.c` and `example1.c`). However, `example2` uses a different `fileio.c` source file, which incorporates a routine for copying the source image data tile-by-tile into an XIL image in memory.

Because `example2` is more advanced, it requires that you understand tiling in greater detail than previous chapters present. This chapter starts by revisiting the tiling concept. Then it provides an overview of `example2`. Finally, it hones in on the code in `the fileio.c` module that shows how you can implement tiling in your own XIL application.

This chapter does not describe the entire program. Instead, it focuses on those portions of its `fileio.c` module that differ from `example1`'s `fileio.c` module. For an overview of `example1`, you should read Chapter 2, "Basic XIL Program," before reading this chapter. This chapter assumes you have read and understand all previous chapters in this guide.

Note – You are not required to use tiling in your applications. However, if you read this chapter, you will understand the advantages of tiling, which may outweigh the extra effort you put into writing this type of program. You can use coding fragments of `example2` as templates to facilitate navigating through tiled memory.

☰ *5*

This chapter has the following main sections:

Tiling

This release of the XIL library supports tiling.

What Is Tiling?

You can specify that, when an image is larger than an XIL-specified threshold, XIL should break up its storage into tiles in memory. When multiple tiles are used to represent a single image, all of the tiles must be the same size.

Note – By default, XIL stores whole images as contiguous data. This is for backward compatibility and consistent performance of existing XIL applications. You must explicitly enable tiling mode. See "How Do You Set Up the Tiling Mode?" on page 86 for details.

A tile represents all of the storage for its spatial region of the image. If an image consists of three bands, every tile represents three bands of storage. How those three bands are represented can change on a per-tile basis. XIL supports storing a tile in three formats: `XIL_PIXEL_SEQUENTIAL`, `XIL_BAND_SEQUENTIAL`, and `XIL_GENERAL`. For details on these formats, see "Storage Formats" on page 68.

The size of a tile in both the x and y direction can be arbitrary to a lower limit of 16x16. (Note that there is no limit in the y direction for tile stripping.) You can set the tile dimensions to power-of-two values, which may yield some performance advantage, but you are not required to do so. Moreover, you are not required to have the same tile size for source and destination images for a single operation. However, if the tiles are not the same size or are misaligned because of origin shifts, your application's performance may degrade somewhat.

Note – Care should be taken when changing the tile size of an image. Operations between images of different tile sizes are slower than operations between images with the same tile size. XIL chooses a default tile size for all images according to the system configuration. Imprudent tile sizes can cause significant performance penalties.

You need to distinguish between the description of tile coordinates and the description of tile storage. This distinction is very important at image boundaries. If, for example, an image's tile size is set to 1024x1024 pixels, that means a 1025x1025-pixel image has four tiles. However, since each tile has its own storage description, it is *not* necessary that all of the tiles be laid out in the same fashion. In particular, it is not necessary that 1024x1024 pixels worth of storage be allocated for each tile. Only the actual data in the tile needs to be represented. In the case of a1025x1025-pixel image, the lower right tile of the image has only one pixel of data. (See Figure 5-1.)

Figure 5-1   Storage Allocation per Tile

Therefore, if you are manipulating an exported image in tiled form, you can make calculations based on pixel coordinates as if the image consisted of fully-populated uniform-sized tiles. This allows easy conversion of image coordinates to x and y tile indices. However, as soon as you reference any actual data storage, you must determine the storage format for each tile you reference.

Why Would You Want to Understand Tiling?

Tiling an image can boost the performance of new XIL applications. It allows your application to process an image region within a single tile without bringing the entire image into memory.

How Do You Set Up the Tiling Mode?

Before XIL will use tiling, your application must explicitly call the `xil_state_set_default_tiling_mode()` function and specify the tiling mode. `xil_state_set_default_tiling_mode()` allows your application to specify one of three image storage configuration values of the enumeration type `XilTilingMode`. These are:

- `XIL_TILING`
- `XIL_STRIPPING`
- `XIL_WHOLE_IMAGE`

The default value is `XIL_WHOLE_IMAGE`, which stores an image in memory in its entirety. `XIL_STRIPPING` is single-buffer storage and is described in "What is Tile Stripping?" on page 99.

To determine the current tiling mode, your application must call the `xil_state_get_default_tiling_mode()` function.

Note – Applications work even if `XIL_TILING` is not specified. In such cases, the image is treated as a single tile. The tile's width and height are the width and height of the whole image.

How are Tiled Images Processed?

You optionally can process tiled images using tile-related functions instead of the "convenience" functions such as `xil_get_storage_with_copy()` and `xil_set_storage_with_copy()` that manage tiled memory for you. To do this, you must write a routine to manage the tiles. For a description of the XIL functions specifically related to tiling, see "Tile-Related Functions" on page 97. Using tile-related functions in your own tile managing routine avoids the costly overhead of having XIL cobble the image into a continuous memory buffer, which it does when you use the convenience functions. It also allows you to access image data by reference for immediate modification. See Chapter 6, "Handling Input and Output," for a comparison of the different methods you can use to perform I/O to and from XIL images.

Example2 Program Overview

The `example2` program contains identical functionality to `example1`. It reads an 8-bit grayscale image from a file and displays it in an X window. The program enables you to take the one's complement of the image displayed in the window, which produces a negative of the image.

Running the `Example2` *Program*

The source files for the `example2` program can be found in the `/usr/openwin/demo/xil/example2` directory. Table 5-1 lists the source files and indicates what the code in each file does.

Table 5-1   Source Files for `Example2`

Source File	Description
example2.c	Contains `main()`, which prepares different types of source images to be copied to various types of display images (is the same file as used by the `example1` program)
fileio.c	Reads an image from a file and copies the image data tile-by-tile into an XIL image in memory
window.c	Creates the program's X colormap and manages that colormap (is the same `window.c` file used by the `example1` program)

The `example2` directory also contains a `Makefile` you can use to build the program.

Just as you did with `example1`, you can run `example2` to see how it works. For instructions, see "Running the Example1 Program" on page 14. Simply change your working directory to the `/usr/openwin/demo/xil/example2` directory, and execute the program by typing:

```
% example2 brainscan.header
```

You can display the negative of the brainscan image. From the user's perspective, `example2` runs exactly the same as example1.

Program Tasks

After initializing the XIL library by calling `xil_open()`, the `example2` program performs four main tasks, namely

- It turns on `XIL_TILING`.
- It acquires an input image.
- It creates an output image.
- It displays the negative of the brain scan image when the user presses a key.

The first task listed here is not performed by the `example1` program because that program used the default tiling mode (`XIL_WHOLE_IMAGE`). The second task is implemented differently by the `example2` program and is explained in detail this chapter. The remaining tasks use the same source files (that is, `windows.c` and `example1.c`) as the `example1` program. Because there are no differences in these modules, they are not addressed in this chapter. For details on these modules, you should read the appropriate sections in Chapter 2, "Basic XIL Program."

The `example2` program code for the first two tasks is contained in the module `fileio.c`. Code fragments from this module are shown in this chapter.

Turning on Tiling

Because `XIL_TILING` is not the default tiling mode, the `example2` program explicitly calls the `xil_state_set_default_tiling_mode()` function and specifies the `XIL_TILING` option, as shown in the code fragment here. This enables the program to take advantage of tiling.

```
xil_state_set_default_tiling_mode(state,XIL_TILING);
```

Acquiring an Input Image

As you inspect `fileio.c`, you will find that the steps to acquiring the input image are the same as they are in the `example1` program. You can refer to these steps in Chapter 2 to understand how they work. For your convenience, references back to the sections with detailed descriptions are provided here.

1. **Read the image header file.**
   See "Step 1: Reading the Header File" on page 18 for detailed information.

2. **Create an XIL image.**
   See "Step 2: Creating an XIL Image" on page 20 for detailed information.

3. **Export the XIL image.**
   See "Step 3: Exporting the Image" on page 21 for detailed information.

4. **Obtain a pointer to the disk-based image data.**
   See "Step 4: Obtaining a Pointer to the Disk-Based Image Data" on page 22 for detailed information.

5. **Create an XIL storage object.**
   See "Step 5: Creating an XIL Storage Object" on page 23 for detailed information.

This brings you to the point at which you need to copy the source image on disk to the XIL image in memory.

The `example2` program uses an entirely different technique than `example1` does to perform this task. Instead of relying on a convenience function that handles tiling, `example2`'s `fileio.c` module incorporates a routine to manage tiled storage for the image. In fact, the module can manage tiled

images in any of the storage formats that XIL supports. Your understanding of the XIL storage formats is imperative to understanding how this module works. This chapter assumes you understand them.

Once the copy routine is complete, however, the `example2` program, like `example1`, calls `xil_import()` to import the image so that XIL can use acceleration to enhance performance.

Copying Source Image to an XIL Image

Image copying is implemented as a series of nested `for` loops. To support copying the image from disk to memory tile-by tile, `example2` uses some new variables and functions that you don't see in the `example1` program.

Getting The Tile Size

The first function performed on tiles in `fileio.c` is `xil_get_tilesize()`. This function gets the image's tile size and is shown here.

```
/*
 *  Get the image's tile size.
 */
xil_get_tilesize(image, &tile_xsize, &tile_ysize);
```

You use the tile size to access the image's storage on a tile basis.

Note – You must export the XIL image before getting the tile size. Exporting freezes tile size while an image is exported.

Main Looping Through the Tiles

The `example2` program uses several `for` loops to copy the image data tile-by-tile from the file to the XIL image in memory. The major double loop shown here loops through the tiles, one tile at a time, in the x and y directions.

```
...
    /*
     *  Loop through tiles in the image,copying storage from
     *  disk into each tile.
     */
    for(y = 0; y < height; y += tile_ysize) {
        for(x = 0; x < width; x += tile_xsize) {
            /*
             *  Loop counters.
             */
            int             i, j, k;
...
```

The inner `for` loop (x loop) copies data from the source file to the XIL image a tile at a time for the width of the image. The outer loop (y loop) advances to the next tile in the vertical direction.

Tiles are processed starting at the x and y coordinates (0,0) or the upper left corner of the image.

A lot of activity takes place within the x and y loops. The loop counters i, j, and k keep track of:

- The number of bands in a pixel
- The number of pixels in a scanline
- The number of scanlines in a tile

Setting Up Known Information

Within the x loop, `example2` sets up already known information about the source image on disk. The disk-based data are in pixel-sequential format. This information, combined with the x and y loop counters, is used to calculate the source scanline stride (`src_scanline_stride`) and the source pointer (`src_start`), as shown here.

```
/*
 *  Calculate the start address that corresponds to the tile
 *  area in our contiguous disk storage.
 */
unsigned int   src_scanline_stride = width*nbands;
Xil_unsigned8* src_start = ((Xil_unsigned8*)mmap_ptr) +
    y * src_scanline_stride + x * nbands;
```

Getting Storage

The first tile-related function called within the x loop is the `xil_get_tile_storage()`function shown below.

```
...
 /*
  *  Get the storage for this tile.
  */
 if(xil_get_tile_storage(image, x, y, storage) == FALSE){
...
```

This function takes the XIL image created, the x and y pixel coordinates that fall within the desired tile, and the storage object as parameters. It fills the storage object with information for the specified tile. The storage attributes (scanline stride, pixel stride, band stride, and so forth) determine the storage type.

Determining the Storage Type

The `xil_storage_is_type()` function returns type `Xil_boolean`. Given the storage object handle and the storage type as arguments, it returns `TRUE` if the given type matches that of the data format contained in the storage object.

The following code determines the XIL storage type of this particular tile of the XIL image.

```
...
    /*
     *  Determine its type (XIL_PIXEL_SEQUENTIAL,
     *  XIL_BAND_SEQUENTIAL or XIL_GENERAL) and copy the
     *  storage from disk into the tile.
     */
  if(xil_storage_is_type(storage, XIL_PIXEL_SEQUENTIAL)) {
...
  } else if(xil_storage_is_type(storage,
              XIL_BAND_SEQUENTIAL)) {
...
  } else {
    /*
     *  Setup to copy into XIL_GENERAL storage which means the
     *  data location and layout can be different in each band.
     */
...
```

With the storage type known, `example2` uses an appropriate strategy to navigate through the bands, pixels, and scanlines, copying the image data— data element by data element—from the disk-based source file to the XIL image storage.

Note that `example2` checks the storage type within the x loop (that is, for *each* tile of the image). You can't assume each tile of an image has the same storage format. Say, for example, an ROI of the image within the boundaries of a tile had been previously processed. In such a case, that tile may have a different format than the other tiles for the image: it may have been processed by a hardware accelerator, whereas the other tiles had not.

`XIL_PIXEL_SEQUENTIAL` *Case*

This chapter looks at the case in which the XIL image is in `XIL_PIXEL_SEQUENTIAL` format. The other two storage formats are left for you to examine. You can use the source code in `fileio.c` as a template for manipulating tiled storage in your XIL application.

> **Note** – The `XIL_GENERAL` loop actually works for all storage types: you do not need to write a separate loop for each. If, however, you know that the image to be processed is in a particular format, writing a separate loop for that format would enhance performance.

Obtaining Storage Object Information

As you recall, the band stride for this format is always 1. The `example2` program obtains the following information from the storage object:

- The destination image data pointer
- The scanline stride
- The pixel stride

The XIL functions for obtaining this information are shown here.

```
...
dst_start           = (Xil_unsigned8*)
                      xil_storage_get_data(storage, 0);

dst_scanline_stride = xil_storage_get_scanline_stride
                      (storage, 0);

dst_pixel_stride    = xil_storage_get_pixel_stride(storage, 0);

src_scanline = src_start;
dst_scanline = dst_start;
...
```

The `xil_storage_get_data()` function returns the data pointer in the storage object. The data pointer is the starting address of the image's storage in memory. This is where `example2` starts copying the image's pixels.

For `XIL_PIXEL_SEQUENTIAL` storage, the program calls `xil_get_scanline_stride()` and `xil_get_pixel_stride()` for band 0 only.

In the above code fragment, `example2` also sets pointers to the source and destination scanlines, which now are known information.

Navigating Through a Tile

In the next code fragment, the program uses three nested `for` loops to account for the bands, pixels, and scanlines of the image for this tile. The code below sets up pointers to source and destination bands, pixels, and scanlines.

```
...
for(i = 0; i < tile_ysize; i++) {
    src_pixel = src_scanline;
    dst_pixel = dst_scanline;

    for(j = 0; j < tile_xsize; j++) {
        src_band = src_pixel;
        dst_band = dst_pixel;

        for(k = 0; k < nbands; k++) {
            *dst_band++ = *src_band++;
        }

        src_pixel += nbands;
        dst_pixel += dst_pixel_stride;
    }

    src_scanline += src_scanline_stride;
    dst_scanline += dst_scanline_stride;
}
...
```

When `(k < nbands)` is no longer true, all bands of a pixel have been copied, and the program increments the source and destination pixel pointers. To be sure that the destination band pointer is synchronized with the destination pixel pointer, the pixel stride (`dst_pixel_stride`) is used to update the destination pixel pointer (`dst_pixel`) as shown in the line

```
dst_pixel += dst_pixel_stride;
```

Now, when the next pixel is processed, the destination band pointer is updated to the destination pixel pointer as shown in the line

```
dst_band = dst_pixel;
```

This is important when traversing the bands in case there is an alpha channel or some other data between pixels. Say, for example, the image is an RGB image. The pixel stride is not necessarily 3. It could be 4 or some other value, indicating padding of some nature between pixels. Using the pixel stride ensures that the band pointer will be pointing to the correct data element (that is, the next pixel of the image) when the next pixel is copied.

Finally, when all scanlines of the tile have been processed (`i < tile_ysize` is no longer true), execution falls out to the end of the inner x loop and the next tile of the image is processed.

Once all scanlines have been processed for the tile, the x loop increments to the next tile in the image's width and processing for that tile takes place. The procedure described in "Main Looping Through the Tiles" on page 91 repeats.

Tile Copying Summary

Figure 5-2 illustrates how an image stored in 4 tiles would be copied. The inner x loop copies pixels from left to right, then top to bottom of each tile. The outer y loop advances to the next row of tiles. The program executes until the image is completely copied from the source file on disk to the destination XIL image.

Figure 5-2   Copying Images Into Tiled Storage

Tile-Related Functions

Getting Tile-Related Data

If you are going to access data already associated with an XIL image, you need to ask XIL to fill in the storage object attributes to learn how the data is formatted. Functions for getting tile data are listed and described in Table 5-2. All of the functions are valid regardless of the tiling mode.

Table 5-2   Getting Tile-Related Information

Function	Description
`xil_get_tilesize()`	Gets the current tile size of the image's data; (gets the image's width and height in `XIL_WHOLE_IMAGE` mode)
`xil_get_tile_storage()`	Fills in the storage object with the appropriate storage attributes and data pointer for the image's storage for a given tile; (the data pointer points to the whole image in `XIL_WHOLE_IMAGE` mode)
`xil_state_get_default_tilesize()`	Gets the default tile size for all images created with a particular object of type `XilSystemState`; (the values returned are 0, 0 unless otherwise set by the user)
`xil_state_get_default_tiling_mode()`	Gets the tiling mode; (is `XIL_WHOLE_IMAGE` unless otherwise set by the user)
`xil_storage_get_coordinates()`	Gets the x and y pixel coordinates of the upper left corner of the tile within the image; values for the storage object must have been previously set by calling either of the following functions: `xil_get_tile_storage()` `xil_get_storage_with_copy()`

≡ *5*

Setting Tile-Related Data

If you are going to provide an XIL image with data that you control, you need to use and set the attributes of the storage object to tell XIL how to process the new data.

Table 5-3 lists and describes the functions for setting tile-related data. All of the functions are valid regardless of the tiling mode.

Table 5-3   Setting Tile-Related Information

Function	Description
`xil_set_tilesize()`	Sets a new tile size on an image; (the only valid values for `t_xsize` and `t_ysize` are values greater than or equal to the respective image width and height in `XIL_WHOLE_IMAGE` mode)
`xil_set_tile_storage()`	Sets one tile of the image's storage; (there is a maximum of one tile for the image in `XIL_WHOLE_IMAGE` mode)
`xil_state_set_default_tilesize()`	Sets the default tile size for all images created with a particular object of type `XilSystemState`; (the tile size can be set to any value but is only referred to when images are created in `XIL_TILING` or `XIL_STRIPPING` mode)

`xil_state_set_default_tiling_mode`	Sets the tiling mode to any of the following values:
	`XIL_TILING`
	`XIL_STRIPPING`
	`XIL_WHOLE_IMAGE`

`xil_storage_set_coordinates()`	Sets the upper left corner of the tile to position the tile within the image; must be called before calling `xil_set_tile_storage()`

> **Note** – If an image already has data associated with it, calling
> `xil_set_tilesize()` on it can seriously impede performance because of
> internal reformatting. In cases where the existing data is not needed, you
> should use a different image or destroy and recreate the image using the
> respective functions, `xil_destroy()` and `xil_create()`.

What is Tile Stripping?

Tile stripping is processing an image in horizontal strips. Each strip is
equivalent to a tile except that tile width always equals the image width. Like
tiled storage, all strips in the image must have the same width and height. To
obtain tile stripping as the image storage configuration, your application must
call the `xil_state_set_default_tiling_mode()` function and specify
`XIL_STRIPPING`. See "How Do You Set Up the Tiling Mode?" on page 86 for
more information.

Usually tile stripping is done by simply referencing a single buffer of memory
at different y locations. In addition to this technique, XIL supports storing each
strip in a separate memory buffer.

Tile stripping can be used to display an image to the user in sections.

What is the Effect of Tiling on Child Images?

Child images are independent of tiles. The following are points you should
remember about child images and tiling:

- A child image shares the storage of its parent. For the implications of shared
  storage, see "Relationship Between a Child and its Parent" on page 55.

- You cannot call `xil_set_tilesize()` on a child image, regardless of the
  tiling mode.

- Calling `xil_get_tilesize()` on a child image returns the parent's tile
  size. To determine the parent tile in which a given child pixel resides, you
  must use child offsets.

≡ 5

Handling Input and Output 6 ≡

This chapter takes a systematic look at how you read and write images in an XIL program. Basically, input comes from, and output goes to:

- A file
- A display
- A device such as a scanner, frame grabber, video card, or printer

This chapter has the following main sections:

Moving Image Data From a File to an XIL Image	*page 102*
Moving Image Data from an XIL Image to a File	*page 104*
Methods of Getting Image Data From a File into an XIL Image	*page 106*
Methods of Getting XIL Image Data Into a File	page 113
Sending Output to (and Reading Input From) the Display	*page 117*
Reading and Writing Devices Other Than Displays	*page 120*

Note – This chapter deals exclusively with reading and writing single images. XIL programs can also read and write data streams that represent compressed video or multipage documents. These data streams are stored in an XIL data structure called a compressed image sequence (CIS). Moving data into and out of these structures is discussed in Chapter 15, "Compressing and Decompressing Sequences of Images."

≡ *6*

Moving Image Data From a File to an XIL Image

The XIL library does not include any single functions that you use to read image data from a particular type of image file into an XIL image. Instead, the library provides routines that enable you to use the following general procedure to load images from files.

1. **Read the image file's header (or take whatever action is appropriate) to determine the dimensions and data type of the image.**
   Specifically, you need to know the image's width in pixels, its height in pixels, the number of bands in the image, and the type of data used to represent the value of one band of one pixel. You need this information to create an XIL image in which you can store the image's pixel values.

2. **Create an XIL image in which to store the pixel values you will read from the image file.**
   To create your XIL image, you use the `xil_create()` function, whose prototype is shown below.

   ```
   XilImage xil_create(XilSystemState state, unsigned int width,
        unsigned int height, unsigned int nbands,
        XilDataType datatype);
   ```

   The parameter `state` is the system-state data structure that was returned when you called `xil_open()` to initialize the library. The parameters `width`, `height`, and `nbands` are integers (`unsigned int`) representing the

image's width, height, and number of bands. The final parameter `datatype` is an enumeration constant of type `XilDataType`. The values of `XilDataType` are listed and described in Table 6-1.

Table 6-1 `XilDataType` Values

Value	Description
`XIL_BIT`	1-bit data
`XIL_BYTE`	8-bit unsigned data
`XIL_SHORT`	16-bit signed data
`XIL_FLOAT`	32-bit single-precision, floating point data

When the image has been created, the library returns a handle to the XIL image.

3. **Export the XIL image you just created.**
   You export the image using the `xil_export()` function, whose prototype is shown below.

```
int xil_export(XilImage image);
```

You pass this routine the handle to the image you created in the last step. The return value—`XIL_SUCCESS` or `XIL_FAILURE`—indicates whether the export operation was successful.

The reason for exporting the image is to gain access to information about how the image is stored in memory. This information is not available to your application while the image is imported.

4. **Choose the optimal method for getting the image file's data into the XIL image.**
   The XIL library offers several functions for getting data into an XIL image. Each has its own considerations. See "Methods of Getting Image Data From a File into an XIL Image" on page 106 for a description of the appropriate function you should use for each of the following situations:

- Sharing data between an image file and an XIL image

- Copying data from an image file to an XIL image

- Copying data from a large image file to an XIL image to maximize performance

- Providing data on demand from an image file to an XIL image

- Running existing applications

5. **Import your XIL image.**
   You import that image using a call to `xil_import()`, whose function prototype is shown below.

```
void xil_import(XilImage image, Xil_boolean change_flag);
```

The parameter `image` is a handle to an XIL image, and `change_flag` is a flag that indicates whether the image was modified while it was exported. If the image was modified, the flag should be set to 1; otherwise, it should be set to 0.

Caution – When you import an image, you give up control of it. Pointers to the image's data received from a previous call to `xil_export()` are invalidated unless `xil_set_storage_movement()` was called to ensure their continued validity. If you need to use the image's data after the image has been imported, you must again call `xil_export()` to export the image. Then use one of the storage access functions described above to get new pointers to the data.

Note – The `xil_export()` and `xil_import()` functions have very little overhead. They simply set a flag.

Moving Image Data from an XIL Image to a File

Just as the XIL library doesn't contain single functions that enable you to read particular types of image files, it does not include single functions that enable you to write images to files in particular formats. Instead, the XIL library provides a general mechanism for writing XIL image data to files. The procedure to follow is discussed below.

1. **Determine the dimensions and data type of your XIL image.**
   You may already know the following attributes of your XIL image: width, height, number of bands, and data type. But if you don't, you can use the `xil_get_info()` function to obtain that information. You need to know these values to determine how much image data to write to the file.

   The function prototype for `xil_get_info()` is shown below.

   ```
   void xil_get_info(XilImage image, unsigned int *width,
       unsigned int *height, unsigned int *nbands,
       XilDataType *datatype);
   ```

   The parameter `image` is a handle to an XIL image. The remaining parameters are the addresses of the variables in which you want the function to return the information you asked for.

   You may want to read other attributes of your XIL image depending on what type of information about the image you must write to your image file. See the section "Additional XIL Image Attributes" on page 47 for a list of image attributes you may be interested in.

2. **Export the image.**
   You export the image using the `xil_export()` function. Again, the reason for exporting the image is to gain access to information about how the image is stored in memory. For example, for an `XIL_BYTE` image, you need to know the address at which the image data begins, the number of bytes between pixels, and the number of bytes between scanlines.

3. **Choose the optimal method for getting the XIL image data into the image file.**
   Just as the XIL library offers several functions for getting data from a file image into an XIL image, it offers several functions for getting data from an XIL image into a file image. See "Methods of Getting XIL Image Data Into a File" on page 113 for a description of the appropriate function you should use for each of the following situations:

   - Copying data from an XIL image to a file

   - Directly accessing data used by an XIL image

   - Running existing applications

≡ *6*

Methods of Getting Image Data From a File into an XIL Image

Several functions can be used to get data into an XIL image, and each has its own considerations. This section discusses the situations in which you might write data into an XIL image and describes the appropriate XIL function for each.

Note – Before you can move image data from a file into an XIL image, you must export the image. See step 3 in the overview section, "Moving Image Data From a File to an XIL Image" on page 102.

Sharing the Data Between an Image File and an XIL Image

When an XIL image and a file image can share data by pointing to the same memory buffer, your application can optimize performance by setting the XIL image to point to the memory-mapped image file's data location by calling the `xil_set_tile_storage()` function. XIL may ultimately copy the data to a new location for acceleration but will only do so when:

- It is necessary to move the data out of the CPU's address space
- A hardware device must reformat the data

To use `xil_set_tile_storage()`:

1. Create a storage object.

```
storage = xil_storage_create(state,image);
```

2. Determine the XIL image's tile size.

```
xil_get_tile_size(image, &tx, &ty);
```

3. Set the information for the next tile's storage format and data pointer in the storage object.

```
<increment next_x, next_y>
<calculate new file_mmap_ptr corresponding to next_x, next_y>

xil_storage_set_pixel_stride(storage,0,file_pixel_stride);
xil_storage_set_scanline_stride(storage,0,file_scanline_stride;
xil_storage_set_coordinates(storage, next_x, next_y);
xil_storage_set_data(storage,0,file_mmap_ptr,NULL);
```

4. Optionally tell XIL to notify your application when it is done with the data pointed to by `data_ptr`**.**

```
xil_storage_set_data_release(storage,0,file_mmap_ptr,data_ptr);
```

Note – Your application cannot release the data in `data_ptr` until notified by the callback because XIL is sharing the data.

5. Inform the XIL image about the next tile's storage information.

```
xil_set_tile_storage(image,storage);
```

Copying the Data From an Image File to an XIL Image

XIL provides the convenience function `xil_set_storage_with_copy()` to set the data associated with an image without requiring your application to handle pixel copying. If tiling is turned on and the image is tiled, the data pointed to by storage is copied into the various storage tiles of the image automatically. Subsequent changes to the original image file's data pointer do not affect the XIL image's data.

To use `xil_set_storage_with_copy()`:

1. Create a storage object.

```
storage = xil_storage_create(state,image);
```

2. Fill the storage object with the file's information.
The following code statements illustrate which functions you would need to call to fill in the storage object if the file is in pixel-sequential format. The functions you use depend on the format of the data. For a description of the storage formats, see "Storage Formats" on page 68.

```
xil_storage_set_pixel_stride(storage, 0, file_pixel_stride);
xil_storage_set_scanline_stide(storage,0,file_scanline_stride);
xil_storage_set_data(storage,0,file_mmap_ptr,NULL);
```

3. Call the `xil_set_storage_with_copy()` **function.**

```
int xil_set_storage_with_copy(XilImage* image, XilStorage
    storage);
```

Note – To use the `xil_set_storage_with_copy()` function, the image file's data must be in a contiguous memory buffer. In addition, the convenience function can only be used to copy data for the whole image.

Copying the Data From a Large Image File to an XIL Image

To copy data from a large image file to an XIL image in order to maximize performance, XIL allows you to write a routine that provides the image file data to the XIL image on a per-tile basis by using `xil_get_tile_storage()`. When tiling is turned on, this function limits the data copy to only those portions of the XIL image that are actually needed. `xil_get_tile_storage()` is illustrated in the `example2` program in Chapter 5, "XIL Program That Uses Tiling." Note that, since XIL is merely returning a pointer to its own data, this function has very little overhead.

To use `xil_get_tile_storage()`:

1. Create a storage object.

```
storage = xil_storage_create(state,image);
```

2. Determine the XIL image's tile size.

```
xil_get_tile_size(image, &tx, &ty);
```

3. Loop to the next tile of the image; then call `xil_get_tile_storage()` **to get the storage information for that tile.**

```
<increment next_x, next_y>
xil_get_tile_storage(image,next_x,next_y,storage);
```

4. Test for the format type of the storage returned.

```
xil_storage_is_type(storage, XIL_PIXEL_SEQUENTIAL);
```

5. To determine how to access the data, retrieve information from the storage object about the tile's storage format and its data pointer.
The following code statements illustrate which functions you would need to call to retrieve tile storage information that is in pixel-sequential format. The functions you use depend on the format of the data. For a description of the storage formats, see "Storage Formats" on page 68.

```
ps = xil_storage_get_pixel_stride(storage,0);
ss = xil_storage_get_scanline_stride(storage,0);
datap = xil_storage_get_data(storage,0);
```

6. Move to the appropriate pointer in the file and copy the data from the file to the tile data pointer (`datap`**).**

7. Repeat steps 3 through 6 until you have processed as much of the file as you want to process.

Note – If the tiling mode of your application is the default `XIL_WHOLE_IMAGE`, the above code still works, although the XIL image's tile size always is the size of the image (the image consists of a single tile).

Note – If the file image data is in tiles, your application can set the XIL image's tile size to match the tile size of the file's data. The tiling mode must be `XIL_TILING` before the XIL image is created.

Providing Data On Demand From an Image File to an XIL Image

When your application can delay providing the data from the image file (such as when the image data is not changing asynchronously), XIL provides a method for the application to supply the image data on demand.

This technique provides the best performance on large images and is also extremely useful in cases where it is not possible to provide a description of the storage for the entire image in the file. For example, tiles may be located across the network or in a compressed image file whose data cannot be memory mapped.

To use an application data supply routine:

1. **Define an application data supply routine according to the prototype shown here.**

```
typedef int  (*XilDataSupplyFuncPtr)(XilImage    image,
                                      XilStorage  storage,
                                      unsigned int x,
                                      unsigned int y,
                                      unsigned int xsize,
                                      unsigned int ysize,
                                      void*       user_args);
```

This is an example.

```
struct arg_info {
   unsigned int width;
   unsigned int height;
   unsigned int nbands;
   };

int myAppSupplyRoutine(XilImage image, XilStorage storage,
   unsigned int x,unsigned int y, unsigned int xsize,
   unsigned int ysize,void* myArgs);
```

2. **Write the application data supply routine to handle each image that you intend to fill.**
    The image is included as an argument in the event that the same data supply routine is used for multiple images. You provide the data to the image through the storage object argument.

```
int
myapp_supply_routine(XilImage    image,
                     XilStorage  storage,
                     unsigned int x,
                     unsigned int y,
                     unsigned int xsize,
                     unsigned int ysize,
                     void        *myArgs)
{
    struct arg_info* argptr;
    unsigned int    width;
    unsigned int    height;
    unsigned int    bands;
    Xil_unsigned8* dataptr;

    argptr = (struct arg_info*)myArgs;
    width =  argptr->width;
    height = argptr->height;
    bands = argptr->nbands;

    /* calculate this pixel stride, etc */
    /* This code assumes PIXEL_SEQUENTIAL data */
    xil_storage_set_pixel_stride(storage,0,pixel_stride);
    xil_storage_set_scanline_stride(storage,0, scanline_stride);
    dataptr = go_mmap_data(image, x, y, xsize, ysize);
    xil_storage_set_data(storage, 0, dataptr);
    return XIL_SUCCESS;
}
```

As the example code shows, the routine must call the appropriate storage functions, such as `xil_storage_set_pixel_stride()`, `xil_storage_set_scanline_stride()`, and `xil_storage_set_data()`, to set the image data.

The `x` and `y` arguments are a convenient way to get the upper left coordinate of the data portion required.

The `xsize` and `ysize` most likely will be the tile xsize and tile ysize, but as the image may have been re-imported, you will have no way to access the values at the time of the callback.

`user_args` are available to provide any specific data that the routine may require, and will match the user arguments provided in the call to `xil_set_data_supply_routine()` for the image.

3. Set the application data supply routine on the image.

```
myarg_info.width = image_width;
myarg_info.height = image_height;
myarg_info.nbands = image_nbands;

xil_set_data_supply_routine(tile_image, myapp_supply_routine,
                            (void*)&myarg_info);
```

Note – The application data supply routine is only called the first time that an image needs data.

Note – To avoid deadlock, your application must not call any other XIL operations on the image while in the callback. The image is provided solely for identification.

Running Existing Applications

XIL continues to support `xil_get_memory_storage()`, for getting data from an XIL image, and `xil_set_memory_storage()`, for having the XIL image share data with a file. Because these functions do not support tiling and cannot be used with any of the above storage functions, they are not recommended for new applications.

Methods of Getting XIL Image Data Into a File

Several functions can be used to access data in an XIL image to be written to a file, and each has its own considerations. This section discusses the situations in which you might write an XIL image to a file and describes the appropriate XIL function for each.

Note – Before you can move image data from an XIL image to a file, you must export the image. See step 2 in the overview section, "Moving Image Data from an XIL Image to a File" on page 104.

Copying Data From an XIL Image to a File

`xil_get_storage_with_copy()` provides a convenient way of retrieving storage for the image without having to loop over tiles. Because the storage data pointer points to a copy of image's storage, no changes made to the XIL image propagate to the file image.

To use `xil_get_storage_with_copy()`:

1. **Call the** `xil_set_storage_with_copy()` **function which returns an** `XilStorage` **object.**
   You do not need to call `xil_storage_create()`.

```
storage = xil_get_storage_with_copy(image);
```

2. **Test for the format type of the storage returned.**
   This statement tests for `XIL_PIXEL_SEQUENTIAL` format.

```
if(xil_storage_is_type(storage,XIL_PIXEL_SEQUENTIAL))
...
```

3. **To determine how to access the data, retrieve information from the storage object about the storage format and its data pointer.**
   The following code statements illustrate which functions you would need to call to retrieve information that is in pixel-sequential format. The functions you use depend on the format of the data. For a description of the storage formats, see "Storage Formats" on page 68.

```
ps = xil_storage_get_pixel_stride(storage,0);
ss = xil_storage_get_scanline_stride(storage,0);
datap = xil_storage_get_data(storage,0);
```

You may then use the data pointer as the basis for the file you intend to make, or you may copy the contents into an existing buffer.

Note – If the XIL Image is very large, copying the image data will result in a performance penalty.

Directly Accessing the Data Used By the XIL Image

If you want to share the XIL image data to avoid a copy or to maintain synchronization, you should use the `xil_get_tile_storage()` function.

To use `xil_get_tile_storage()`:

1. **Create a storage object.**

```
storage = xil_storage_create(state,image);
```

2. **Determine the XIL image's tile size.**

```
xil_get_tile_size(image, &tx, &ty);
```

3. Loop to the next tile of the image, and get the storage information for that tile.

```
<increment next_x,next_y>
xil_get_tile_storage(image,next_x,next_y,storage);
```

4. Test for the format type of the storage returned.
This statement tests for XIL_PIXEL_SEQUENTIAL format.

```
xil_storage_is_type(storage, XIL_PIXEL_SEQUENTIAL);
```

5. Retrieve information about the tile's storage format and its data pointer from the storage object.
The following code statements assume pixel-sequential format. The functions you use depend on the storage format. For details, see "Storage Formats" on page 68.

```
ps = xil_storage_get_pixel_stride (storage, 0);
ss = xil_storage_get_scanline_stride(storage, 0);
datap = xil_storage_get_data(storage,0);
```

Ideally you would use this data pointer directly to represent this portion of the image file, but you could also copy the contents into an existing buffer. The data pointer and image format remain valid as long as the image remains exported.

Note – If the tiling mode of your application is the default XIL_WHOLE_IMAGE, the above code still works, although the XIL image's tile size always is the size of the image (the image consists of a single tile).

Running Existing Applications

Existing applications may continue to use xil_get_memory_storage() for accessing the data associated with an XIL image. If your application is using tiling, xil_get_memory_storage() cobbles the image data together. Because xil_get_memory_storage()cannot be used in conjunction with the other storage functions, it is not recommended for use in new applications.

Sending Output to (and Reading Input From) the Display

If you want to display an image in an XIL program, you must display that image in an X window. To perform this task, you first create an X window of the correct size. You then call the function `xil_create_from_window()` to create an XIL display image. This function turns an X window into a legitimate destination image for an XIL operation. You can then call an XIL function such as `xil_rotate()` and name the display image as the function's destination image. If you do, a rotated image is written to your display's frame buffer.

The steps listed below discuss this procedure in more detail:

1. **Use the Xlib function** `XCreateSimpleWindow()` **or** `XCreateWindow()` **to create your X window.**
   Before you create this window, you need to know—at a minimum—the width and height of the image you want to display. If you don't have this information, call the `xil_get_info()` function. It gives you information about an image's width, height, number of bands, and data type. You can then call one of the Xlib functions mentioned above to create the window.

   For information about using `XCreateSimpleWindow()` and `XCreateWindow()`, see the *Xlib Programming Manual* and the *Xlib Reference Manual*.

Note – A display image cannot be created from an X window associated with an 8-bit `StaticColor`, `TrueColor`, or `DirectColor` visual.

---

2. **Create an XIL display image.**
   As mentioned above, you create this display image by calling the `xil_create_from_window()`function, whose function prototype is shown below.

   ```
   XilImage xil_create_from_window (XilSystemState state,
        Display *display, Window window);
   ```

   The `state` parameter is the system-state data structure that was returned when you called `xil_open()` to initialize the XIL library. The `display` and `window` parameters are of data types defined in Xlib. The `display` parameter is a pointer to a structure that is returned when you initially connect to the X server. It contains information about the server and the

screens the server controls. The `window` parameter is the ID of the X window you created in the previous step. The return value of `xil_create_from_window()` is a handle to your newly created display image.

Once you have created this display image, you can use it as a destination image for XIL operations just as you would an XIL memory image. When you name the display image as your destination image, the operation's output is written directly to your display's frame buffer and displayed in the X window you created in Step 1.

3. **Call an XIL function that processes a source image and writes its output to the display image.**
   `xil_copy()` is the simplest function to use to copy a source XIL image to the display. This function takes two arguments: a handle to the source image and a handle to the display image.

Note – To see a simple example of a program that displays its output in an X window, look in `/examples/example1`. The relevant code is in the `example1.c` and `window.c` source files.

To resize a window that contains an `XilImage`, destroy the `XilImage` attached to the window, resize the window, wait for a `ConfigureNotify` event to ensure the `XResizeWindow()` is complete, then call `xil_create_from_window()` to recreate the image in the new window size.

You cannot use an X window's `backing_store` attribute to maintain an image in the window when the window is obscured or unmapped (see the *Xlib Programming Manual*). Thus, your code should always check for an `Expose` event and take the appropriate measures for displaying the image again when the window is exposed.

Possible Complications

One concern is "What if the depths of your source image and your X window are different?" What if you have a source image that has three bands and contains 8-bit data (24 bits per pixel) and an X window that is 8 bits deep? The answer to this question might be that you call an XIL function to dither the

source image from 24 bits to 8 bits before writing it to the window. But other types of mismatches can occur. Chapter 8, "Preparing Images for Display," discusses an example program that handles many of these mismatches.

Another concern is "What happens if the currently installed X colormap is not appropriate for the image you want to display?" You must solve this problem using Xlib calls. Specifically, you need to create and install an X colormap suitable for displaying your image. For complete information about how to create and install an X colormap, see the *Xlib Programming Manual* and the *Xlib Reference Manual.* For examples of functions that handle this task, look in the directory `/examples`, which contains a number of sample programs. For example, you might be interested in looking at the following routines:

- `window.c`, which is located in the `example1` subdirectory

- `window.c` (a modified version of the routine mentioned above), which is located in the `display` subdirectory

- `xilcis_color.c`, which is located in the `movie_player_example` subdirectory

Reading a Display Image

Although display images are generally used for output, you can also read a display image. Assuming that you have created such an image, you can read the data in the X window associated with it into an XIL image by taking these steps:

1. **If you don't already have an XIL image into which you can read the image data in the frame buffer, you must create one.**
   Use the `xil_create()` function to create this image. The image you create must have the proper width, height, number of bands, and data type.

2. **Perform an XIL operation using the display image as your source image.**
   For example, you could use the `xil_copy()` function to copy an image from your screen to an XIL memory image. Any pixels in the on-screen image that are obscured by another window when the read occurs are undefined.

$\equiv 6$

Reading and Writing Devices Other Than Displays

The XIL library enables you to get input from, and write output to, not only displays, but other devices such as scanners, frame grabbers, video cards, and printers. The interface to these devices is very simple. You call a single XIL function to link a device to an XIL device image, which for most practical purposes is treated like an XIL memory image. Keep in mind that when you perform an XIL operation and your source image is a device image, the operation reads its input from the associated device. Likewise, if an operation's destination image is a device image, output is written to the associated device.

Each device type has attributes associated with it. For example, the PhotoCD reader that comes with the XIL library (see Chapter 7, "Reading Kodak PhotoCD Images") has a `FILEPATH` attribute that indicates the PhotoCD image you want to read, and a `RESOLUTION` attribute that specifies a display resolution.

Depending on the device, you either need to create the device image, then set its attributes, or initialize the device attributes *before* creating the device image. The method to use depends on the device. Typically, devices don't require initialized attributes, so you create the device image, then set its attributes. However, some devices do require initialized attributes; this generally occurs when:

- The device needs one or more attributes defined at the time the device is created

- Multiple attributes are interdependent and need to be set simultaneously for the device

- Setting the attributes requires you to allocate a substantial block of memory before creating the image

The group that writes the device handler must indicate whether the device requires initialized attributes.

Initializing a Device's Attributes

As mentioned in the previous section, some device types require you to initialize one or more device attributes before creating an image for the device. To initialize a device's attributes in the XIL library, you create an XIL object of type `XilDevice`. Once this device object exists, you use it to store the device-initialization values. You then pass the device object as an argument in the

`xil_create_from_device()` function that creates the device image, as discussed in "Creating a Device Image" on page 123; all the attributes stored by the device object applies to the device image.

Note – Devices that don't require attribute initialization typically don't recognize or support device objects. For these devices, you can't use a device object to initialize device attributes, and you pass `NULL` for the `deviceObj` parameter in the `xil_create_from_device()` function.

After you're done using the device object, you need to destroy it to release the memory allocated to it. You can destroy it immediately after using it, or you can keep the object around for creating other devices of the same type, then destroy the object when you are finally done with it.

The following steps discuss the device object in more detail. These steps can be used only for devices that require or can recognize initialized device attributes.

1. Create the device object.
   To create the device object, call the function `xil_device_create()`, whose function prototype is shown below.

```
XilDevice xil_device_create(XilSystemState state,
    char *devicename);
```

The `state` parameter is the system-state data structure that was returned by `xil_open()` when you initialized the XIL library. The `devicename` parameter is the name of the device to be associated with the device image. This name must be provided by the group that writes the device handler that enables the device in the XIL library.

For example, if you plan to write an image to a video card, you might associate a device object with it as shown in the following code fragment:

```
XilSystemState state = xil_open();
XilDevice      deviceObj;

deviceObj = xil_device_create(state, "vidCard");
```

In this example, the device's name is `vidCard`. This name is provided by the group that writes the device handler. The `deviceObj` object is now associated only with a `vidCard` device type; the object cannot subsequently be associated with a different device type.

The device object's only use is to initialize device attributes when you call the `xil_create_from_device()` function that creates the device image. It cannot be used to adjust a device image's attributes after the device image is created; `xil_set_device_attribute()` does that, as discussed on page 124. However, after using the device object to create one device image, you can use the same object to store different initialization attributes, then use the modified device object when you create another device image of the same type.

2. **Set the device initialization values.**
   To store the device-initialization values in a device object, call the function `xil_device_set_value()`, whose function prototype is shown below.

```
void xil_device_set_value(XilDevice deviceObj,
    char *attribute, void *value);
```

The `deviceObj` parameter is the device object associated with the device type. The `attribute` parameter is the name of the attribute you want to set, and `value` is the attribute's value. As with device names, the attribute names and their possible values are defined by the group that writes the device handler.

Only attributes the device understands should be set on the device object. Setting attributes the device doesn't recognize generates an error. Attributes and their associated values may reference data in an application's data space; therefore, any data associated with an `XilDevice` object must remain valid while it is referenced by the device object.

To set multiple attributes, make a separate `xil_device_set_value()` function call for each; although the attributes are set individually on the device object, they are applied simultaneously to the device image when that device image is actually created (see "Creating a Device Image" on page 123). You can set as many attributes as you need to derive all required initialization attributes for the device.

For example, if you created `deviceObj` for a video card as shown in Step 1 on page 121, you might initialize its volume and speed attributes as shown in the following code fragment:

```
int vol_control = 4, speed_control = 5;

xil_device_set_value(deviceObj,
                          "VOLUME", (void*) vol_control);
xil_device_set_value(deviceObj,
                          "SPEED", (void*) speed_control);
```

3. **Create the device image.**
   Do this only after setting all needed device attributes. The next section tells you how to create the image.

4. **Destroy the device object.**
   You can do this immediately after creating the device image. However, if you plan to create other devices of the same type, you might want to keep the device object to initialize those other devices. If desired, you could set different initialization values on the object. When you're finished using it, destroy the object, see "Destroying a Device Object" on page 125.

Creating a Device Image

As explained in "Initializing a Device's Attributes" on page 120, some device types require that one or more device attributes be initialized before the device image is created; for those devices, initialize the required attributes before following the steps in this section. For devices that don't support initialized attributes, the first step for using the device is to create a device image for it. Once the device image is created, you can set its required attributes, then use the image in an XIL operation.

The following steps show you how to create a device image and set its attributes.

1. **Create a device image.**
   To create a device image, call the function `xil_create_from_device()`, whose function prototype is shown below.

   ```
   XilImage xil_create_from_device(XilSystemState state,
        char *devicename, XilDevice deviceObj);
   ```

   The `state` parameter is the system-state data structure that was returned to you when you called `xil_open()` to initialize the XIL library. The `devicename` parameter is the name of the device to be associated with the device image; the name of the device must be provided by the group that writes the device handler. If you initialized attributes for this device type, the device name should be the same name specified in the `xil_device_create()` function shown in Step 1 on page 121. The `deviceObj` parameter is the device object you created to initialize the device's attributes. If you didn't create a device object for this device type, pass NULL for this parameter.

   Note – Devices that don't require attribute initialization typically don't recognize or support device objects. For these devices, you can't use a device object to set attributes and you must pass `NULL` for the `deviceObj` argument.

2. **Set any required attributes for the device you're reading or writing.**
   For instance, you want to acquire an image from a frame grabber, and the frame grabber has a brightness control. Before you grab the image, you might want to adjust that control. The `xil_set_device_attribute()` function enables you to make this adjustment. Its prototype is shown below.

   ```
   int xil_set_device_attribute(XilImage image, char *attribute,
        void *value);
   ```

   The parameters to this function are a handle to your device image, the name of an attribute, and a value for the attribute. For example, you might set the frame grabber's brightness using the function call:

   ```
   xil_set_device_attribute(device_image, "BRIGHTNESS", (void *).5);
   ```

   Like device names, the names of device attributes and their possible values are defined by the group that writes the handler for the device.

Normally, `xil_set_device_attribute()` returns the value
`XIL_SUCCESS`. If the function is unable to set the attribute, it returns
`XIL_FAILURE`.

The XIL library also provides the `xil_get_device_attribute()`
function, which reads the value of a device attribute. Its prototype is shown
below.

```
int xil_get_device_attribute(XilImage image,
    char *attribute, void **value);
```

3. **Perform an XIL operation using your device image as either the source or
destination for the operation.**
If the operation's source image is a device image, the operation reads an
image from the associated device. For example, the operation might read a
document on a scanner or get a frame of video from a video card. If the
operation's destination image is a device image, the operation writes an
image to the associated device. For instance, the operation might print a
color image on a color printer.

Destroying a Device Object

When you create a device object, you associate it with a particular device type.
You can then use the device object to initialize as many device images of the
same device type as needed. When you are done using the device object, you
must destroy it to release the memory allocated to it.

To destroy a device object, call the function `xil_device_destroy()`, whose
function prototype is shown below.

```
void xil_device_destroy(XilDevice deviceObj);
```

The only parameter on `xil_device_destroy()` is the handle to the device
object.

≡ 6

Reading Kodak PhotoCD Images 7≡

The XIL library includes a device handler that reads and decodes images stored in the Eastman Kodak Company's PhotoCD format. This chapter explains how you use the XIL library to read such images. Before delving into that subject, however, the chapter briefly discusses PhotoCD technology in general: how images are scanned in, the PhotoYCC color space in which image data are stored, and how the data are stored on the compact disk.

This chapter has the following main sections:

The PhotoCD Technology	*page 127*
How Images Are Stored	*page 128*
Reading PhotoCD Images Using the XIL Library	*page 129*
Creating a Device Image	*page 130*
Setting Device-Image Attributes	*page 130*
Capturing an Image From a PhotoCD Disk	*page 134*
Converting the Image's Color Space	*page 135*

The PhotoCD Technology

You can take your 35 mm film or slides to a licensed photofinisher and have your pictures stored on a PhotoCD disk. Each disk can accommodate over 100 photographs. You can then view your pictures on a television set using either a PhotoCD player or a Philips CD-I system. Or you can view them on your computer using a CD-ROM player, if you have the necessary software.

≡ *7*

The PhotoCD Imaging Workstation

When you take your film or slides to the photofinisher, the photofinisher puts your pictures on a PhotoCD disk using a PhotoCD Imaging Workstation. This workstation includes a scanner, a workstation that serves as a "Data Manager," and a CD writer. The scanner scans your 35 mm film or slides and produces digital images.

The data in these images represent RGB values. The Data Manager then:

- Converts the RGB image data to the PhotoYCC color space

- Reduces the number of chrominance values in each PhotoYCC image (using the mathematical procedure *subsampling*) so that there is one pair of chrominance values for each two-by-two block of luminance values

- Produces multiple versions of each image with varying resolutions

- Uses the CD writer to write the multiple versions of each image to a compact disk

How Images Are Stored

For XIL programming, the two main things to keep in mind about PhotoCD images are:

- The images are stored in the PhotoYCC color space

- Each image is stored at multiple resolutions

The PhotoYCC Color Space

Kodak decided to convert images to a YC_bC_r color space because the data in the chrominance channels of YC_bC_r images can be subsampled without greatly affecting image quality. This ability to subsample the image data was considered important because a single scanned-in image requires 18 Mbytes of storage. After subsampling, an image requires only half that amount.

Kodak decided to develop the PhotoYCC color space instead of using an existing YC_bC_r color space (such as the one defined in CCIR Recommendation 709) because it wanted a device-independent color space, one with a broad color gamut. CCIR Rec. 709 YC_bC_r, for example, limits its color gamut to colors that can be displayed on a high-definition television. PhotoCD images may be displayed on televisions, but they may also be output to high-quality printers.

As an XIL programmer, you really only need to know that:

- When you read a PhotoCD image into an XIL image, the image data is PhotoYCC data.

- XIL removes the chrominance channel subsampling (so that the image requires more storage space).

- The XIL library supports the conversion of images to and from the PhotoYCC color space.

You must convert the data to another color space before you can display or print the image. For more information on this subject, see "Converting the Image's Color Space" on page 135.

Image Resolutions Supported in the XIL Library

As mentioned in "The PhotoCD Imaging Workstation" on page 128, PhotoCD images are stored at multiple resolutions. Table 7-1 shows the resolutions supported by the XIL library and the names Kodak has given to them.

Table 7-1   Resolutions of PhotoCD Images

Resolution	Name
192 by 128	Base/16
384 by 256	Base/4
768 by 512	Base
1536 by 1024	4Base
3072 by 2048	16Base
6144 by 4096	64Base

Reading PhotoCD Images Using the XIL Library

Chapter 3, "XIL Images," introduced the idea of a device image—an image that resides on a device such as a scanner or a printer and can be used as the source or destination image for an XIL operation. In general, support for these devices is provided by third parties. However, the XIL library does include a loadable device handler that reads PhotoCD images. This means that PhotoCD images can be treated as device images.

The section "Reading and Writing Devices Other Than Displays" on page 120 covered the general procedure for creating a device image, setting device-image attributes, and using a device image in an operation. The sections below detail the specific calls you use to read and operate on a PhotoCD image.

Creating a Device Image

To create a device image that represents a PhotoCD image, use the `xil_create_from_device()`function, whose prototype is shown below.

```
XilImage xil_create_from_device(XilSystemState state,
    char* devicename, XilDevice deviceObj);
```

The first parameter is a handle to the system state. The second is a string that identifies the PhotoCD device handler: "`SUNWPhotoCD`". And the third specifies a device object that has been associated with the device type for initializing device attributes. In this release of the XIL library, the PhotoCD reader recognizes the device object and initializes the attributes.

The code to create a device image associated with a PhotoCD image would look something like this.

```
XilImage ycc_photocd_image;
XilSystemState state;

ycc_photocd_image = xil_create_from_device(state,
"ioSUNWPhotoCD", NULL);
```

The return value of the function, `ycc_photocd_image`, is a handle to the device image.

Setting Device-Image Attributes

Before you can read a PhotoCD image, you must either

- Use a device object of type `XilDevice` (see "Initializing a Device's Attributes" on page 120 for details).

- Create the appropriate device image; then set the necessary attributes for that image.

The attributes you can set for a PhotoCD device image are shown in Table 7-2.

Table 7-2   PhotoCD Image Attributes

Attribute	Description
FILEPATH	The full pathname of the PhotoCD image you want to read.
RESOLUTION	The resolution at which you want to read the image. The possible resolutions are listed in Table 7-1 on page 129.
MAX_RESOLUTION	A read-only attribute indicating the highest resolution at which an image is stored. (In general, each image is stored at all possible resolutions, but there are exceptions to this rule.)
ROTATION	A read-only attribute indicating the amount of rotation required to display the image in its proper orientation.

A PhotoCD device image doesn't require any attribute values at creation time; therefore, it does not require a device object with it to initialize its attributes at creation time (see "Initializing a Device's Attributes" on page 120).

The FILEPATH *Attribute*

After creating a PhotoCD image, you *must* set a FILEPATH attribute for it that indicates which image you want to read. To set this attribute, call the xil_set_device_attribute() function.

```
XilImage ycc_photocd_image;
char *pathname = "...";

xil_set_device_attribute(ycc_photocd_image, "FILEPATH",
    (void *)pathname);
```

Note – The last part of the path to the image usually takes the form of
.../PHOTO_CD/IMAGES/IMG*nnnn*.PCD. This part of the pathname reflects the directory structure on the PhotoCD disk.

You can also read the value of the FILEPATH attribute if you need to determine which image you'll be reading if you read the device image. You read this value using the xil_get_device_attribute()function.

```
XilImage ycc_photocd_image;
char *pathname;

xil_get_device_attribute(ycc_photocd_image, "FILEPATH",
    (void **)pathname;
```

The RESOLUTION *Attribute*

By default, when you read a PhotoCD image, you read the Base version of that image (the one whose resolution is 768 by 512). To read a higher or lower resolution version of the image, you must set the value of the RESOLUTION attribute to one of the enumeration constants shown here.

```
typedef enum{
    XIL_PHOTOCD_16TH_BASE,
    XIL_PHOTOCD_4TH_BASE,
    XIL_PHOTOCD_BASE,
    XIL_PHOTOCD_4X_BASE,
    XIL_PHOTOCD_16X_BASE,
    XIL_PHOTOCD_64X_BASE
} XilPhotoCDResolution;
```

These constants correspond to the resolutions listed in Table 7-1 on page 129.

For example, to read the 192-by-128 version of an image, you would set the RESOLUTION attribute as shown below.

```
XilImage ycc_photocd_image;

xil_set_device_attribute(ycc_photocd_image, "RESOLUTION",
    (void *)XIL_PHOTOCD_16TH_BASE);
```

You can also read the value of the attribute using code similar to that shown below.

```
XilImage ycc_photocd_image;
XilPhotoCDResolution resolution;

xil_get_device_attribute(ycc_photocd_image, "RESOLUTION",
    (void **)&resolution);
```

The MAX_RESOLUTION *Attribute*

This is a read-only attribute that enables you to determine the highest resolution version of an image that is available for a particular PhotoCD image. In general, each image is available at all possible resolutions; however, on some pre-recorded PhotoCD disks, this may not be the case.

To read the value of this attribute, you call the xil_get_device_attribute() function and request that it store the value in a variable of the enumerated type XilPhotoCDResolution.

```
XilImage ycc_photocd_image;
XilPhotoCDResolution resolution;

xil_get_device_attribute(ycc_photocd_image, "MAX_RESOLUTION",
    (void **)&resolution);
```

The value returned in resolution will be one of the enumeration constants shown below.

```
typedef enum{
    XIL_PHOTOCD_16TH_BASE,
    XIL_PHOTOCD_4TH_BASE,
    XIL_PHOTOCD_BASE,
    XIL_PHOTOCD_4X_BASE,
    XIL_PHOTOCD_16X_BASE,
    XIL_PHOTOCD_64X_BASE
} XilPhotoCDResolution;
```

The ROTATION *Attribute*

This is a read-only attribute that enables you to determine the proper orientation for displaying a particular PhotoCD image. The rotation attribute is analogous to the *portrait* or *landscape* modes of printed text: it merely indicates the orientation of the data.

To read the value of this attribute, you call the `xil_get_device_attribute()` function and request that it store the value in a variable of the enumerated type `XilPhotoCDRotate`.

```
XilImage ycc_photocd_image;
XilPhotoCDRotate rotation;

xil_get_device_attribute(ycc_photocd_image, "ROTATION",
    (void **)&rotation);
```

The value returned in `rotation` will be one of the enumeration constants shown below.

```
typedef enum{
    XIL_PHOTOCD_CCW0,
    XIL_PHOTOCD_CCW90,
    XIL_PHOTOCD_CCW180,
    XIL_PHOTOCD_CCW270
} XilPhotoCDRotate;
```

Based upon the constant returned by `xil_get_device_attribute()`, you can perform the appropriate rotation on the image by calling `xil_transpose()` (see "Transposing Images" on page 241).

Capturing an Image From a PhotoCD Disk

Once you've created a PhotoCD device image and set the appropriate device image attributes, capturing an image from the device is a simple two-step process. You need to:

1. Create an XIL image that has the same dimensions and data type as the image you're reading.

The width and height of this image will be determined by the resolution of the PhotoCD image and also by its rotation. The number of bands in the image should be three, since an image in PhotoYCC format has three channels. And the data type of the image should be XIL_BYTE, since each Y, C_b, and C_r value in a PhotoCD image is an 8-bit value.

2. Perform an XIL operation using your device image as the source image and the XIL image mentioned above as the destination.

The code fragment below creates a destination image that will hold a Base PhotoCD image and then copies an image from the PhotoCD disk to the destination.

```
XilSystemState state;
XilImage ycc_photocd_image, dst_image;

dst_image = xil_create(state, 768, 512, 3, XIL_BYTE);
xil_copy(ycc_photocd_image, dst_image);
```

It is important to note that, after the copy operation, the data in dst_image is in the PhotoYCC color space. The next section tells you how to convert this data to another color space.

Converting the Image's Color Space

Before you can display or print an image you've read from a PhotoCD image file, you must convert the data in the image from the PhotoYCC color space to another color space. For example, if you wanted to display the image on a 24-bit frame buffer, you would convert the data to an RGB space before

displaying it. Table 7-3 lists some of the different output devices to which you might want to send the image and the color space conversion that is necessary for each case.

Table 7-3   Converting PhotoYCC Data to Another Color Space

Output Device	Color-Space Conversion
True-color display	Convert the data to an RGB color space, such as the one named `rgb709` in the XIL library.
Indexed-color display	Convert the data to an RGB color space, such as the one named `rgb709` in the XIL library. Then, dither the RGB image to an 8-bit image. For information about XIL dithering operations, see "Dithering an Image" on page 161.
Grayscale display	Convert the data to a luminance-only color space, such as the XIL color space `y709` or `ylinear`.
Monochrome display	Convert the data to a luminance-only color space, such as the XIL color space `y709` or `ylinear`. Then, dither the 8-bit luminance data to a 1-bit image. For information about XIL dithering operations, see "Dithering an Image" on page 161.
Color printer	Convert the data to a CMYK color space. The XIL library supports a linear CMYK space named `cmyk`.

For detailed information about the color spaces that the library supports and about converting image data from one color space to another, see "Color Space Manipulation" on page 178.

The code fragment on this and the following pages shows how the ROTATION attribute from a PhotoCD image is read and used, and how the image's data is converted to the RGB color space specified in CCIR Recommendation 709.

Code Example 7-1   Using the ROTATION attribute

```
XilSystemState    state;
XilImage          ycc_photocd_image;
XilImage          rgb_photocd_image;
XilImage          rotated_photocd_image;
XilImage          display;
XilPhotoCDRotate  rotation;
unsigned int      width;
```

Code Example 7-1 Using the ROTATION attribute

```
unsigned int     height;
unsigned int     nbands;
unsigned int     datatype;
char*            pathname = "my_photocd_image";

/*
 *  Open the XIL Library
 */
state = xil_open();

if (state == NULL) {
    fprintf(stderr, "Failed to open XIL library.\n");
    return 1;
}

/*
 *  Create the PhotoCD device image.
 */
ycc_photocd_image =
    xil_create_from_device(state, "SUNWPhotoCD", NULL);

if (ycc_photocd_image == NULL) {
    fprintf(stderr, "Failed to construct SUNWPhotoCD device image.\n");
    return 1;
}

/*
 *  Set the file name.  The default resolution is XIL_PHOTOCD_BASE.
 */
xil_set_device_attribute(ycc_photocd_image, "FILEPATH", pathname);

/*
 *  Get the rotation attribute and image's width and height.
 */
xil_get_device_attribute(ycc_photocd_image,
                         "ROTATION", (void**)&rotation);

xil_get_info(ycc_photocd_image, &width, &height, &nbands, &datatype);

/*
 *  Transpose (rotate) the image based on the rotation angle.
 *  Depending upon the rotation angle, construct an image to store
 *  the transpose results.
```

Code Example 7-1   Using the ROTATION attribute

```
 */
switch (rotation) {
  case XIL_PHOTOCD_CCW0:
    rotated_photocd_image = ycc_photocd_image;
    break;

  case XIL_PHOTOCD_CCW90:
    /*
     *  Flip the image's width and the height.
     */
    rotated_photocd_image =
        xil_create(state, height, width, nbands, datatype);
    xil_transpose(ycc_photocd_image,
                  rotated_photocd_image, XIL_FLIP_90);
    xil_get_info(rotated_photocd_image, &width, &height, NULL, NULL);
    break;

  case XIL_PHOTOCD_CCW180:
    rotated_photocd_image =
        xil_create(state, width, height, nbands, datatype);
    xil_transpose(ycc_photocd_image,
                  rotated_photocd_image, XIL_FLIP_180);
    break;

  case XIL_PHOTOCD_CCW270:
    /*
     *  Flip the image's width and the height.
     */
    rotated_photocd_image =
        xil_create(state, height, width, nbands, datatype);
    xil_transpose(ycc_photocd_image,
                  rotated_photocd_image, XIL_FLIP_270);
    xil_get_info(rotated_photocd_image, &width, &height, NULL, NULL);
    break;
}

/*
 *  Perform a color space conversion to rgb709.
 */
rgb_photocd_image =
    xil_create(state, width, height, nbands, datatype);

/*
```

Code Example 7-1   Using the ROTATION attribute

```
 *  Set color spaces for color space conversion
 */
xil_set_colorspace(rotated_photocd_image,
    xil_colorspace_get_by_name(state, "photoycc"));
xil_set_colorspace(rgb_photocd_image,
    xil_colorspace_get_by_name(state, "rgb709"));

/*
 *  Convert the image's color space so it can be displayed
 */
xil_color_convert(rotated_photocd_image, rgb_photocd_image);

/*
 *  ...code to open an X window of correct depth...
 */
display = xil_create_from_window(state, xdisplay, xwindow);

if (display == NULL) {
    fprintf(stderr, "Failed to construct display device\n");
    return 1;
}

/*
 *  Copy the RGB image to the display and continue to
 *  redisplay on Expose events.
 */
xil_copy(rgb_photocd_image, display);

while (1) {
    XNextEvent(xdisplay, &event);
    if (event.xany.type == Expose) {
        xil_copy(rgb_photocd_image, display);
    } else if (event.xany.type == ButtonPress)
        break;
    }
}

/*
 *  Destroy images.
 */
xil_destroy(display);
xil_destroy(rgb_photocd_image);
```

Code Example 7-1   Using the `ROTATION` attribute

```
if(rotated_photocd_image != ycc_photocd_image) {
    xil_destroy(rotated_photocd_image);
}

xil_destroy(ycc_photocd_image);
```

Preparing Images for Display 8≣

To introduce you to programming with the XIL library, Chapter 2, "Basic XIL Program," presented an example program that read in an 8-bit grayscale image and displayed it in an X window created from one of four X visuals. This chapter looks at a revision of that example called `display`. The new example can display images of several data types and images of more than one band and was designed to illustrate how to handle some of the many display cases you may encounter.

The first part of this chapter explains briefly how to build and run the `display` example. The bulk of the chapter then discusses the most important tasks performed in the example.

This chapter has the following main sections:

Converting a Single-Band Image to a Multiband Image	*page 142*
Converting an XIL_SHORT Image to an XIL_BYTE Image	*page 145*
Converting an RGB Image to an Indexed-Color Image and a Colormap	*page 147*
Converting a 24-Bit Image to a 1-Bit Image	*page 148*
Converting an 8-Bit Image to a 1-Bit Image	*page 150*
Displaying a 1-Bit Image on a Monochrome Display	*page 150*

The chapter concludes with a list of all the cases handled in the program.

Running the `Display` *Program*

The source files for the `display` program can be found in the `/usr/openwin/demo/xil/display` directory. Table 8-1 lists these source files and indicates what the code in each file does.

Table 8-1   Source Files for `display`

Source File	Description
`display.c`	Contains `main()`, which prepares different types of source images to be copied to various types of display images
`fileio.c`	Reads an image from a file and loads the image data into an XIL image (is the same `fileio.c` that is used by the `example1` program)
`window.c`	Creates the program's X colormap and manages that colormap

To run this example:

1. **Change your working directory to** `/usr/openwin/demo/xil/display`**.**

2. **Build the program using the** `Makefile` **in that directory.**

3. **Execute the program from the directory in which you built it, using the command line:**
   % **`display toys.header`**

The program displays an RGB image of a group of toys in an X window. To stop the program, move your pointer into the program's window, and click any mouse button.

Converting a Single-Band Image to a Multiband Image

The `display` program approaches this problem in two ways: by passing the source image through a lookup table and by replicating the source image in the multiple bands of the destination image.

Passing the Source Image Through a Lookup Table

If there are only a few different values in the source image—for example, when the source image is a 1-bit image—passing the source image through a lookup table is an effective method of performing this conversion. For instance, if all

the values in the source are 0 or 1, you can use a lookup table with only two entries: one that specifies the values to appear in each band of the destination for pixels that have a value of 0 in the source, and another that specifies the values to appear in each band of the destination for pixels that have a value of 1.

For example, the program uses this method to prepare a bit image for display in a 24-bit window. To make the source image compatible with the display image, the program converts the source image to a 3-band, 8-bit image. To do this, the example:

1. Creates a temporary image, `retained_image`, that has the same width and height as the source image, but is 3 bands deep and contains `XIL_BYTE` data. (This temporary image is copied later to the display.)

2. Creates the lookup table (an object of type `XilLookup`) shown in Figure 8-1.

Lookup Index (XIL_BIT) Values to be written to temporary image (XIL_BYTE)

| 0 | 0 | 0 | 0 |
| 1 | 255 | 255 | 255 |

Figure 8-1   Lookup Table

3. For each pixel in the source image, looks up the proper value in the left column of the table and writes the three values to its right to the first, second, and third bands of the corresponding pixel in `retained_image`.

The code that implements these steps is shown below.

```
...
#define BITSIZE 2
...
XilImage image            = NULL;
XilImage retained_image   = NULL;
...
XilLookup lookup          = NULL;
Xil_unsigned8 lookupdata[] = {0, 0, 0, 255, 255, 255};
...
retained_image = xil_create(state, width, height, 3, XIL_BYTE);
lookup = xil_lookup_create(state, XIL_BIT, XIL_BYTE, 3,
    BITSIZE, 0, lookupdata);
xil_lookup(image, retained_image, lookup);
```

Replicating the Source Image in the Bands of the Destination

The second method the example uses to convert a single-band source image to a multiband destination image is to replicate the source image in each band of the destination. To perform this task using the XIL library, you must first create a child image representing each band of the destination and then copy the source to each child image. Because any changes made to an XIL child image affect the parent, these copies result in the source being copied to each band of the destination.

The display program converts a 1-band `XIL_BYTE` source image to a 3-band `XIL_BYTE` image using the following code.

```
...
XilImage image            = NULL;
XilImage retained_image   = NULL;
XilImage band0            = NULL;
XilImage band1            = NULL;
XilImage band2            = NULL;
...
/*
 * Copy the grayscale image into each of the bands
/*
retained_image = xil_create(state, width, height, 3, XIL_BYTE);
band0 = xil_create_child(retained_image, 0, 0, width, height, 0,
     1);
band1 = xil_create_child(retained_image, 0, 0, width, height, 1,
     1);
band2 = xil_create_child(retained_image, 0, 0, width, height, 2,
     1);
xil_copy(image, band0);
xil_copy(image, band1);
xil_copy(image, band2);
...
```

Converting an `XIL_SHORT` *Image to an* `XIL_BYTE` *Image*

Converting an `XIL_SHORT` image to an `XIL_BYTE` image is a two-step process. First, you must rescale the values in the source image so that they fall in the range 0 to 255. Then, you must cast the values in the source image to values of type `XIL_BYTE`.

In the rescaling step, you could simply map the lowest possible value in the source (-32768) to the lowest possible value in the destination (0) and the highest possible value in the source (32767) to the highest possible value in the destination (255). However, you'll generally achieve better results by first finding the extreme values in the source and then mapping the lowest value actually in the source to 0 and the highest value to 255.

The code that `display` uses to perform this task (in one instance) is as follows:

```
...
#define CMAPSIZE 256
...
    /*
     *  Rescale Parameters
     */
    float           mult;
    float           offset;
...
    /*
     *  Range of data
     */
    float           low;
    float           high;
...
    /*
     *  Remap the range of image data to 0 to 255
     */
    xil_extrema(image, &high, &low);
    mult = (CMAPSIZE - 1) / (high - low);
    offset = -((low * (CMAPSIZE - 1)) / (high - low));
    xil_rescale(image, image, &mult, &offset);

    /*
     *  Cast the remapped image into each band of a 3 banded
     *  byte image
     */
    xil_cast(image, band0);
    xil_cast(image, band1);
    xil_cast(image, band2);
...
```

The call to `xil_extrema()` stores the source image's highest value in `high` and its lowest value in `low`. Then, the call to `xil_rescale()` maps the lowest value in the image to 0 and the highest to 255. Once the values in `image` have been brought into the proper range, the example casts them to values of type `XIL_BYTE`. When casting a 16-bit value to an 8-bit value, `xil_cast()` preserves the 8 least significant bits of each input value, which is what the program requires.

Converting an RGB Image to an Indexed-Color Image and a Colormap

The `display` program prepares a true-color image to be shown on an indexed-color display by performing an ordered dither on the source image. The general procedure the example employs is to create a colorcube and a dither mask and then to call the `xil_ordered_dither()` function. After this operation, the program has a single-band version of the source image and a colorcube that defines the RGB values to be associated with each value in the indexed-color image. Before displaying the indexed-color image, the example creates a virtual X colormap and writes the values in the XIL colorcube to the X colormap.

Note – For a more detailed explanation of ordered dithering, see "Dithering an Image" on page 161.

The code the example uses to perform the ordered dither is shown below.

```
...
XilImage image              = NULL;
XilImage display_image      = NULL;
XilImage retained_image     = NULL;
...
XilLookup colorcube         = NULL;
...
XilDitherMask dithermask;
...
retained_image = xil_create(state, width, height, 1, XIL_BYTE);
colorcube = xil_lookup_get_by_name(state, "cc496");
dithermask = xil_dithermask_get_by_name(state, "dm443");
...
set_colormap(display_image, xdisplay, xwindow, colorcube);
xil_ordered_dither(image, retained_image, colorcube,
    dithermask);
...
```

The `cc496` colorcube and the `dm443` dither mask are default objects the library makes available when it is initialized. The colorcube is appropriate for dithering an RGB image to 216 colors (the indexes range from 38 to 253), and `dm443` is a 4-by-4 dither mask.

≡ *8*

Before the indexed-color image can be displayed, the color values in the colorcube `colorcube` must be stored in color cells 38 to 253 in the application's X colormap. This task is handled by the `set_colormap()` function. Note that the colormap data that is being loaded into the X colormap is taken from the lookup table that was used as the colorcube for the dithering operation.

The XIL library also includes two other types of dithering operations that you can use in preparing a true-color image to be shown on an indexed-color display. The functions that perform these dithering operations are called `xil_nearest_color()` and `xil_error_diffusion()`. For further information about these functions, see "xil_nearest_color()" on page 164 and "xil_error_diffusion()" on page 170.

Converting a 24-Bit Image to a 1-Bit Image

The example uses a two-step process to convert an RGB image to a 1-bit image. The first step is to convert the 24-bit source image to an 8-bit grayscale image by extracting luminance information from the source image. Using the XIL library, you perform this task by converting the source image from the RGB color space to the Y color space. The destination image for this color-space conversion must be a single-band image.

The code `display` used to perform this color-space conversion is as follows:

```
XilImage image              = NULL;
XilImage bit_image          = NULL;
...
XilColorspace rgb_cspace;
XilColorspace ylinear_cspace;
...
bit_image = xil_create(state, width, height, 1, XIL_BYTE);
...
rgb_cspace = xil_colorspace_get_by_name(state,"rgblinear");
xil_set_colorspace(image, rgb_cspace);
ylinear_cspace = xil_colorspace_get_by_name(state, "ylinear");
xil_set_colorspace(bit_image, ylinear_cspace);
xil_color_convert(image, bit_image);
...
```

The second step in the process is to dither the 8-bit image to a 1-bit image using either the `xil_ordered_dither()` or `xil_error_diffusion()` function. If execution speed is your primary concern, use the ordered-dither function. To obtain the best quality image, use the error-diffusion function. For more information about these functions, see "xil_ordered_dither()" on page 174 and "xil_error_diffusion()" on page 170.

The `display` example dithers the image using `xil_error_diffusion()`.

```
...
XilKernel distribution;
...
XilLookup      colorcube   = NULL;
int            multipliers = -1;
unsigned int   dimensions  = 2;
...
retained_image = xil_create(state, width, height, 1, XIL_BIT);
distribution = xil_kernel_get_by_name(state, "floyd-steinberg");
colorcube = xil_colorcube_create(state, XIL_BIT, XIL_BYTE, 1, 0,
    multipliers, dimensions);
xil_error_diffusion(bit_image, retained_image, colorcube,
    distribution);
...
```

Converting an 8-Bit Image to a 1-Bit Image

This task is the same as the second step described in the preceding section. You should dither the 8-bit image to a 1-bit image using either the `xil_ordered_dither()` or the `xil_error_diffusion()` function. As noted above, the first of these functions is the fastest, but the second generally produces better looking results.

Displaying a 1-Bit Image on a Monochrome Display

The `display` program is set up to deal with `XIL_BIT` images in which 0's represent black pixels and 1's represent white pixels (the normal case). In the X colormap for monochrome displays supported by the XIL library, however, 0 represents white and a 1 represents black; therefore, the `display` program takes the one's complement of the source image (using `xil_not()`) before displaying it.

Types of Images Displayed

This section simply lists the cases handled by the `display` example so that you can quickly determine whether a case you're interested in is covered. In Table 8-2, the types of images the program can process are shown in the left column, and the supported X visuals are listed in the top row. A check mark indicates that the program can display a particular type of image in an X window created using a particular X visual.

Table 8-2   Cases Handled by the `display` Program

	TrueColor 24 Bits	PseudoColor 8 Bits	GrayScale 8 Bits	StaticGray 8 Bits	StaticGray 1 Bit
XIL_BIT 1 Band	✔		✔	✔	✔
XIL_BYTE 1 Band	✔			✔	✔
XIL_BYTE 3 Bands	✔	✔	✔		✔
XIL_SHORT 1 Band	✔		✔	✔	✔

8≡

> **Note** – For an explanation of X visuals and their uses, see the *Xlib Programming Manual.*

≡ *8*

Presentation Functions 9≡

This chapter covers a group of functions you are likely to use when displaying or printing images.

This chapter has the following main sections:

Copying an Image to the Display	*page 153*
Rescaling an Image	*page 158*
Casting an Image From One Data Type to Another	*page 160*
Dithering an Image	*page 161*
Color Space Manipulation	*page 178*
Black Generation (preparing CMY images for printing)	*page 186*

Copying an Image to the Display

The `xil_copy()` function lets you copy a source image to a destination image. `xil_copy()` copies each *plane* (bit) in a source-image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel. To control the pixel planes copied from the source to the destination, you can define a *plane mask* to specify the bit planes to copy, then call `xil_copy_with_planemask()` to perform the copy.

The `xil_copy()` and `xil_copy_with_planemask()` functions are discussed separately in the following sections.

$\equiv$ *9*

Copying All Bit Planes

The simplest way to display an image is to use `xil_copy()` to copy your source image to the display. The `xil_copy()` function copies each *plane* (bit) in a source image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel. For example, an eight-bit-deep frame buffer has eight bit-planes; if you use `xil_copy()` to copy an `XIL_BYTE` image to that frame buffer, all eight planes of each source pixel are copied to all eight planes in the frame buffer's destination pixel.

For `xil_copy()` to work, you must have created a display image that serves as the destination for the copy. (For information about what display images are and how you create them, see "Display Images" on page 45.) In addition, the source image—the image to be displayed—and the display image must use the same number of bits to represent a pixel. Table 9-1 lists the combinations of source and display images that match in this way.

Table 9-1   Matching Source and Display Images

Source Image Type	Required Display Image Depth
Single-band `XIL_BIT` image	One bit
Single-band `XIL_BYTE` image	Eight bits
Three-band `XIL_BYTE` image	Twenty-four bits

If your source and destination images match, you can display the source image using code similar to this.

```
XilImage src, display_image;

xil_copy(src, display_image);
```

Besides performing this kind of explicit copy using `xil_copy()`, you can also request that an image processing function (like a rotate function) copy its output to the display. You do this by specifying a display image as the

destination for the operation. For example, the following call to `xil_rotate()` rotates the source image and writes the rotated image to the display.

```
XilImage src, display_image;

xil_rotate(src, display_image, "bilinear", -0.7854);
```

Note – Before this type of operation will work, the source and display images must use the same number of bits to represent a pixel. If your source and display images do not match in this respect, you must perform one or more of the tasks discussed in this chapter before displaying your image.

If overlapping but not coincident sibling images (children of the same parent) are specified as the source and destination, `xil_copy()` detects the overlap and correctly generates the destination image. All other operations generate a warning message under these conditions and have undefined results, as mentioned in the discussion on parent and child images on page 58.

Copying Only the Planes Defined in a Plane Mask

The `xil_copy()` function copies each plane (bit) in a source-image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel. To control the pixel planes copied from the source to the destination, you can define a plane mask to specify the bit planes to copy. Then call the `xil_copy_with_planemask()` function to perform the copy. The prototype for `xil_copy_with_planemask()` is shown here.

```
void xil_copy_with_planemask(XilImage src, XilImage dst,
    unsigned int planemask[]);
```

A typical reason to copy with plane mask control is to overlay one image over another. Because you can copy specific planes from each source image pixel, the underlying image isn't visible where the overlay is drawn; however, all planes from the underlying image are still available and can be refreshed in the

destination by rendering it with another plane mask. Overlays can improve performance by reducing the amount of graphic information that has to be redrawn. They can also be used to highlight graphics for selection.

Note – Because read/write color cells can be allocated only in PseudoColor and DirectColor visuals, you need to provide an alternative technique to overlays for other visuals. You may also want to provide an alternative in case the overlay fails. For more information on overlays and their advantages and disadvantages, see the *Xlib Programming Manual.*

When you use `xil_copy_with_planemask()`, each pixel in the destination image is defined by the following operation:

```
dst = (dst & ~mask) | (src & mask)
```

Here, `dst` is the destination image, `mask` is the plane mask, and `src` is the source image. Thus, if the plane-mask bit is "on," the copy overwrites the corresponding bit in the destination image; otherwise, the bit in the destination image is unchanged.

As an example, assume you have two single-band `XIL_BYTE` images you want to overlay. Table 9-2 shows the plane masks you might use.

Table 9-2   Plane Masks for an Overlay

Plane Mask	Binary Base	Hexadecimal Base	Decimal Base
Mask 1	00000001	0x1	1
Mask 2	11111110	0xfe	254

In Table 9-2, Mask 1 has only its low order bit turned on; thus, it ensures that only the source image's low order bit is copied to the destination. Mask 2, on the other hand, has its seven high-order bits turned on, so it ensures that the source image's seven high-order bits are copied to the destination.

The following code shows how you might define and use the plane masks shown in Table 9-2.

```
XilImage src1, src2, dst;
unsigned int planemask1[1], planemask2[1];

planemask1[0] = 0x1;
planemask2[0] = 0xfe;

xil_copy_with_planemask(src1, dst, planemask1);
xil_copy_with_planemask(src2, dst, planemask2);
```

The plane mask must be an array of unsigned integers. The number of array elements must match the number of image bands; each array element specifies the plane mask for the corresponding band in the destination image. Both the source and destination images must have the same type and number of bands, and in-place operations are supported.

When using a plane mask for copying an image to the display, the depth of the window is the upper limit on the number of meaningful bits you can set in the plane mask, and you must manipulate the colormap to get a reasonable display. For more information on allocating color cells for overlays, see the *Xlib Programming Manual.*

In addition to overlays, you can use a plane mask for double buffering on hardware that doesn't have separate memory buffers. For details on XIL's double buffering support, see "Double Buffering," below.

Double Buffering

On hardware that has separate memory buffers, the XIL library directly supports double buffering of device images.

The `xil_create_double_buffered_window()` function creates an image associated with a specified X window. This function works similarly to `xil_create_from_window()` except it attempts to establish hardware double buffering.

The *active* buffer represents the buffer that will be affected when an operation uses a double-buffered image. At creation of a double-buffered image, the active buffer is the back buffer.

`xil_get_active_buffer()` returns the current `XilBufferId` for the active buffer. `XilBufferId` is an enumeration type that can be one of the following enumeration constants:

- `XIL_FRONT_BUFFER`
- `XIL_BACK_BUFFER`

`xil_set_active_buffer()` sets the active buffer to either `XIL_FRONT_BUFFER` or `XIL_BACK_BUFFER`. Changing the active buffer to `XIL_FRONT_BUFFER` does not affect the buffer contents.

The `xil_swap_buffers()` function swaps the the contents of the back buffer to the front buffer. The contents of the back buffer is undefined after the swap.

Double buffering is useful for animation because you can render an image in a hidden memory buffer while you display another image in a second buffer. Once rendering is complete in the hidden buffer, you can display its contents, then hide the second buffer and render a new image into it. Rendering into the hidden buffer and quickly switching between buffers smooths the transition between images.

As with overlays, double buffering requires you to toggle between two colormaps as you change plane masks between images. For more information on using double buffering for animation, consult Foley, et al. *Computer Graphics: Principles and Practice* (see Appendix E, "Bibliography").

Rescaling an Image

The `xil_rescale()` function maps the values in each band of an image from one range (for example, 0 to 2047) to another range (for example, 0 to 255). The function performs this mapping by multiplying each value in a band by one constant and then adding a constant to the result of the multiplication.

The prototype for `xil_rescale()` is shown below.

```
void xil_rescale(XilImage src, XilImage dst, float *scale,
    float *offset);
```

The `src` and `dst` parameters are handles to the source and destination images. The `scale` parameter is an array of floating-point numbers that serve as the constants by which values in the bands of the source image are multiplied. The

values in band 0 are multiplied by `scale[0]`, the values in band 1 are multiplied by `scale[1]`, and so on. Therefore, the number of elements in the array must match the number of bands in the image. The final parameter, `offset`, is an array of floating-point numbers, one of which is added to the scaled values in each band of the source image.

There are many cases in which this function is useful as you prepare an image for display. Two are mentioned below.

One obvious case is that in which you want to display an image containing `XIL_SHORT` values. Say, for example, you want to display a single-band `XIL_SHORT` image in an X window created using an 8-bit `GrayScale` visual. The values in your image may range from -32768 to 32767, but the values you write to the display image must fall in the range 0 to 255 (or some subset of that range). To handle this situation, you could perform these steps:

1. **Use the** `xil_extrema()` **function to determine the minimum and maximum values in your source image.**
   For a description of `xil_extrema()`, see the section entitled "Converting an XIL_SHORT Image to an XIL_BYTE Image" on page 145. Your image looks better when displayed if you map the minimum value to 0 and the maximum value to 255, as opposed to mapping -32768 to 0 and 32767 to 255. In the latter case, you might wind up with very few gray levels in the image to be displayed.

2. **Rescale the values in your** `XIL_SHORT` **image so that they fall in the range 0 to 255.**
   You might use the following code to accomplish this step.

```
XilImage src, dst;
float maximum[1], minimum[1];
float multiplier[1], offset[1];

multiplier[0] = 255 / (maximum[0] - minimum[0]);
offset[0] = -((minimum * 255) / (maximum[0] - minimum[0]));
xil_rescale(src, dst, multiplier, offset);
```

3. Cast the values in the rescaled image to be of type `XIL_BYTE`**.**
Although you have rescaled your image so that its values fall in the range 0
to 255, those values are still 16-bit signed values. Before copying your image
to the display, you must use the `xil_cast()` function to create an
`XIL_BYTE` version of the image. For more information about `xil_cast()`,
see "Casting an Image From One Data Type to Another," below.

Here's another case in which you may need to rescale an image before
displaying it. Suppose you want to display an 8-bit grayscale image in an 8-bit
`GrayScale` window, but you do not want to use the currently installed X
colormap in displaying it. You want to create a new virtual X colormap, store a
grayscale ramp in that colormap, and have it installed when your application
is active. If you write values to all 256 color cells in the virtual colormap,
you're almost certainly going to see colormap flashing when the colormap is
installed, so you may decide not to write values to the first 16 color cells—to
write all the values needed to display your image in color cells 17 through 255.
This strategy requires that you rescale the values in your image so that they fall
in the range 17 to 255.

Casting an Image From One Data Type to Another

The `xil_cast()` function casts an XIL image from one data type to another.
The function's prototype is shown below.

```
void xil_cast(XilImage src, XilImage dst);
```

The `src` parameter is a handle to your source image, and `dst` is a handle to a
destination image. This destination must have the same width, height, and
number of bands as the source image and must have the data type to which
you want to cast the source image.

There are a number of instances in which you may need to use this function as
you prepare an image for display. For example, you may have a single-band
`XIL_SHORT` image that you want to display in an 8-bit window. To do this you
need to follow these steps.

1. **Rescale the image.**
   Unless the values in your `XIL_SHORT` image already fall in the range 0 to
   255, you must use the `xil_rescale()` function to map them to that range,
   or a subset of that range. For more information about rescaling images, see
   "Rescaling an Image" on page 158.

2. **Cast the `XIL_SHORT` image to an `XIL_BYTE` image.**
   Before you can display your image in an 8-bit window, you must cast the
   16-bit values in the source image to 8-bit values, as shown in this code
   fragment.

```
XilImage short_image, byte_image;
unsigned int width, height, nbands; /* Dimensions of images */

byte_image = xil_create(state, width, height, nbands,
    XIL_BYTE);
xil_cast(short_image, byte_image);
```

Similarly, if you have an `XIL_BIT` image that you want to display in an 8-bit
window, you must cast the source image to an `XIL_BYTE` image before
displaying it. In this cast, the values 0 and 1 in the `XIL_BIT` image are cast to
indices 0 and 1 in the `XIL_BYTE` image. If you want different indices, convert
the image by passing it through a lookup table rather than by casting its data
type. For information about lookup tables, see "Passing an Image Through a
Lookup Table" on page 271.

Note – When casting the image so you can display it, you may want to use a
display image as the destination image for the cast. However, if you intend to
use the converted image again, you can cast the source image to an interim
destination image, then use `xil_copy()` to copy the interim image to the
display.

Dithering an Image

The XIL library provides several functions you can use to prepare images for
display by dithering them. Before looking at these functions, though, this
section explains what it means to dither an image in an XIL application.

≡ *9*

What Is Dithering?

First, you need to know that the XIL library gives you the ability to pass a single-band image of any data type through a lookup table to produce a single-band or multiband image of the same or another data type. The lookup table used for this operation is a data structure of type `XilLookup` and has (among others) the following attributes:

- An input data type
- An output data type
- A number of bands on the output side

Figure 9-1 shows a single-band `XIL_BIT` image being passed through a lookup table to produce a three-band `XIL_BYTE` image.

Figure 9-1   XIL Lookup Operation

A dither operation is an *inverse* lookup operation. As Figure 9-2 indicates, the dither operation matches a value (or values) in the source image—the image to be dithered—with a value from the *output* side of the lookup table being used, and then writes the corresponding value from the *input* side of the lookup table to a single-band destination image.

> **Note** – Since the values in the source image don't actually match values in the output side of the table, each source-image value is paired with the value in the table closest to it.

Figure 9-2   Dithering an Image

The purpose of this dithering operation is to produce an image that, when mapped forward through the lookup table, produces an image as similar as possible to the original source image. Another way to state this is that the dithering operation produces an image that, when displayed using the lookup table as its colormap, looks as much like the original as possible.

Such dithering operations have many applications, but a couple of them are by far the most common:

- To convert a true-color (3-band `XIL_BYTE`) image to a pseudocolor (1-band `XIL_BYTE`) image
- To convert a grayscale (1-band `XIL_BYTE`) image to a monochrome (1-band `XIL_BIT`) image

Methods of Dithering

This section contains information about the three functions the XIL library provides for performing dithering operations: `xil_nearest_color()`, `xil_error_diffusion()`, and `xil_ordered_dither()`.

xil_nearest_color()

The `xil_nearest_color()` function is the simplest of the dithering functions in that its algorithm for performing the inverse lookup described above is the most straightforward. This algorithm includes no provision for eliminating unwanted contours in the dithered image.

The function prototype for `xil_nearest_color()` is shown below.

```
void xil_nearest_color(XilImage src, XilImage dst,
    XilLookup lookup);
```

The `src` and `dst` parameters are handles to the source image—the image to be dithered—and the destination image. The `lookup` parameter is a handle to the lookup table through which the source image is passed to produce the destination. There are actually two types of lookup tables that can be referred to here. One is the generic lookup table that is used for lookup operations, and the other is a special type of lookup table called a *colorcube*. The basic difference between the two types of lookup tables is this: when you create a generic lookup table, you specify the dimensions of the table and the data to be stored in the table; when you create a colorcube, you specify the dimensions of the colorcube, and the function you use to create the colorcube fills in the data in the table for you. The sections below provide additional information about these two types of lookup tables.

Lookup Tables

You can create a generic lookup table using either of two functions: `xil_lookup_create()` or `xil_choose_colormap()`. In either case, `xil_nearest_color()` pairs the values in your source image with values on the output side of the lookup table by searching for nearest matches. This means that for each pixel in your source image, `xil_nearest_color()` must

examine each entry in the lookup table. This is a time consuming process. However, generic lookup tables do give you the best control over the contents of your lookup table and, therefore, over the quality of the dithered image.

The function prototype for `xil_lookup_create()` is shown below.

```
XilLookup xil_lookup_create(XilSystemState state,
    XilDataType input_datatype, XilDataType output_datatype,
    unsigned int output_nbands, unsigned int num_entries,
    short first_entry_offset, void *table_data);
```

For more information on the parameters to `xil_lookup_create()` and on creating a generic lookup table, see "Creating a Lookup Table" on page 272.

Here's an example of a time when you might want to use this function to create your lookup table: You have a true-color image that you want to dither to a single-band `XIL_BYTE` image, and you want to be able to display the dithered image using the currently installed X colormap. The code you use to create this lookup table might look like this.

```
/* Read the color values stored in the X colormap and write them
   to an array of Xil_unsigned8. The values should be written
   in the following order: b₀, g₀, r₀, b₁, g₁, r₁, and so on.
   Let's say the array is called table_data */

xil_lookup_create(state, XIL_BYTE, XIL_BYTE, 3, 256, 0,
    table_data);
```

Using this table for your dither operation enables you to produce a dithered image that contains the maximum number of unique values (256) and to display the image without having to write anything to the X colormap. You won't have to worry about colormap flashing. The disadvantage of this method is that there may be few or no exact matches between the RGB values in your source image and the RGB values on the output side of the lookup table. Clearly, this can affect how accurately the dithered image, when displayed, reflects the original source image.

The other function you might use to create a lookup table for dithering the true-color image mentioned above to a pseudocolor image is `xil_choose_colormap()`. This function returns the best lookup table of a

particular size to use in dithering the image. The best lookup table is defined to be the one that contains as many as possible of the most frequently used colors in the source image.

The function prototype for `xil_choose_colormap()` is shown below.

```
XilLookup xil_choose_colormap(XilImage src, unsigned int size);
```

When you use this function to create your lookup table, there are two basic approaches you might take to dithering your true-color image. One approach is to use `xil_choose_colormap()` to create a lookup table that contains 256 entries. This strategy produces a dithered image that looks very much like the original source image when displayed because the 256 most frequently used colors in the original are reproduced exactly. However, you also have to get 256 specific colors into a hardware colormap to display your image. This leads to colormap flashing on most displays.

A second approach is to use `xil_choose_colormap()` to create a lookup table with fewer than 256 entries, say 240 entries. You still have most of the colors you need to display your image, and you can write those colors to your hardware colormap (if you only have one) without overwriting the first 16 entries in that colormap. This should prevent your application from contending with the window system tools for colors.

Note – `xil_choose_colormap()` accepts only 3-banded `XIL_BYTE` source images.

Colorcubes

You create a colorcube using the `xil_colorcube_create()` function. The prototype for this function is shown below.

```
XilLookup xil_colorcube_create(XilSystemState state,
    XilDataType input_type, XilDataType output_type,
    unsigned int nbands, short offset, int multipliers[],
    unsigned int dimensions[]);
```

Note that the parameters to this function are similar to those for
`xil_lookup_create()`. However, you don't specify a number of entries in
the lookup table or the data to be loaded into the table. Instead you provide
arrays called `dimensions` and `multipliers`. The number of elements in both
of these arrays must equal the number of bands in the image being dithered.

To understand the roles of `dimensions` and `multipliers`, consider the
situation where you want to dither an `XIL_BYTE` RGB image to a 1-band
`XIL_BYTE` image. To handle this case, you might declare and initialize
`dimensions` and `multipliers` shown below.

```
unsigned int dimensions[] = {4, 9, 6};
int multipliers[] = {1, 4, 36};
```

These values would lead to the creation of a colorcube that would dither blue
values in the source image to one of 4 blue levels, green values to one of 9
green levels, and red values to one of 6 red levels. You could picture this
colorcube as a cube with dimensions of 4, 9, and 6, but it's probably more
helpful to think of it as a lookup table with three bands on the output side.
Figure 9-3 shows what the first 22 elements of the 216-element table would
look like.

Values to be written
to indexed-color
image.

	Blue values	Green values	Red values
0	0	0	0
1	85	0	0
2	170	0	0
3	255	0	0
4	0	32	0
5	85	32	0
6	170	32	0
7	255	32	0
8	0	64	0
9	85	64	0
10	170	64	0
11	255	64	0
12	0	96	0
13	85	96	0
14	170	96	0
15	255	96	0
16	0	128	0
17	85	128	0
18	170	128	0
19	255	128	0
20	0	159	0
21	85	159	0

Figure 9-3   Colorcube for Dithering a True-Color Image to a Pseudocolor Image

This illustration should help clarify the significance of the three elements of `multipliers`. The `multipliers[0]` element indicates the distance between changes in blue levels on the output side of the table, `multipliers[1]` indicates the distance between changes in green levels, and `multipliers[2]` indicates the distance between changes in red levels. That is, the blue level changes with each entry, the green level changes every fourth entry, and the red level changes every thirty-sixth entry.

Note – The elements of `multipliers` can be negative numbers. Negative numbers indicate that the values in a color ramp should appear in decreasing as opposed to increasing order.

The values in the table are calculated by `xil_colorcube_create()` based on the dimensions and multipliers you supply. This means that when you dither an image using a colorcube, the dither function does not need to search the output side of a lookup table to find a set of values. Instead, it can calculate the value to be written to the dithered image using the values stored in the `dimensions` and `multipliers` arrays. As a result, dithering operations that use a colorcube are much faster than those that use a generic lookup table. What you give up is control over the exact contents of the colorcube.

One other note about colorcubes. Although they are called color*cubes*, colorcubes need not have three dimensions. As mentioned earlier, a colorcube has a number of dimensions equal to the number of bands in the image to be dithered. Therefore, to create a colorcube suitable for dithering a grayscale image to a monochrome image, you might define `dimensions` and `multipliers` as follows.

```
unsigned int dimensions[] = {2};
int multipliers[] = {-1};
```

Note – The XIL library creates two colorcubes when you initialize the library. One of these has the dimensions 4:9:6 and is useful for dithering RGB images to 216 colors. The other has the dimensions 8:5:5 and is useful for dithering YC_bC_r images to 200 colors. To get a handle to one of these colorcubes, use the `xil_lookup_get_by_name()` function.

In addition to `xil_colorcube_create()`, the XIL library contains the colorcube-related functions listed and described in Table 9-3.

Table 9-3   Functions for Managing Colorcubes

Function	Description
`xil_lookup_get_colorcube()`	Determines whether a lookup table is a colorcube or a generic lookup table
`xil_lookup_get_colorcube_info()`	Determines whether a lookup table is a colorcube and, if it is, returns the dimensions and multipliers used to create the colorcube and the colorcube's origin

`xil_error_diffusion()`

The `xil_error_diffusion()` function is similar to `xil_nearest_color()`, but in addition to performing an inverse lookup, it uses a process called *error diffusion* to deal with any difference between a source-image pixel value and a value on the output side of a lookup table with which that pixel value is matched. The difference (or error) is distributed to the source-image pixels immediately to the right of and below the last pixel processed.

The function prototype for `xil_error_diffusion()` is shown below.

```
void xil_error_diffusion(XilImage src, XilImage dst,
    XilLookup lookup, XilKernel distribution);
```

The first three parameters are the same as the parameters to `xil_nearest_color()`. The `src` parameter is a handle to your source image, and `dst` is a handle to your destination image. The `lookup` parameter can be either a generic lookup table or a special type of lookup table called a colorcube. For information about these lookup tables, see "Lookup Tables" on page 164 and "Colorcubes" on page 166. The final parameter, `distribution`, is an error-distribution kernel.

Note – XIL kernels (data structures of type `XilKernel`) are used primarily for convolution operations, which are discussed in "Filtering an Image" on page 257. The utility functions that affect kernels are also discussed in that section.

You create the error-distribution kernel using the `xil_kernel_create()` function, whose function prototype is shown below.

```
XilKernel xil_kernel_create(XilSystemState system_state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, float *data);
```

The `width` and `height` parameters determine the size of the kernel, `keyx` and `keyy` determine the kernel's *origin*, and `data` is a pointer to the floating-point values to be stored in the kernel. For example, the code below creates a 3-by-3 kernel with an origin of 1,1.

```
XilKernel kernel;
float kernel_data[] = {
    0/16.0, 0/16.0, 0/16.0,
    0/16.0, 0/16.0, 7/16.0,
    3/16.0, 5/16.0, 1/16.0
};

kernel = xil_kernel_create(state, 3, 3, 1, 1, kernel_data);
```

Note – The library provides a special shorthand method of creating the kernel shown above because it is used so commonly in error-diffusion operations. This method is to call the `xil_kernel_get_by_name()` function using a name of "`floyd-steinberg`".

Figure 9-4 shows what this kernel looks like.

Figure 9-4   Error-Distribution Kernel

Note – The value of the kernel's origin and of the elements above and to the left of the origin must be 0 when you're calling `xil_error_diffusion()`.

Here's how `xil_error_diffusion()` uses the kernel. Say that you're dithering a grayscale image with values in the range 0 to 255 to a grayscale image with values in the range 0 to 4. This situation is shown in Figure 9-5.

Figure 9-5   Using `xil_error_diffusion()` to Dither an Image

When the pixel in the middle of the top scanline of the source image (31) is passed through the lookup table, it is matched with a 26, so a 0 is written to the destination image. When the destination is displayed using the lookup table as its colormap, this pixel has the value 26. Thus, in the dither-and-reconstruction process, there is an error of 5 (31 - 26).

To counteract the loss of data inherent in this process, `xil_error_diffusion()` distributes this error before passing the next pixel (219) through the lookup table. The way in which the error is distributed depends on the kernel created earlier. See Figure 9-6 for an example.

Figure 9-6　　Error Diffusion

The kernel is laid on top of the source image so that its origin aligns with the last pixel to be passed through the lookup table. Before the next pixel is passed through, the following processing takes place:

- The pixel at 0,2 is set to 219 + (5 * (7 / 16.0))
- The pixel at 1,0 is set to 187 + (5 * (3 / 16.0))
- The pixel at 1,1 is set to 121 + (5 * (5 / 16.0))

- The pixel at 1,2 is set to 73 + (5 * (1 / 16.0))

This processing is fairly time-consuming, but can greatly reduce contouring in the dithered image.

xil_ordered_dither()

The XIL library's third dithering function is `xil_ordered_dither()`. Like `xil_nearest_color()` and `xil_error_diffusion()`, `xil_ordered_dither()` processes an image by performing an inverse lookup. However, two characteristics distinguish `xil_ordered_dither()` from the other dithering functions.

- The lookup table the function uses to do its job must be a colorcube; it cannot be a generic lookup table. (For information about the differences between colorcubes and generic lookup tables, see "Lookup Tables" on page 164 and "Colorcubes" on page 166.)
- `xil_ordered_dither()` makes use of an XIL data structure called a dither mask to help eliminate contouring in the dithered image

The function prototype for `xil_ordered_dither()` is shown below.

```
void xil_ordered_dither(XilImage src, XilImage dst,
    XilLookup lookup, XilDitherMask mask);
```

The parameters `src` and `dst` are handles to the source and destination images, and `lookup` is the lookup table to be used for the dither operation. As mentioned above, this lookup table must be a colorcube; that is, it must have been created with the `xil_colorcube_create()` function—or `xil_lookup_get_by_name()` function. The final parameter, `mask`, is the dither mask, a data structure of type `XilDitherMask`.

You create the dither mask using the `xil_dithermask_create()` function, whose prototype is shown below.

```
XilDitherMask xil_dithermask_create(XilSystemState state,
        unsigned int width, unsigned int height,
        unsigned int nbands, float *data);
```

The `width` and `height` parameters define the width and height of the dither mask in pixels. These values are user defined. The next parameter, `nbands`, determines the number of bands in the dither mask. This number must match the number of bands in the image being dithered. Finally, `data` is a pointer to the data to be stored in the dither mask. All the values in the mask must fall in the range 0.0 to 1.0. For example, the code below creates a 1-band, 4-by-4 dither mask.

```
XilDitherMask mask;
float mask_data[] = {
     0/16.0, 8/16.0, 2/16.0,10/16.0,
    12/16.0, 4/16.0,14/16.0, 6/16.0,
     3/16.0,11/16.0, 1/16.0, 9/16.0,
    15/16.0, 7/16.0,13/16.0, 5/16.0
}

mask = xil_dithermask_create(state, 4, 4, 1, mask_data);
```

Note – When you initialize the XIL library, four standard dither masks are created: 1- and 3-band 4-by-4 masks and 1- and 3-band 8-by-8 masks. To get a handle to one of these masks, you use the `xil_dithermask_get_by_name()` function.

Here's how the mask is used. Assume that you're dithering a grayscale image with values in the range 0 to 255 and that the destination is to contain values in the range 0 to 15. First, think of the 4-by-4 dither mask as having been replicated over the entire source image, as depicted in Figure 9-7.

8-by-8 source image

Figure 9-7 Dither Mask Replicated over a Source Image

At this point, each pixel in each 4-by-4 block of the image has been associated with (in this case) a unique dither-mask value. For each pixel in the image, the `xil_ordered_dither()` function performs the following sequence:

1. It passes the value through the lookup table.

The function actually divides values in the source image by 17 (the number of values that can be represented in 8 bits divided by the length of the colorcube minus one). That is, if no further processing were to take place, a 200 in the source image would map to an 11 since 200 divided by 17 equals 11.76.

2. It considers the dither-mask value associated with the source-image pixel value.

The dither function now compares the fractional part of the quotient from the division operation with the source pixel's dither-mask value. If the fractional part is greater than the dither-mask value, the dividend shown above (11) is incremented by one, so a 12 is written to the destination image. Otherwise, an 11 is written to the image.

When you use the dither mask in this way, you're essentially performing averaging. This averaging helps prevent undesirable contours from appearing in the dithered image.

Table 9-4 lists and describes the XIL utility functions that affect dither masks.

Table 9-4   Utility Functions for Dither Masks

Function	Description
`xil_dithermask_destroy()`	Deallocates the memory used by a dither mask
`xil_dithermask_get_height()`	Returns the height of a dither mask in pixels
`xil_dithermask_get_width()`	Returns the width of a dither mask in pixels
`xil_dithermask_get_nbands()`	Returns the number of bands in a dither mask
`xil_dithermask_create_copy()`	Returns a copy of a dither mask
`xil_dithermask_set_name()`	Sets the name of a dither mask
`xil_dithermask_get_name()`	Returns a copy of a dither mask's name
`xil_dithermask_get_values()`	Returns a copy of the internal values in a dither mask

When to Use Each Dithering Function

When you're trying to decide on the best way to perform a dither operation, it's helpful to think of the matrix shown in Table 9-5.

Table 9-5   Review of Dithering Operations

Function	Generic Lookup	Colorcube	Speed	Quality
`xil_nearest_color()`	X	X	fastest	poor
`xil_error_diffusion()`	X	X	slow	best
`xil_ordered_dither()`		X	faster	better

If you look at the right column—labeled "Colorcube"—you'll see that each of the dithering functions can perform its inverse lookup using a colorcube. If you compare the functions used with a colorcube, you'll see that choosing one over another involves a trade-off between speed and quality. The `xil_nearest_color()` function is the fastest, but produces the poorest quality image, and `xil_error_diffusion()` is the slowest, but produces the best quality image. The `xil_ordered_dither()` function is somewhat faster than `xil_error_diffusion()`, but does not produce quite as clear an image.

Similarly, if you compare operations that use generic lookup tables, the trade-off is one between speed and quality. The guidelines mentioned in the preceding paragraph apply, except that `xil_ordered_dither()` cannot work with a generic lookup table.

Now, if you read the table from left to right, you can compare operations that use generic lookup tables with corresponding operations that use colorcubes. Operations that use colorcubes are much faster than their counterparts because they do not need to search a lookup table. On the other hand, when you use a colorcube for dithering, you lose some of your control over the quality of the dithered image, because you cannot specify the exact values in the colorcube. You also lose the ability to use the colors in the currently installed colormap, again because you don't specify the values to be written to the lookup table.

Color Space Manipulation

The XIL library provides functions for manipulating the color spaces associated with an image.

Converting an Image to a Different Color Space

The XIL library enables you to convert an image from any one to any other of the following color spaces:

- CCIR Rec. 709 RGB
- A linear version of CCIR Rec. 709 RGB
- CCIR Rec. 709 YC_bC_r
- A luminance-only space derived from CCIR Rec. 709 YC_bC_r
- A linear version of the luminance-only space mentioned above
- CCIR Rec. 601 YC_bC_r
- A luminance-only space derived from CCIR Rec. 601 YC_bC_r
- A YC_bC_r color space defined by Kodak for PhotoCD
- A linear CMY
- A linear CMYK

The procedure for converting an image from one color space to another is outlined below.

1. **Set the color-space attribute of your source image.**
   When you create an XIL image, it does not have a color space associated
   with it. To associate one with it, you must call the
   `xil_colorspace_get_by_name()` function to get the appropriate color-
   space data structure; then, you call `xil_set_colorspace()` to set the
   attribute.

   The following code fragment shows a program getting an object of type
   `XilColorspace` that describes a linear RGB color space.

   ```
   XilColorspace colorspace;

   colorspace = xil_colorspace_get_by_name(state, "rgblinear");
   ```

   The `state` parameter is a handle to the system state, and `rgblinear` is a
   string identifying the color space. The strings you use to identify the various
   XIL color spaces are listed in Table 9-6.

   Table 9-6 Strings Used to Specify Color Spaces

Color Space	String
CCIR Rec. 709 RGB	rgb709
Linear version of CCIR Rec. 709 RGB	rgblinear
CCIR Rec. 709 YC_bC_r	ycc709
Luminance-only space derived from CCIR Rec. 709 YC_bC_r	y709
Linear version of y709	ylinear
CCIR Rec. 601 YC_bC_r	ycc601
Luminance-only space derived from CCIR Rec. 601 YC_bC_r	y601
YC_bC_r color space defined by Kodak for Photo CD	photoycc
Linear CMY	cmy
Linear CMYK	cmyk

After you've gotten this color-space data structure, you set an image's color-space attribute using the `xil_set_colorspace()` function.

```
XilImage image;
XilColorspace colorspace;

xil_set_colorspace(image, colorspace);
```

2. Create a destination image, and assign it the color space to which you want to convert your source image.
The destination image must have the same width, height, and data type as the source image, and it must have a number of bands appropriate to the color space you assign it. That is, if you want to assign the destination image the `ycc709` color space, it must have three bands. You assign a color space to the destination using the same method you used to assign a color space to the source image.

Note – It is possible to perform an in-place color conversion if the destination image is a child of the source and has the same width and height as the source. This procedure is not generally recommended, however, because the color-conversion operation overwrites some or all of the data in the source image.

3. Perform the color conversion.
You perform the color conversion by calling the `xil_color_convert()` function. This function takes handles to your source and destination images as its two parameters.

Note – Regions of interest are ignored when you perform a color conversion.

There are many applications for this type of color conversion. For example, you may want to:

- Convert an image from a YC_bC_r color space to an RGB color space in order to display the image

- Convert an image from an RGB color space to a YC_bC_r color space to prepare it as input to the JPEG or CellB compressor

- Convert an image to the CMY or CMYK color space to prepare it to be printed on a subtractive color printer

- Convert an RGB or YC$_b$C$_r$ image to the `y709`, `ylinear`, or `y601` color space as a way of converting a 24-bit image to an 8-bit grayscale image

- Convert an `XIL_BYTE` image from a linear color space to a gamma-corrected color space to prevent contouring in low-intensity regions of the image

Color Correcting an Image

Each display device has an associated profile containing characteristics about that device. If a color image is created on one device, it may not display the colors with the exact same intensities, hues, and so forth on other devices.

To compensate for differences in device characteristics, you can use the `xil_color_correct()` function. It uses Kodak Color Management System (KCMS) functions integrated into the XIL library to color correct images displayed on various devices. For details on KCMS, you should read the *KCMS Application Developer's Guide.*

≡ *9*

The following is a simple example that shows how to compensate for differences in device characteristics.

```
#define SRC_PROFILE "kcmsEKphcdcn.inp"
#define DST_PROFILE "kcmsEKsony20.mon"

    XilSystemState      state;
    XilImage            src, dst;
    XilColorspace       cspaces[2];
    XilColorspaceList   cspaceList;
    float               values[3];

    /* Open XIL and create 2 three banded byte images */
    state = xil_open();
    src = xil_create(state, 512, 512, 3, XIL_BYTE);
    dst = xil_create(state, 512, 512, 3, XIL_BYTE);

    /* Set some values on the src image */
    values[0] = 0.0;
    values[1] = 255.0;
    values[2] = 0.0;
    xil_set_value(src, values);

    /*
     * Create the colorspaces using a filename
     */
    cspaces[0] = xil_colorspace_create(state,
                    XIL_COLORSPACE_FILENAME,SRC_PROFILE);
    cspaces[1] = xil_colorspace_create(state,
                    XIL_COLORSPACE_FILENAME,DST_PROFILE);

    /* Create the colorspace list */
    cspaceList = xil_colorspacelist_create(state, cspaces, 2);

    /* Color correct the images */
    xil_color_correct(src, dst, cspaceList);


    /* Destroy the colorspace list then the colorspaces */
    xil_colorspacelist_destroy(cspaceList);
    xil_colorspace_destroy(cspaces[0]);
    xil_colorspace_destroy(cspaces[1]);
```

The example shows the following steps:

1. **It creates source and destination images whose width, height, number of bands, and data type match.**

2. **It uses the** `xil_colorspace_create()` **function to create a color space (**`cspaces[0]`**) for the source image based on a given profile type.**
   The `xil_colorspace_create()` function accepts profiles from any of three sources based on the `XilColorspaceType` enumeration type, which is shown below.

```
typedef enum {
    XIL_COLORSPACE_NAME,
    XIL_COLORSPACE_FILENAME,
    XIL_COLORSPACE_KCS_ID
} XilColorspaceType;
```

   In this case, `xil_colorspace_create()` specifies the `XIL_COLORSPACE_FILENAME` enumerator. It tells XIL to use the profile `SRC_PROFILE` in the current directory to create the color space.

```
cspaces[0] = xil_colorspace_create(state,
                 XIL_COLORSPACE_FILENAME,SRC_PROFILE);
```

   The other enumerators of `XilColorspaceType` are `XIL_COLORSPACE_NAME` and `XIL_COLORSPACE_KCS_ID`. `XIL_COLORSPACE_NAME` tells XIL to use one of the color spaces in the default list of color spaces listed in "Color Space Manipulation" on page 178. The `XIL_COLORSPACE_KCS_ID` enumerator tells XIL to use a KCMS profile you created.

3. **It uses the** `xil_colorspace_create()` **function to create a color space (**`cspaces[1]`**) for the destination image.**
   This call is shown below.

```
cspaces[1] = xil_colorspace_create(state,
                 XIL_COLORSPACE_FILENAME,DST_PROFILE);
```

Note – If you move your image to any other display devices between the source and destination, you need to call `xil_colorspace_create()` to create a color space based on a profile for each of these devices as well.

---

4. Use the `xil_colorspacelist_create()` **function to create a list of the color spaces.**
The order of the color spaces in the color space list is important. The first color space is the one associated with the source image and the last, the color space associated with the destination image. If the image is routed to any other devices between the source and destination devices, color spaces for those devices must be created and included in the color space list in the order in which they are used.

5. Use the `xil_color_correct()` **function to color correct the images.**
This call is shown below.

```
xil_color_correct(src, dst, cspaceList);
```

The `cspaceList` parameter is the color space list with color spaces in the order described in step 4.

The following is additional information you should consider when color correcting an image:

- When no longer needed, you should destroy the color spaces and the color space list. The calls to do this in the example are shown below.

```
xil_colorspacelist_destroy(cspaceList);
    xil_colorspace_destroy(cspaces[0]);
    xil_colorspace_destroy(cspaces[1]);
```

- If you had previously set color spaces for the images using the `xil_set_colorspace()` function (described in "Color Space Manipulation" on page 178), they would be ignored by `xil_color_correct()`. The `xil_color_correct()`function only recognizes the color spaces you specify in the color space list.
- If, however, you call `xil_color_correct()` with an `XilColorspaceList` object containing two `XIL_COLORSPACE_NAME` color spaces, this is equivalent to calling `xil_set_colorspace()` on the source

and destination images and then calling `xil_color_convert()`. (The procedure for doing this is described in "Color Space Manipulation" on page 178.)

- Finally, to get the color space of a device image, you must use the COLORSPACE attribute. You can then add this color space to the color space list in the call to `xil_color_correct()`. The following code fragment shows how to get the attribute.

```
XilColorspace* cspace
...
wimage = xil_create_from_window(state, display, win);
...
xil_get_device_attribute(wimage, "COLORSPACE", (void**)&space);
```

Additional Color Space Manipulation Functions

The XIL library provides the additional color space manipulation functions listed and described in Table 9-7.

Table 9-7   Color Space Manipulation Functions

Function	Description
`xil_colorspace_get_name()`	Gets any associated name of the color space, or, if no name is set on the color space, returns NULL
`xil_colorspace_get_type()`	Gets the type of the color space and the associated data
`xil_colorspace_set_name()`	Sets the name on the color space with the specified name
`xil_colorspacelist_get_by_name()`	Returns a color space list associated with the name, or, if there is no color space list, returns NULL
`xil_colorspacelist_get_name()`	Gets any associated name of the color space list, or, if there is no associated color space list, returns NULL
`xil_colorspacelist_set_name()`	Sets the name of the color space list with the specified name

Color Spaces Represented by I/O Devices

COLORSPACE is a required attribute that XIL device images provide. This attribute describes the color space of the device. You can use this attribute to correct the color of images to and from the device.

Black Generation

The XIL library also enables you to perform undercolor removal and black generation on CMYK images—images to be printed on a four-color printing system. Normally, the process of undercolor removal and black generation goes something like this. For each pixel in the image:

- Black (K) equals the minimum of C, M, and Y
- C = C - K
- M = M - K
- Y = Y - K

However, the XIL function that performs undercolor removal and black generation gives you more control over how much cyan, magenta, and yellow are removed from your image and how much black is added to it. This function is called xil_black_generation(), and its function prototype is shown below.

```
void xil_black_generation(XilImage src, XilImage dst,
    float black, float undercolor);
```

The src and dst parameters are handles to your source and destination images. Both images must be CMYK images. Therefore, if you want to process a CMY image, you must first convert it to a CMYK image. For instructions on how to do this, see "Color Space Manipulation" on page 178. In-place operations are allowed.

The black and undercolor parameters are used as follows. For each pixel in your source image:

- K = (black * minimum of C, M, and Y)
- C = C - (undercolor * minimum of C, M, and Y)
- M = M - (undercolor * minimum of C, M, and Y)
- Y = Y - (undercolor * minimum of C, M, and Y)

Thus, if both `black` and `undercolor` are set to 1.0, you'll get standard undercolor removal. You'll frequently achieve better results, though, if the percentage of black you add to your image is slightly higher than the percentage of C, M, and Y you remove from it. For example, you might try using the call shown here.

```
XilImage src, dst;

xil_black_generation(src, dst, 0.7, 0.5);
```

Note – Regions of interest are ignored when you perform undercolor removal.

9

Error Handling *10*

This chapter discusses the XIL default error handler and how to write and install your own error handler(s).

This chapter has the following main sections:

XIL Error Handling Summary

The list below summarizes the most important facts about XIL error handling.

- Error handling is asynchronous.

  For example, assume that your program includes a call to `xil_copy()` that attempts to copy a three-band image to a single-band image. This is an error, and at some point the library reports that the source and destination images do not contain the same number of bands. However, this error may not be reported when the line in your program containing the `xil_copy()` is called. This is true because the XIL library's deferred execution scheme enables it to store many operations before executing them. (For more information on deferred execution, see Chapter 23, "Acceleration in XIL Programs.") Later, when a set of stored operations is performed, the error is reported.

≡ *10*

The fact that error reporting is asynchronous does not really affect the way you handle errors in your XIL programs, but you should be aware of it.

- The amount of error handling that you can do by checking the return values of XIL functions is limited.

  About half of the functions in the XIL library return `void`, and for the remaining functions, while you may be able to tell from the return value whether an error occurred, you won't be able to determine exactly what the error was. To handle errors properly, your program needs to examine the error object (of type `XilError`) that the library creates whenever an error occurs. This object contains such information as an error ID and a string containing a description of the error.

- Your can deal with the error object referred to above in one of two ways:

  One way is to have your application do nothing. In this case, the library's default error handler looks at the error object and prints a message to `stderr`. As shown in "An Error Handler Example" on page 198, this option isn't recommended because the default handler prints information that is intended for application developers and isn't useful to an end user.

  Alternately, write your own error handler and install it. This is the recommended method. For information on writing such an error handler, see "Writing an Error Handler." For information on installing the routine, see "Installing and Chaining Error Handlers" on page 200.

Note – You can actually have more than one error handler installed at a time. In addition, you can chain these error handlers so that if the first error handler in the chain does not deal with an error, it can pass the error object containing information about the error to a second error handler, and so on. This subject is explained in "Installing and Chaining Error Handlers" on page 200.

Writing an Error Handler

Any error handler you write should take one parameter, a handle to the error object the library created when the last error occurred. In addition, the function should return a value of type `Xil_boolean`. Thus, the header of your function should look like this.

```
Xil_boolean function-name(XilError error);
```

The function returns `TRUE` if it has handled the error and `FALSE` if it has not.

The XIL library contains a number of routines you can use in building the body of your error handler. These are discussed in "Functions You Can Call in Your Error Handler." To see how you might use these functions to write an error handler, see "An Error Handler Example" on page 198.

Functions You Can Call in Your Error Handler

This section discusses the XIL functions you can use in writing your own error handler. These functions fall broadly into two groups:

- Those that get information from the error object that the library passes to your error handler

- Those that retrieve information from objects of type `XilObject`.

Note – The error handler of a multithreaded application may only make XIL calls on the object that was returned to the error handler; doing otherwise could lead to a deadlock.

Getting Information From an Error Object

The functions listed in this section take as their only parameter the name of the error object the library has passed to your error handler.

`xil_error_get_string()`

This function returns an internationalized string (`char *`) that describes the error that just occurred. For a complete list of possible messages, see Appendix B, "XIL Error Messages."

You should either use this string immediately or make a copy of it because the pointer to it becomes invalid after the next call your program makes to `dgettext(3I)` or to one of the XIL error functions that returns a string.

You should not free or modify this string.

`xil_error_get_id()`

This function returns a string (`char *`) of the form `di-`*n*, where is *n* is a unique identifier. For a complete list of possible error IDs, see Appendix B, "XIL Error Messages."

You should not free or modify this string.

Note – Errors other than those generated by the base XIL library may return a device name instead of `di-` in the error identifier. You should check your device's documentation for more information.

`xil_error_get_category()`

This function returns an enumeration constant of type `XilErrorCategory`. The declaration of this enumeration is shown here.

```
enum XilErrorCategory {
    XIL_ERROR_SYSTEM,
    XIL_ERROR_RESOURCE,
    XIL_ERROR_ARITHMETIC,
    XIL_ERROR_CIS_DATA,
    XIL_ERROR_USER,
    XIL_ERROR_CONFIGURATION,
    XIL_ERROR_OTHER
    XIL_ERROR_INTERNAL
} XilErrorCategory;
```

Table 10-1 explains briefly what types of error fall into each category.

Table 10-1 XIL Error Categories

Category	Explanation
XIL_ERROR_SYSTEM	The library cannot perform an operation correctly. This type of error is usually a secondary error caused by a user, resource, or configuration error.
XIL_ERROR_RESOURCE	Usually means that there is insufficient memory for the library to create an image or other object.
XIL_ERROR_ARITHMETIC	Indicates an arithmetic error such as a divide by 0.
XIL_ERROR_CIS_DATA	Means that the data stream being decompressed does not conform to the appropriate datastream definition.
XIL_ERROR_USER	The programmer has specified an invalid parameter to an XIL function. For example, for a lookup operation, the data type of the source image and the input data type of the lookup table might not match.
XIL_ERROR_CONFIGURATION	Usually indicates that the XIL software has not been installed properly or that an environment variable has not been set properly. The specific error may be that the runtime system is unable to find a loadable driver that it needs.
XIL_ERROR_OTHER	Miscellaneous errors, such as Xlib and DGA errors.
XIL_ERROR_INTERNAL	Indicates an error in the XIL library.

`xil_error_get_category_string()`

This function returns a string (`char *`) that indicates which category of error has occurred. The strings for the possible categories are:

- `System`
- `Resource`
- `Arithmetic`
- `Cis Data`
- `User`
- `Configuration`
- `Other`

≡ 10

As with the string returned by `xil_error_get_string()`, you should either use this string immediately or make a copy of it because the pointer to it becomes invalid after the next call your program makes to `dgettext(3I)` or to one of the XIL error functions that returns a string.

You should not free this string.

xil_error_get_primary()

This function returns a value of type `Xil_boolean`, either `TRUE` or `FALSE`. A value of `TRUE` indicates that the error being processed was the primary cause of a problem you're seeing. For example, if you run out of memory and `xil_create()` is unable to create an image, the primary error produced is a resource error. The XIL library may also generate secondary errors as the `NULL` image is used internally.

xil_error_get_location()

This function returns a string (`char *`) that indicates where in the XIL library the error was generated. This location helps Customer Support to resolve any problems you report to them.

You should not free this string.

xil_error_get_object()

This function returns a handle of type `XilObject` to the object affected by the error. This object may be:

- A compressed image sequence (CIS)
- A color space
- A color space list
- A device
- A dither mask
- An error
- A histogram
- An image
- An image type
- An interpolation table
- A kernel
- A lookup table
- A region of interest
- A structuring element

- A storage object
- A system state object

Once you have this handle, you can use the functions discussed in the next section to obtain further information about the object.

Note – If the error did not affect a particular object, you receive a `NULL` handle.

`xil_error_is_warning()`

This function allows your application to determine whether an error returned is a warning that it can handle or an error resulting from a catastrophic event.

Getting Information About the Object Affected by the Error

If your error handler uses the function `xil_error_get_object()` to retrieve a handle to the object affected by the last error, you can use the functions discussed below to request further information about the object.

`xil_object_get_error_string()`

This function attempts to get from an object a string containing additional information about the object. For instance, if you query a CIS object, you get a string containing information about the values of the start frame, read frame, and write frame at the time of the error.

The prototype for this function is shown below.

```
void xil_object_get_error_string(XilObject object,
    char *string, int string_size);
```

The `object` parameter is the object handle returned by `xil_error_get_object()`. `string` is an array of `chars` in which the string is returned; and `string_size` is the maximum number of characters the array can contain.

In the following code fragment, an error handler uses
`xil_object_get_error_string()` to request information about the object
affected by an error and prints any information that is available.

```
#define MAX 1024

Xil_boolean UserDefinedErrorFunc(XilError error)
{
    XilObject object;
    char buffer[MAX];

    object = xil_error_get_object(error);
    if (object) {
        xil_object_get_error_string(object, buffer, MAX);
        if (buffer[0] != 0)
            fprintf(stderr, "object info: %s\n", buffer);
```

`xil_object_get_type()`

You use this function to determine what kind of object the last error affected.
The prototype for the function is shown below.

```
XilObjectType xil_object_get_type(XilObject object);
```

...

The parameter to the function is the object handle returned by
`xil_get_error_object()`. The return value is an enumeration constant of
type `XilObjectType`. The declaration of this enumeration is written as shown
here.

```
typedef enum {
    XIL_IMAGE,
    XIL_IMAGE_TYPE,
    XIL_LOOKUP,
    XIL_CIS,
    XIL_DITHER_MASK,
    XIL_KERNEL,
    XIL_SEL,
    XIL_ROI,
    XIL_HISTOGRAM,
    XIL_COLORSPACE,
    XIL_ATTRIBUTE,
    XIL_INTERPOLATION_TABLE,
    XIL_STORAGE,
    XIL_DEVICE,
    XIL_COLORSPACE_LIST
} XilObjectType;
```

Note – The `XIL_ROI_LIST` enumerator is obsolete.

Once you know the type of object that was affected by the last error, you can
cast the object handle returned by `xil_get_error_object()` to a handle to
the particular type of object you have. For example, if the object is an image,
you can cast the handle of type `XilObject` to one of type `XilImage`. Doing so
gives you the ability to write code like the following.

```
XilObject object;

object = xil_error_get_object(error);
if (object)
    if (xil_object_get_type(object) == XIL_IMAGE)
        fprintf(stderr, "image bands: %d\n",
            xil_get_nbands((XilImage)object));
```

Because the error handler has a handle to the image, it can use any of the XIL functions that read image attributes to get additional information about the image. In this case, the error handler finds out how many bands are in the image.

An Error Handler Example

The library's default error handler prints a message like the following one each time an error occurs.

```
XilDefaultErrorFunc:
    error category: User
      error string: Destination colorspace not specified
           error id: di-203
    primary error detected at location colorConvert79 in XIL
```

As you can see, the message indicates that the XIL default error handler handled the error, and it prints the category of the error, an error message, an error ID, and the location at which the error occurred, indicating whether the error was a primary or secondary error.

This information should be useful to the application developer because it verifies that the XIL handler handled this error. Also the error location is useful when contacting Customer Support. However, you wouldn't want your application to identify the error handler and error location because this information isn't useful to an end user and, moreover, would be confusing to

that user. The following code shows how you could implement an error
handler that doesn't identify the error handler and doesn't print the error
location.

```
Xil_boolean UserDefinedErrorFunc(XilError error)
{
    if (xil_error_get_primary(error) == TRUE)
        fprintf(stderr, "\nPrimary Error:\n");
    else
        fprintf(stderr, "\nSecondary Error:\n");
    fprintf(stderr, "   error category: %s\n",
        xil_error_get_category_string(error));
    fprintf(stderr, "     error string: %s\n",
        xil_error_get_string(error));
    fprintf(stderr, "         error id: %s\n",
        xil_error_get_id(error));
    return TRUE;
}
```

≡ 10

Installing and Chaining Error Handlers

By default, when an XIL error occurs, the library creates an error object of type `XilError` and passes that object to the default error handler. To override this behavior, you can write your own error handler and install it. Once you install your error handler, the library calls it instead of the default error handler when an error occurs. If you write and install more than one error handler, the library always calls the most recently installed error routine. For more information on installing error handlers, see "Installing Error Handlers."

The XIL library allows you to have more than one user-defined error routine installed at the same time. At first glance, this seems a waste because the library can only call the most recently installed routine when an error occurs. However, having more than one user-defined error handler installed is often useful because the library allows you to call one error handler from another, an action referred to as *chaining*. For an explanation of why you might want to chain error routines and how to chain them, see "Chaining Error Handlers" on page 203.

Installing Error Handlers

To install an error handler you've written, you call the function `xil_install_error_handler()`, whose prototype is shown here.

```
int xil_install_error_handler(XilSystemState state,
    XilErrorFunc func);
```

The function's return value indicates whether the installation was successful. A value of `XIL_SUCCESS` means that it was successful, and a value of `XIL_FAILURE` means that it wasn't.

The `state` parameter is the system-state data structure you received when you initialized the library, and `func` is a pointer to your error handler. (`XilErrorFunc` is a defined type representing a pointer to a function.) Thus, the call you would use to install the error handler shown in "An Error Handler Example" on page 198 would look like this.

```
XilSystemState state;

xil_install_error_handler(state, UserDefinedErrorFunc);
```

When you install an error handler, the library adds it to a list of error handlers that it recognizes. Figure 10-1 shows what the list would look like at this point.

Figure 10-1 List of Error Handlers

If you then installed a third error handler, called `UserDefinedErrorFunc2()`, the list would grow as shown in Figure 10-2.

Figure 10-2   Adding to the List of Error Handlers

When an XIL error occurs, the library always calls the topmost routine (the last routine to be installed) in the list.

The library also includes a routine for removing error handlers from this list. Its name is `xil_remove_error_handler()`.

```
void xil_remove_error_handler(XilSystemState state,
    XilErrorFunc func);
```

The parameters to this function are the same as those for `xil_install_error_handler()`.

Note – If the same error handler has been installed twice, only the most recent version of the error handler is removed.

Chaining Error Handlers

Again, having more than one error handler installed is useful because the library enables you to call one error handler from another. From the topmost error handler in a list of handlers, you can call either the next-to-last error handler to be installed or the default error handler. Thus, in your topmost error routine, you might handle only resource errors. If the error you're passed is a resource error, you take whatever action is appropriate and return TRUE; otherwise, you call the next error handler in the list. This next error handler might also check for one special type of error. Like the previous routine, if it is passed the proper type of error, it handles the error and returns TRUE; if it receives another type of error, it might call the error handler below it on the list. This process can continue until the error handler being called is the default error handler. This routine handles any XIL error and always returns TRUE.

To chain error handlers you use the `xil_call_next_error_handler()` and the `xil_default_error_handler()` functions.

```
Xil_boolean xil_call_next_error_handler(XilError error);

Xil_boolean xil_default_error_handler(XilError error);
```

The `error` parameter is the error object that the library passed to the topmost error handler in its list of error handlers (the last one to be installed). The return value of both functions indicates whether the error routine being called handled the error. A return value of TRUE indicates that the routine did handle the error, and a return value of FALSE indicates that it didn't.

The brief example below shows the `xil_call_next_error_handler()` function being used in an error-handling routine. This routine simply ensures that if the error is a resource error, the program exits after handling the error in the normal way.

```
Xil_boolean resource_errors(XilError error)
{
    if (xil_error_get_category(error) == XIL_ERROR_RESOURCE)
        exit(1);
    return (xil_call_next_error_handler(error));
}
```

≡ 10

Arithmetic, Relational, and Logical Functions   *11*

This chapter discusses the XIL library's arithmetic, relational, and logical functions. These functions can be used to lighten or darken an image, to increase or decrease contrast in an image, or to produce the "negative" of an image.

This chapter has the following main sections:

Arithmetic Functions	*page 205*
Relational Functions	*page 208*
Logical Functions	*page 208*
Operations With Constants	*page 209*
Arithmetic and Logical Operations With Bit Images	*page 212*

Note – All the functions discussed in this chapter can be performed in place; that is, the source and destination images can be the same image.

Arithmetic Functions

If you have two source images and a destination image, the XIL library enables you to:

- Add the two source images and store the results in the destination (`xil_add()`)

- Subtract one source image from the other and store the results in the destination (`xil_subtract()`)

- Multiply the two source images and store the results in the destination (`xil_multiply()`)

- Divide one source image into the other and store the results in the destination (`xil_divide()`)

For a single source image, you can find the absolute value of pixels in the image and store the result in a destination image (`xil_absolute()`).

For each of these operations to work, the source and destination images must have the same data type: `XIL_BIT`, `XIL_BYTE`, `XIL_SHORT`, or `XIL_FLOAT`. Also, the images must have the same number of bands. The images do *not* have to have the same width and height. For an explanation of which pixels an operation would affect if the images have different sizes, see "Region of Interest" on page 49.

When you add two images, you take the value at location 0,0 in one source image, add it to the value at location 0,0 in a second source image, and write the sum at location 0,0 in a destination image. You then follow the same procedure for all other points in the images. Subtraction, multiplication, and division are handled similarly. When multiband images are involved, the arithmetic operation is performed on corresponding bands in the source images; that is, band 0 in the first source image is added to band 0 in the second source image, and so on.

This all seems very straightforward. But there are a few points you should keep in mind when using these functions:

- If the result of an operation is out of range for a particular data type, the result is not truncated, but is clamped to the minimum or maximum value for the data type.

Thus, if you're working with `XIL_BYTE` images, adding a 200 and a 200 gives a result of 255 because 400 cannot be represented in 8 bits and the greatest valid value for an `XIL_BYTE` image is 255. Likewise, if you subtract 200 from 100, the result is 0. Table 11-1 indicates the valid range of values for each XIL data type.

Table 11-1 Valid Values for Each XIL Data Type

Data Type	Lowest Value	Greatest Value
XIL_BIT	0	1
XIL_BYTE	0	255
XIL_SHORT	-32768	32767
XIL_FLOAT	min	max

- Division by 0 *is* permitted.

  If the image serving as the divisor contains 0's, you receive *one* XIL error message indicating that division by 0 occurred. However, the division operation is performed. If a 0 in an image is divided by 0, the result is 0, and if any other value is divided by 0, the result is the greatest valid value for the data type (or the lowest valid value if the numerator is a negative value of type `XIL_SHORT` or `XIL_FLOAT`).

The `xil_absolute()` function finds the absolute value of pixels in a source image and stores the result in a destination image. Since `XIL_BIT` and `XIL_BYTE` images don't have negative values, `xil_absolute()` is useful only for `XIL_SHORT` and `XIL_FLOAT` images. For each band in the image, the absolute value at location 0,0 in the source is written to location 0,0 in the destination; the same procedure is then followed for all other points in the image.

Note – On `XIL_BIT` and `XIL_BYTE` images, `xil_absolute()` is effectively `xil_copy()`.

≡ 11

Relational Functions

If you have two source images and a destination image, the XIL library enables you to:

- Find the larger of pixels in the two source images and store the results in the destination (`xil_max()`)

- Find the lesser of pixels in the two source images and store the results in the destination (`xil_min()`)

As with the arithmetic operations, `xil_max()` and `xil_min()` require the source and destination images to have the same data type and the same number of bands. The images do not have to have the same width and height.

Taking the maximum or minimum values from two images performs a band by band comparison. For example, if `src1` and `src2` images are 4-banded images, comparing the value at location 0,0 writes to location 0,0 in the destination image the maximum (or minimum) for each source band; thus, the first band at 0,0 in the destination might receive its value from `src1`, whereas the remaining three bands might receive their values from `src2`. This band by band comparison is repeated for all other points in the source images.

Note – Don't confuse these relational functions with finding the maximum or minimum pixel value within each band of a single image. To find those values, use `xil_extrema()`, discussed in "Finding the Minimum and Maximum Values in an Image" on page 246.

Logical Functions

If you have two source images and a destination image, the XIL library also enables you to:

- Take the bitwise AND of the two source images and store the results in the destination (`xil_and()`)

- Take the bitwise OR of the two source images and store the results in the destination (`xil_or()`)

- Take the bitwise XOR of the two source images and store the results in the destination (`xil_xor()`)

> **Note** – The supported data types for bitwise operators are `XIL_BIT`, `XIL_BYTE`, and `XIL_SHORT`.

As with the arithmetic and relational operations, for these operations to work, all three images must have the same data type and the same number of bands. The images do not have to have the same width and height.

The XIL library also contains a bitwise NOT operator (`xil_not()`). This function works on a single source image. It looks at the values in that image as binary values and changes all the 1's in those values to 0's, and all the 0's to 1's. The function then writes this one's complement version of the source image to the destination.

Operations With Constants

For each of the arithmetic and logical functions that operate on two source images—addition and so on—the XIL library includes a similar function that takes as input one source image and a constant. For instance, the library contains the `xil_add()` function, whose prototype is shown below.

```
void xil_add (XilImage src1, XilImage src2, XilImage dst);
```

It also contains the `xil_add_const()` function, whose prototype is shown below.

```
void xil_add_const (XilImage src1, float *constants,
    XilImage dst);
```

To add a constant to a single-band image, you declare a one-element array of type `float`, assign that element a value, and then call `xil_add_const()`. For instance, the code to add 8.0 to each value in a single-band image might look like this.

```
XilImage src1, dst;
float constants[1] = {8.0};

xil_add_const(src1, constants, dst);
```

This operation is roughly the equivalent of adding to `src1` a second source image, all of whose values are 8. The only difference between these two operations is that using a constant instead of a second source image enables you to use noninteger values in the operation. When you add a constant to an image, which has data type `XIL_BIT`, `XIL_BYTE`, or `XIL_SHORT` (no rounding for `XIL_FLOAT`), each sum is rounded and then cast to the data type of the source and destination images.

Note – As stated on page 206, if the result of an operation is out of range, the value is clamped and assigned the minimum or maximum value for the data type. It's important to realize that the operation is performed first, then the result is clamped. For example, if the operation is division and you divide a negative constant into the values for an `XIL_BYTE` image, the division is first performed. Thus, for a pixel value of 10 divided by the constant -2, the result is

   10 ∕ -2 = -5, which is clamped to 0.

Notice that clamping the -2 to 0 before the division is performed would yield 10 ∕ 0, which would result in a value of 255 in the destination image.

In operations that involve a multiband source image and a constant, the constant must actually be an array of constants, and the number of constants in the array must equal the number of bands in the image. The following code adds a different constant to each band of a 3-band image.

```
XilImage src1, dst;
float constants[3] = {8.0, 12.0, 2.0};

xil_add_const(src1, constants, dst);
```

This operation adds 8.0 to each value in band 0 of the image, 12.0 to each value in band 1, and 2.0 to each value in band 2.

For operations that are not associative—subtraction and division—the XIL library enables you to specify the constant as the first operand or the second. That is, for a subtraction, the constant can be either the minuend or the subtrahend, and for a division, it can be either the dividend or the divisor.

Table 11-2 lists and describes the arithmetic functions that can take constants.

Table 11-2 Arithmetic Operations Using a Source Image and a Constant

Function	Description
xil_add_const()	Adds an image and a constant
xil_subtract_const()	Subtracts a constant from an image
xil_subtract_from_const()	Subtracts an image from a constant
xil_multiply_const()	Multiplies an image by a constant
xil_divide_by_const()	Divides an image by a constant
xil_divide_into_const()	Divides an image into a constant

You can also perform logical operations using an image and a constant. For instance, besides the function `xil_and()`, the library contains the function `xil_and_const()`, whose prototype is shown below.

```
void xil_and_const (XilImage src1, unsigned int *constants,
    XilImage dst);
```

To take the logical AND of a constant and a single-band image, you declare a one-element array of type `unsigned int`, assign that element a value, and then call `xil_and_const()`. For instance, the code to find the logical AND of each value in an image and 8 might look like this.

```
XilImage src1, dst;
unsigned int constants[1] = {8};

xil_and_const(src1, constants, dst);
```

This operation is the equivalent of taking the logical AND of `src1` and a second source image, all of whose values are 8.

Table 11-3 lists and describes the logical functions that can take constants.

Table 11-3 Logical Operations Using a Source Image and a Constant

Function	Description
xil_and_const()	ANDs an image and a constant
xil_or_const()	ORs an image and a constant
xil_xor_const()	XORs an image and a constant

Note – The relational operations xil_max() and xil_min() always require two source images and cannot operate on one source image and one constant.

Arithmetic and Logical Operations With Bit Images

The arithmetic and logical operations discussed in this chapter *are* defined for XIL_BIT images. If you keep in mind that the values in the destination image produced by such an operation cannot fall below 0 or go above 1, the results to expect are usually obvious. For example, when you add two XIL_BIT images, the following calculations are used:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 1

Note that 1 plus 1 equals 1. Two is not a valid value for a 1-bit data element, so the result has been clamped at 1. Similarly, when you subtract one XIL_BIT image from another, 0 minus 1 equals 0 because -1 is not a valid value.

When performing division, keep in mind that 0 divided by 0 equals 0 and that a positive nonzero value divided by 0 equals the maximum valid value for the data type, in this case 1.

When you perform an arithmetic or logical operation that involves a bit image and a constant, similar rules apply. For arithmetic operations, think of the following sequence taking place:

1. Addition, subtraction, multiplication, or division is performed using a 1-bit value (0 or 1) and a floating-point value.

2. The result is rounded and clamped to 0 or 1.

3. The 0 or 1 is cast to the `XIL_BIT` data type.

For logical operations involving a constant, the constant (an `unsigned int`) is set to 1 if it is greater than 0; then the operation is performed.

≡ *11*

Geometric Functions 12 ≡

This chapter discusses the XIL library's geometric functions. Because you must pass an interpolation option as an argument to most of these functions, this chapter begins with a discussion of the available interpolation options.

This chapter has the following main sections:

Note – The functions discussed in this chapter cannot be performed in place. That is, the source and destination images for a geometric operation must be different images.

≡ *12*

Interpolation Options

Geometrically transforming images centers around the notion of *point sampling*. In point sampling, each pixel in a destination image is located with integer coordinates at a distinct point *D* in the image plane. The geometric transform *T* identifies each destination pixel with a corresponding point *S* in the source image; thus, *T* maps *D* to *S*. In general, *S* doesn't correspond to a single source pixel; that is, it doesn't have integer coordinates. Therefore, the value assigned to the pixel *D* must be computed as an estimated or *interpolated* combination of the pixel values closest to *S* in the source image.

When performing most geometric transformations, you must specify the interpolation method to be used in calculating the destination pixel values. Table 12-1 shows the interpolation options the XIL library provides, and the strings you use to request them.

Table 12-1 Types of Interpolation

Interpolation Type	String	Assigned to *D*
Nearest neighbor	`nearest`	The value of the pixel nearest *S*
Bilinear	`bilinear`	A value that's a bilinear function of the four pixels surrounding *S*
Bicubic	`bicubic`	A value that's a bicubic function of the sixteen pixels surrounding *S*
General	`general`	A value that's a separable user-defined function of the pixels in a rectangular region surrounding *S*

To see how the interpolation type can affect the result of a geometric operation, see Color Plate 1.

Note – The XIL library does not generate results in the destination for which there are no source data defined for the operation. This means that, if the interpolation kernel must read outside the source image for the source image edge, XIL won't generate a destination pixel for that particular location. The values of the destination pixel that cannot be calculated by interpolation remain as they were before the geometric operation was performed.

Nearest Neighbor Interpolation

Nearest-neighbor interpolation, sometimes called zero-order interpolation, is the fastest interpolation method because it simply assigns to point *D* in the destination image the value of the pixel nearest S in the source image (see "Interpolation Options" on page 216). Though it's a good choice when speed is important or when you want to retain the spectral distribution of the source image, nearest-neighbor interpolation can produce undesirable artifacts in the destination image, especially near edges where there may be a big change in color or gray level between two adjacent pixels. For example, smooth lines in the source image may show up as jagged lines in the destination.

Bilinear Interpolation

A routine that performs bilinear, or first-order, interpolation assigns to point *D* in the destination image a value that's a bilinear function of the four pixels surrounding *S* in the source image (see "Interpolation Options" on page 216). This interpolation type yields better results than nearest-neighbor interpolation, but can itself have an undesirable smoothing effect on an image. To alleviate this problem, you can use bicubic interpolation.

Bicubic Interpolation

A routine that performs bicubic interpolation assigns to point *D* in the destination image a value that's a bicubic function of the 16 pixels surrounding S in the source image (see "Interpolation Options" on page 216). Using bicubic interpolation preserves fine detail present in a source image, but it takes more time than the nearest-neighbor or bilinear interpolation methods.

General Interpolation

The interpolation options discussed so far base the interpolated values on relatively few pixels: nearest neighbor uses 1 pixel, bilinear uses 4 pixels, and bicubic uses 16. If these options don't provide the quality you need, you can use the general interpolation option. For example, if you're subsampling by a factor of four, bicubic interpolation would result in aliasing artifacts that can be improved by using more source pixels in the interpolation.

General interpolation lets you:

- Determine how many pixels nearest point *S* in the source image are used to calculate the interpolated pixel value of point D in the destination image (see "Interpolation Options" on page 216). If desired, you could use every pixel in the source image.

- Weight the pixels used in the calculation. By ensuring that the pixels closest to point *S* in the source image have more influence on the value assigned to *D* than pixels that are further from *S*, you can reduce the contrast between adjacent pixels in the destination image, thus providing smoother line and color transitions.

- Designate the number of pixel subsamples to use for interpolating pixel values in the destination. This effectively lets you divide into fractional locations the space between adjacent source-image pixels so you can interpolate a destination pixel's value differently, depending on which location point *S* falls in.

To use general interpolation, you must:

1. Create horizontal and vertical interpolation tables; these form the filters or *kernels* for the interpolation. The tables determine the kernel sizes, weighted pixel values, and number of subsamples used for a general interpolation.

2. Set the interpolation tables on the system-state object. The tables affect all general interpolation operations using images created from this system-state object.

3. Pass `general` as the interpolation string on any XIL function that requires an interpolation argument. This causes the interpolation tables to be used as the interpolation method.

4. Destroy the interpolation tables when they are no longer needed. This releases the memory that was allocated for them.

The sections that follow discuss these steps in more detail. Before proceeding to them, however, it's useful to consider a conceptual model of a general interpolation.

Figure 12-1 represents an interpolation kernel as a 5-by-5 matrix of weighted values; for simplicity, the values are labelled `a` through `y`.

Figure 12-1 Conceptual Model of a General Interpolation

During a general interpolation, the kernel's key value—generally its center value—is laid over the source-image pixel to be processed; this means the other kernel values lie over neighboring pixels. Each source-image pixel that is covered by the kernel is then multiplied by the kernel value that lies over it. In Figure 12-1, the source-image pixel 2,2 is multiplied by kernel value m; pixel 0,0 is multiplied by kernel value a; pixel 1,0 is multiplied by b; and so on. The multiplication products are then summed together, and this sum becomes the pixel value in the destination.

Though Figure 12-1 provides a useful conceptual model, the general interpolation method doesn't actually construct the two-dimensional kernel shown in the figure because it's computationally expensive. To improve efficiency, general interpolation uses separate horizontal and vertical vector arrays—the kernels in the interpolation tables you create—to calculate the same values a two-dimensional kernel would calculate. The vector arrays require you to provide fewer data elements for the kernel values; this reduction is particularly significant for large tables with many subsamples. Nonetheless, for a horizontal table with M elements and a vertical table with N elements, the number of pixels that contribute to the interpolated value is still given by M * N.

Note – To translate the horizontal and vertical kernels into a two-dimensional matrix as shown in Figure 12-1, the 1xn kernel values would have to be matrix-multiplied to obtain the corresponding two-dimensional kernel values.

When interpolating pixels that are at, or near, the source image edges, the general interpolation method temporarily reduces the kernel height and width so the edges fit within the defined kernel size. This strategy lets the method interpolate pixels that originally formed the image edges. Pixels outside of the source image don't contribute to any interpolation values.

Creating Vertical and Horizontal Interpolation Tables

To support general interpolation, XIL has an `XilInterpolationTable` object, which is an array of 1xn kernels representing either a horizontal or vertical interpolation filter. General interpolation can be performed using two interpolation tables, one to represent the horizontal filter and one to represent the vertical filter. Or it can be performed using a single interpolation table that represents either the horizontal or the vertical filter, in which case the missing matrix dimension is 1. For example, if the horizontal table defines 1-by-7 kernels and the vertical table defines 1-by-5 kernels, the effective matrix is 7-by-5; if only the 1-by-7 horizontal table is defined, the effective matrix is 7-by-1. If both the horizontal and vertical interpolation tables are `NULL`, nearest-neighbor interpolation is performed.

To create an `XilInterpolationTable` object, you call the `xil_interpolation_table_create()` function, whose prototype is shown here.

```
XilInterpolationTable xil_interpolation_table_create(
      XilSystemState state, unsigned int kernel_size,
      unsigned int subsamples, float* data);
```

The table's data type is `XIL_FLOAT`. The `state` parameter is the handle to the system-state object created when you initialized the XIL library. The `kernel_size` parameter specifies the number of elements in each kernel. The `subsamples` parameter indicates the number of subsamples or fractional locations between source-image pixels; each subsample requires its own kernel data. The `data` parameter specifies the data values for each kernel. There is no limit or restriction on the `kernel_size` or number of `subsamples`.

The following code fragment creates a horizontal and a vertical interpolation table. The horizontal table has seven kernel elements, and the vertical table has three; each table has only one subsample.

```
XilSystemState state;
XilInterpolationTable horizTable;
XilInterpolationTable vertTable;

int subsamples = 1,
    horizSize  = 7,
    vertSize   = 3;

float horizData[] = {.005, .005, .04, .9, .04, .005, .005};
float  vertData[] = {.12, .8, .08};

horizTable = xil_interpolation_table_create(state,
                 horizSize, subsamples, horizData);

vertTable = xil_interpolation_table_create(state,
                 vertSize, subsamples, vertData);
```

Kernel Size

The `xil_interpolation_table_create()` function's `kernel_size` parameter determines the number of data elements in each subsample of the kernel defined by the interpolation table. There is no limit on the kernel size. The horizontal and vertical tables can have different kernel sizes, as well as different subsampling values.

The key element in a kernel is its center element; for an even-numbered kernel size, the key element is offset to the next-lowest index element. Thus, the key element for a 5-element table is the center element, which has array index 2; the key element for an 8-element table is the fourth element, which has index 3. The key element's array index can be computed as an integer calculation:

```
int array_index = (kernel_size - 1) / 2
```

Subsamples

The `xil_interpolation_table_create()` function's `subsamples` parameter determines the number of subsamples used for a general interpolation. Increasing the number of subsamples lets you specify different data values to be used for the interpolation kernel, depending on where point

S in the source image falls between pixels. There is still only one interpolation performed to determine each destination pixel value, but the weights used to interpolate the value depend on which subsample's kernel is used, and that, in turn, depends on the exact location of point *S*.

There is no limit on the number of subsamples you can use. For each subsample, you must define separate kernel data. Thus, for *n* subsamples, an interpolation table must have *n* * `kernel_size` data elements. For example, if you create a 7-element horizontal interpolation table with 4 subsamples, you must define 28 data elements. The first 7 elements define the first subsample's kernel, the second 7 elements define the second subsample's kernel, and so on.

Figure 12-2 shows how the interpolation tables are used to determine which kernel applies to a particular subsample location. In the figure, the subsampling is 4 (0-3) in both the horizontal and vertical directions.

Figure 12-2   Determining the Kernel to Use for a General Interpolation

Typically, the kernel values for each subsample are weighted according to the subsample location's proximity to the pixels used in the calculation: the closer a pixel is to the subsample location, the larger its weight value in the kernel.

Kernel Data

The kernel data for each table is an array of floating point numbers. As mentioned in "Subsamples" on page 221, for *n* subsamples, there must be *n * kernel_size* data elements. Thus, for a 3-element kernel size with 2 subsamples, you must define an array of 6 floating point numbers; the first 3 numbers form the first subsample's kernel, and the second 3 numbers form the second subsample's kernel.

> ⚠️ **Caution** – Providing too few kernel values to complete an interpolation table results in bad values.

To preserve the source image's intensity in the destination image, the sum of the data values in each interpolation kernel should equal one. Kernel values whose sum is greater than one tend to increase the destination image's intensity, and those whose sum is less than one tend to diminish the intensity. The results of applying a given kernel depend on the image it is used on.

To sharpen an image, the kernel values should be weighted heavily toward the center, with approximately 90% of the kernel value being concentrated there. To blur an image, the kernel values can be weighted away from the center.

Setting the Interpolation Tables on the System-State Object

After creating the vertical and horizontal tables, you must set them on the system-state object by calling the `xil_state_set_interpolation_tables()` function, whose prototype is shown here.

```
void xil_state_set_interpolation_tables(
    XilSystemState state,
    XilInterpolationTable horiz_kernel,
    XilInterpolationTable vertical_kernel);
```

The `state` parameter is a system-state data structure; the `horiz_kernel` and `vertical_kernel` parameters are the horizontal and vertical interpolation tables you created for general interpolation (see "Creating Vertical and Horizontal Interpolation Tables" on page 220).

The following code fragment creates horizontal and vertical interpolation tables, sets them on the system-state object, and uses them to perform a general interpolation for translating an image (see "Translating Images" on page 226).

```
XilSystemState state;
XilInterpolationTable horizTable;
XilInterpolationTable vertTable;
XilImage src, dst;

int subsamples = 8,   /* subsample and kernel sizes */
    horizSize  = 2,
    vertSize   = 2;

float kernelData[] = {1.0,  0.0,
                       .875, .125,    /* 7/8,  1/8 */
                       .75,  .25,     /* 6/8,  2/8 */
                       .625, .375,    /* 5/8,  3/8 */
                       .5,   .5,      /* 4/8,  4/8 */
                       .375, .625,    /* 3/8,  5/8 */
                       .25,  .75,     /* 2/8,  6/8 */
                       .125, .875 };  /* 1/8,  7/8 */

horizTable = xil_interpolation_table_create(state,
                  horizSize, subsamples, kernelData);

vertTable = xil_interpolation_table_create(state,
                  vertSize, subsamples, kernelData);

xil_state_set_interpolation_tables(
    state, horizTable, vertTable);

xil_translate(src, dst, "general", 50.0, 50.0);
```

In this example, the general interpolation effectively performs a bilinear interpolation with 8 subsamples. The kernel values indicate how much influence the source-image pixels have on the destination value; the kernel value 1 indicates that a source pixel completely determines the value for the destination pixel, and 0 indicates a source pixel has no influence on the destination value. If S is a source-image point to which a given destination pixel D maps (S and D are described in "Interpolation Options" on page 216), the closer S is to the source pixel used to interpolate D's value, the more

influence that source pixel has on the computation. Conversely, the further *S* is from the source pixel, the less influence the source pixel has on *D*'s computed value.

Additional Interpolation Table Functions

The XIL library also contains additional interpolation table functions. These are listed and described in Table 12-2.

Table 12-2 Additional Interpolation Table Functions

Function	Description
`xil_interpolation_table_get_kernel_size()`	Gets the kernel size of the subsample kernels in an interpolation table object
`xil_interpolation_table_get_subsamples()`	Gets the number of subsample kernels in an interpolation table object
`xil_interpolation_table_get_data()`	Gets the data in an interpolation table object
`xil_state_get_interpolation_tables()`	Gets an interpolation table
`xil_interpolation_table_get_values()`	Gets the values stored in an interpolation table object
`xil_interpolation_table_create_copy()`	Creates and returns a copy of the specified interpolation table object

Note – You should use the `xil_interpolation_table_get_values()` function rather than `xil_interpolation_table_get_data()` in new applications that you write. `xil_interpolation_table_get_data()` requires that you allocate memory for, and provide the address of, the floating point data array.

Destroying an Interpolation Table

After performing a general interpolation, you should destroy the vertical and horizontal interpolation tables if you aren't going to use them again. Destroying the tables releases the memory that was allocated to them.

To destroy an interpolation table, call the `xil_interpolation_table_destroy()` function, whose prototype is shown here.

```
void xil_interpolation_table_destroy(
     XilInterpolationTable table);
```

The only parameter to this function is a handle to the kernel you want to destroy.

Translating Images

Translating an image means moving it up, down, left, or right. The XIL function you use to translate an image is called `xil_translate()`. The prototype for this function is shown here.

```
void xil_translate(XilImage src, XilImage dst,
    char *interpolation, float xoffset, float yoffset);
```

The `src` and `dst` parameters are handles to the source and destination images. The `interpolation` parameter is a string that specifies an interpolation type. For information on this parameter, see "Interpolation Options" on page 216. The `xoffset` and `yoffset` parameters are floating-point numbers that represent the number of pixels the image should be moved horizontally and vertically. If `xoffset` is positive, the image is moved to the right, and if it is negative, the image is moved to the left. If `yoffset` is positive, the image is moved down, and if it is negative, the image is moved up.

If you have already read "Origin" on page 47, which discusses image origins, you may think that translating an image is equivalent to setting the destination image's origin and then copying the source image to the destination. For the most part, this is true. That is, the following two code fragments have the same effect.

```
XilImage src, dst;

xil_set_origin(dst, 50.0, 50.0);
xil_copy(src, dst);
```

```
XilImage src, dst;

xil_translate(src, dst, "nearest", 50.0, 50.0);
```

Both fragments move the source image 50 pixels to the right and 50 pixels down.

However, what if the 50.0's in the fragments above are changed to 49.5's? In the first case, the coordinates that make up the destination image's origin (49.5, 49.5) are rounded to integers (50, 50) before the copy occurs, so the result does not change. In the second case, though, `xil_translate()` actually moves the image 49.5 pixels down and 49.5 pixels to the right so that pixels in the destination image map to noninteger coordinates in the source image. Since there is no longer a one-to-one correspondence between the pixels in the source and destination images, the values of the pixels in the destination image must be interpolated. The different types of interpolation the XIL library provides are discussed in "Interpolation Options" on page 216.

Scaling and Subsampling Images

You can change the width and height of an image with these XIL library functions:

- `xil_scale()`
- `xil_subsample_adaptive()`
- `xil_subsample_binary_to_gray()`

The first function, `xil_scale()`, is the most general of the routines: it increases or decreases either the width or height of an image. The second, `xil_subsample_adaptive()`, is useful only for reducing an image's size; it is not useful for enlarging an image. It is available for reducing images because it often produces better results than using `xil_scale()` to reduce images. The last function, `xil_subsample_binary_to_gray()` is a special function used only for reducing the size of `XIL_BIT` images.

xil_scale()

The XIL library's general scaling routine is `xil_scale()`, whose function prototype is as follows:

```
void xil_scale(XilImage src, XilImage dst, char *interpolation,
    float xscale, float yscale);
```

The `src` and `dst` parameters are handles to the image to be scaled and to the destination image, respectively. The `interpolation` parameter is a string that specifies the type of interpolation to be used for the operation: nearest neighbor, bilinear, bicubic, or general. (For information about these interpolation methods and how to request one of them, see "Interpolation Options" on page 216.) The `xscale` and `yscale` parameters are the horizontal and vertical scale factors. If `xscale` is greater than 1, the width of the source image increases, and if it is between 0 and 1 (exclusive), the width decreases. Similarly, if `yscale` is greater than 1, the height of the source image increases, and if it is between 0 and 1 (exclusive), the height decreases.

When scaling images, image origins can affect the scaling operation. (For information about image origins, see "Origin" on page 47.) For instance, assume the source and destination images are the same size and that you want

to scale the source image by a factor of 2 both horizontally and vertically. Figure 12-3 shows what happens if, when you scale the image, the origins of both the source and destination images are set to 0.0, 0.0 (the default).

Source image   Destination image

Figure 12-3 Zooming the Upper-Left Corner of an Image

The image on the left is the original image, and the image on the right is the scaled image. Because the origins are in the upper-left corner of the images, the destination image contains a zoomed version of the upper-left quadrant of the source image.

If, on the other hand, the origins of the source and destination images are set to the pixels at the centers of the images, the scale operation produces the results shown in Figure 12-4.

Source image   Destination image

Figure 12-4 Zooming the Center of an Image

This time the destination image contains a zoomed version of a block taken from the center of the source image.

xil_subsample_adaptive()

The `xil_scale()` function discussed in the last section is a convenient way to enlarge images. You can reduce an image's size using either `xil_scale()` or `xil_subsample_adaptive()`.

The XIL library includes the second function because when you scale down an image using `xil_scale()`, you must request one of the interpolation methods discussed earlier: nearest neighbor, bilinear, bicubic, or general. The first three of these interpolation methods look at relatively few source-image pixels when calculating the value of a destination pixel. Therefore, if you use one of these options on `xil_scale()` to reduce the size of an image, there are many cases where some pixels in the source image make no contribution to the value of a pixel in the destination. For example, say that you scale down a 512-by-512 image using an *x* scale factor of .25, a *y* scale factor of .25, and bilinear interpolation. The source image contains 262,144 pixels, and the part of the destination image that represents the scaled-down source image contains 16,384 pixels. Since the operation used bilinear interpolation, the maximum

number of source-image pixels that could have contributed to the scaled-down image is 4 times 16,384, or 65,536. In other words, at least three-fourths of the pixels in the source image had no effect on the values in the scaled-down image.

To increase the number of source-image pixels used in the calculation with `xil_scale()`, you could specify the general interpolation method and define your own interpolation tables. However, the `xil_subsample_adaptive()` function circumvents this problem for you because it guarantees that no matter what scale factors you use, every pixel in the source image contributes to the value of one (and only one) pixel in the scaled-down image. For this reason, scaling down images using `xil_subsample_adaptive()` is generally preferable to using `xil_scale()`.

The function prototype for `xil_subsample_adaptive()` is shown here.

```
void xil_subsample_adaptive(XilImage src, XilImage dst,
    float xscale, float yscale);
```

The parameters to this function look much like those used by the `xil_scale()` function. Note, however, that you don't specify an interpolation type. The reason for this is that `xil_subsample_adaptive()` has an interpolation scheme built in.

Because `xil_subsample_adaptive()` is useful only for reducing the size of images, `xscale` and `yscale` must be less than or equal to 1.

xil_subsample_binary_to_gray()

The `xil_subsample_binary_to_gray()` function is similar to `xil_subsample_adaptive()` in that you use it to scale down an image and it guarantees that all the pixels in the source image make a contribution to the value of one pixel in the scaled-down image. However, `xil_subsample_binary_to_gray()` is designed specifically for scaling down `XIL_BIT` images.

The function prototype for `xil_subsample_binary_to_gray()` is shown here.

```
void xil_subsample_binary_to_gray(XilImage src, XilImage dst,
    float xscale, float yscale);
```

This specialized function is necessary because if you scale down a 1-bit image using `xil_subsample_adaptive()` or `xil_scale()`, no matter how many source pixels you consider in determining the value of a destination-image pixel, the destination pixel can only have one of two values: 0 or 1. Obviously, a great deal of information about the image can be lost in this way.

The `xil_subsample_binary_to_gray()` function helps solve this problem by allowing up to 256 gray levels in the destination image, which must have the `XIL_BYTE` data type. For defining gray levels, one colormap index is used for each possible gray level that could result from the *x* and *y* scaling factors. The colormap indexes are consecutive values, with 0 representing all 0's in the source image. You must modify your colormap to define a gray level for each resulting index.

For example, Figure 12-5 shows a source image being scaled down by a factor of .5 in both the *x* and *y* dimensions. This means that a block of four pixels in the source image determines the value of each pixel in the destination. Since there are five possible combinations of values that can appear in a 2-by-2 block of pixels in the source, there are five possible gray levels in the destination. The equations below indicate how the subsampling is performed:

- Zero 1's in the source (all 0's) = 0 in the destination
- One 1 in the source = 1 in the destination
- Two 1's in the source = 2 in the destination
- Three 1's in the source = 3 in the destination
- Four 1's in the source = 4 in the destination

0	0	0	1	0	1	0	1
0	0	0	0	0	1	1	1
1	1	0	0	0	0	0	0
1	1	0	0	1	1	0	0
0	0	1	0	0	1	1	0
0	0	0	1	0	0	1	1
0	0	0	0	1	0	0	1
0	0	0	0	0	1	1	1

0	1	2	3
4	0	2	0
0	2	1	3
0	0	2	3

XIL_BYTE destination image

XIL_BIT source image

Figure 12-5 Subsampling Bit Images

If the scaling factors require a fractional block of pixels in the source to determine the destination pixel values, the block size is rounded up. For example, if a 2.2-by-2.2 block of pixels would be required to determine pixel values in the destination, a 3-by-3 block is used, resulting in 10 possible gray levels and therefore 10 colormap indexes whose values are 0 through 9.

The result of this type of operation is much clearer than the image obtained by scaling down a 1-bit image using `xil_subsample_adaptive()` or `xil_scale()`.

Rotating Images

The XIL library's rotation function is called `xil_rotate()`. This function's prototype is shown below.

```
void xil_rotate(XilImage src, XilImage dst, char *interpolation,
     float angle);
```

The `src` parameter is a handle to the image to be rotated, and `dst` is a handle to the destination image. The `interpolation` parameter is a string that specifies the type of interpolation to be used for the operation: nearest neighbor, bilinear, bicubic, or general. For information about these interpolation methods and how to request one of them, see "Interpolation Options" on page 216. The last parameter, `angle`, indicates the desired angle of rotation in radians. (One radian is approximately equal to 57.2958 degrees.) A positive angle results in a counterclockwise rotation, and a negative angle results in a clockwise rotation.

Note – XIL does not touch pixels that are not written by the source.

When you're rotating an image, `xil_rotate()` rotates an image around its origin. (For information about image origins, see "Origin" on page 47.) For example, consider the situation where you have not set the origin of the source or destination image. Figure 12-6 shows what happens when you rotate the source image by 90 degrees counterclockwise.

Source image   Destination image

Figure 12-6   Rotating an Image Around Its Default Origin

Because you're rotating the image about its default origin, which is in the upper-left corner, the image is rotated out of sight. Only the pixels in the leftmost column of the source image are written to the destination. They become part of the topmost scanline in that image. XIL does not touch pixels in the destination that are not written by the source.

To rotate the source image in the usual sense, you must first set the origins of the source and destination images to their respective centers.

```
unsigned int width, height;

width = xil_get_width(src);
height = xil_get_height(src);
xil_set_origin(src, (width-1.0)/2.0, (height-1.0)/2.0);
xil_set_origin(dst, (width-1.0)/2.0, (height-1.0)/2.0);
xil_rotate(src, dst, "nearest", 1.5708);
```

This code produces the results shown in Figure 12-7.

Figure 12-7 Rotating an Image Around Its Center

Performing General Affine Transforms

An *affine transform* is a transformation of an image in which straight lines remain straight and parallel lines remain parallel, but the distance between lines and the angles between lines may change. Thus, translation, scaling, and rotation are all affine transforms.

The XIL functions that perform *one* of these transforms are discussed in "Translating Images," "Scaling and Subsampling Images," and "Rotating Images." However, the XIL library also contains a function that can perform *any* of these transforms, or a combination of them, such as a rotate and a scale. This function is called `xil_affine()`, and its prototype is shown below.

```
void xil_affine(XilImage src, XilImage dst, char *interpolation,
    float *matrix);
```

The `src` parameter is a handle to the image to be transformed, and `dst` is a handle to the destination image. The `interpolation` parameter is a string that specifies the type of interpolation to be used for the operation. For information about the available methods of interpolation and how to request one of them, see "Interpolation Options" on page 216. The last parameter, `matrix`, must be a six-element array of floating-point numbers.

If we call the six elements in the array a, b, c, d, tx, and ty, the equations used in performing the transform look like this:

xd = (a * xs) + (c * ys) + tx
yd = (b * xs) + (d * ys) + ty

Or written as a matrix operation:

$$\begin{bmatrix} xd \\ yd \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} xs \\ ys \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

where (xs, ys) are the source image coordinates and (xd, yd), the destination image coordinates.

To illustrate how these equations relate to particular transforms, look at how you would perform the translation, scale, and rotation considered earlier in the chapter using `xil_affine()`.

This code fragment illustrates an image being translated 50 pixels to the right and 50 pixels down.

```
XilImage src, dst;
float matrix[6] = {1.0, 0.0, 0.0, 1.0, 50.0, 50.0};

xil_affine(src, dst, "nearest", matrix);
```

The fifth value in the matrix is used to control the horizontal translation of the image, and the sixth is used to control the amount of vertical translation.

This code fragment shows an image being zoomed by a factor of 2 in both the *x* and *y* directions:

```
XilImage src, dst;
float matrix[6] = {2.0, 0.0, 0.0, 2.0, 0.0, 0.0};

xil_affine(src, dst, "nearest", matrix);
```

And this code fragment shows an image being rotated 90 degrees counterclockwise.

```
XilImage src, dst;
float matrix[6];

matrix[0] = (float)cos(1.5708);
matrix[1] = (float)-sin(1.5708);
matrix[2] = (float)sin(1.5708);
matrix[3] = (float)cos(1.5708);
matrix[4] = 0.0;
matrix[5] = 0.0;
xil_affine(src, dst, "nearest", matrix);
```

You can also use `xil_affine()` to perform many other affine transforms. For example, the code below shears a source image along the *x* axis and then translates the image to the left so that it is centered in the destination.

```
XilImage src, dst;
float matrix[6] = {1.0, 0.0, -0.4, 1.0, 50.0, 0.0};

xil_affine(src, dst, "nearest", matrix);
```

Note – In this case, the *y* coordinate of a pixel in the source image affects the *x* coordinate of a pixel in the destination. Figure 12-8 shows an image before and after this shearing operation.

Source image   Destination image

Figure 12-8 Shearing an Image Along Its *x* Axis

Warping Images

Each of the geometric functions discussed so far in this chapter performs some type of affine transformation: `xil_translate()` moves an image up, down, left, or right; `xil_scale()` changes an image's width or height; and so on. The XIL library also contains a set of general functions that let you customize a transformation to meet an individual set of needs by creating *warp* tables that perform pixelwise displacement horizontally, vertically, or in both directions. The warp tables are XIL images whose pixel values define the backward

mapping from a pixel in the destination to a pixel in the source. Note that this differs from affine transformation wherein the supplied coefficients define the forward mapping.

Warp tables are typically used to stretch an image according to predefined rules and are most useful for performing nonlinear transformations. For example, they might be used to correct distortions that were imposed on an image by the equipment used to capture it. Or they might be used for cartographic projection of an image.

To displace pixels in a single dimension, use either the `xil_tablewarp_horizontal()` or `xil_tablewarp_vertical()` function.

```
void xil_tablewarp_horizontal(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

```
void xil_tablewarp_vertical(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

To displace pixels in two dimensions, use the `xil_tablewarp()` function.

```
void xil_tablewarp(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

`xil_tablewarp_horizontal()` specifies a horizontal displacement, `xil_tablewarp_vertical()` specifies a vertical displacement, and `xil_tablewarp()` specifies a displacement in both directions. For these functions, the `src` and `dst` parameters are handles to the source and destination images. The `interpolation` parameter is a string that specifies an interpolation type for the operation: nearest neighbor, bilinear, bicubic, or general (see "Interpolation Options" on page 216). The `warp_table` parameter is a handle to a defined warp table. For a one-dimensional displacement, the warp table is a 1-banded image that specifies the displacement in one direction. For a two-dimensional displacement, the warp table is a 2-banded image; the first band specifies the horizontal displacement and the second band specifies the vertical displacement.

A warp table is applied at the destination image's origin; the source image's origin is then added to the backward mapping position specified by the warp table. By default, an image's origin is 0,0; to change the origin, use the `xil_set_origin()` function.

A warp table must have either the `XIL_SHORT` or `XIL_FLOAT` data type, though it can be used to warp images of any data type. In the case of an `XIL_SHORT` warp table, each `XIL_SHORT` value is interpreted as fixed point data with a 12-bit value and 4 bits of fractional precision. Both the source and destination images must have the same type and number of bands. The tablewarp operations cannot be performed in place.

Warp tables can be used to perform any type of geometric transformation. For example, as discussed in "Translating Images" on page 226, you can move a source image up, down, left, or right with `xil_translate()`. You can also use a warp table to move the image. The following code fragment shows how you could define a two-dimensional warp table to move an image to the left 100 pixels and up 500 pixels. Note that the source of the translation is opposite to that of `xil_translate()`: the tablewarp operation defines a backward mapping, whereas the affine operation defines a forward mapping.

```
XilSystemState state;
XilImage src, dst, warp_table;
float values[2];
int width, height;

warp_table = xil_create(state, width, height, 2, XIL_SHORT);
/* multiply offsets by 16 because of 12 bit values with
 * 4 bit precision */
values[0] = 100.0 * 16;
values[1] = 500.0 * 16;
xil_set_value(warp_table, values);

xil_tablewarp(src, dst, "bilinear", warp_table);
```

In this example, the `warp_table` variable stores the warp table for the operation; it is assumed that the source image's width and height have already been stored in the `width` and `height` variables. The `warp_table` image has two bands; the first band is needed to store the horizontal displacement defined by `values[0]`, and the second is needed to store the vertical displacement defined by `values[1]`. The call to `xil_set_values()` sets all

pixels in the table-warp image to the values defined by the `values` array; thus, the displacement is identical for all pixels. The call to `xil_tablewarp()` then applies the warp table to the source image, using a bilinear interpolation.

Of course, you wouldn't use a warp table to perform a simple translation; warp tables are better used to define algorithms that implement nonlinear displacements. The algorithm would typically define a pixelwise displacement for source-image pixels, in which case you would set the displacements on the warp table by calling `xil_set_pixel()` rather than `xil_set_value()`. For a discussion of warping transformations, see Gonzalez and Wintz, *Digital Image Processing* (see Appendix E, "Bibliography").

Transposing Images

The XIL library's transposition function, `xil_transpose()`, enables you to:

- Flip an image across an imaginary horizontal line that runs through the center of the image

- Flip an image across an imaginary vertical line that runs through the center of the image

- Flip an image across its main diagonal

- Flip an image across its antidiagonal

- Rotate an image counterclockwise about its center by 90, 180, or 270 degrees

The function's prototype is shown here.

```
void xil_transpose(XilImage src, XilImage dst,
    XilFlipType fliptype);
```

12

The `src` and `dst` parameters are handles to the source and destination images. The `fliptype` parameter is an enumeration constant of type `XilFlipType`. The constants in the enumeration are listed and described in Table 12-3.

Table 12-3 Constants in the Enumeration `XilFlipType`

Constant	Description
`XIL_FLIP_X_AXIS`	Flips the image across a horizontal line running through its center
`XIL_FLIP_Y_AXIS`	Flips the image across a vertical line running through its center
`XIL_FLIP_MAIN_DIAGONAL`	Flips the image across its main diagonal
`XIL_FLIP_ANTIDIAGONAL`	Flips the image across its antidiagonal
`XIL_FLIP_90`	Rotates the image 90 degrees counterclockwise about its center
`XIL_FLIP_180`	Rotates the image 180 degrees counterclockwise about its center
`XIL_FLIP_270`	Rotates the image 270 degrees counterclockwise about its center

Note – You don't specify an interpolation type when you call `xil_transpose()`. This is not necessary because `xil_transpose()` differs slightly from the other geometric operators in that it maps pixels in the source image directly to pixels in the destination. Also, `xil_transpose()` ignores image origins when it flips or rotates images. The function always flips an image across a line that passes through the center of the image and always rotates an image around its center.

Figure 12-9 illustrates several of the operations that `xil_transpose()` can
perform.

Original image   Flipped across vertical axis

Flipped across horizontal axis   Rotated 180 degrees counterclockwise

Figure 12-9 Flipping and Rotating Images Using `xil_transpose()`

≡ *12*

Miscellaneous Image Processing Functions 13≡

Earlier chapters in this book discussed several classes of XIL image-processing functions:

- Chapter 9, "Presentation Functions," discussed the XIL functions to use in preparing an image for display.
- Chapter 11, "Arithmetic, Relational, and Logical Functions," considered the XIL library's arithmetic and logical functions.
- Chapter 12, "Geometric Functions," discussed the library's geometric functions.

This chapter covers the library's remaining image-processing functions. The chapter has the following main sections:

Finding the Minimum and Maximum Values in an Image

The `xil_extrema()` function finds the minimum and maximum values in a single-band image or the minimum and maximum values in each band of a multiband image. The prototype for this function is as follows:

```
void xil_extrema(XilImage src, float *max, float *min);
```

The `src` parameter is a handle to a source image. The `max` parameter is an array of `float`s in which the function returns the maximum value found in each band of the source image. The `min` parameter is an array of `float`s in which the function returns the minimum value found in each band of the image. The number of elements in the `max` and `min` arrays must match the number of bands in the image.

Consider the example shown here.

```
XilImage src;
float max[3], min[3];

xil_extrema(src, max, min);
```

After this call, `max[0]` contains the maximum value in band 0 of `src`, `max[1]` contains the maximum value in band 1, and `max[2]` contains the maximum value in band 2. Similarly, `min` contains the minimum values in the three bands of the image.

Note – `xil_extrema()` returns a minimum or maximum value, depending on which one you specify. Don't confuse this with finding the larger or lesser of pixels between two source images and storing the results in a destination image. To perform that relational function, use `xil_max()` or `xil_min()`, discussed on page 208.

Producing a Histogram for an Image

The XIL library defines a data type called `XilHistogram`. An object of this type is designed to hold information on the frequency of occurrence of gray levels (or color-levels) in an image. This section discusses how to produce a histogram for an image and how to read intensity-level information from an XIL histogram structure. The primary tasks you need to perform are listed below:

- Create a histogram data structure.

- Collect gray- or color-level frequency information on an image in the data structure.

- Read the histogram data stored in the structure.

- Destroy the histogram.

These tasks are discussed in detail in the sections that follow.

Creating a Histogram

An XIL histogram structure contains a set of *bins* for each band of the image whose histogram you want to read. These bins are used to hold information on gray or color levels. For instance, if you want to take the histogram of an 8-bit grayscale image, you might create a histogram structure that contains 256 bins, where the number of 0's in the image is stored in bin 0, the number of 1's in the image is stored in bin 1, and so on up to bin 255.

Your histogram need not contain a bin for each possible value in the image. You can specify the lowest and highest values that results in a bin count being incremented. In addition, you can specify that the histogram structure contain a number of bins less than the number of levels you are checking for. In this case, each bin holds a count for a range of values. For example, if you create a histogram for an 8-bit grayscale image and specify four bins, occurrences of the values 0 to 63 are stored in bin 0, occurrences of 64 to 127 are stored in bin 1, and so on.

Note that for multi-band images, the histogram produced is, in fact, a joint histogram across all bands of the image. A joint histogram holds information on the frequency of occurrence of *combinations* of color levels. If, for example,

there are 4 bins per band, the joint histogram contains 4*4*4=64 bins. Each bin corresponds to a frequency p(v_1, v_2, v_3) giving the total number of pixels or having band 1, 2, and 3 values equal to v_1, v_2, and v_3, respectively.

You create a histogram data structure by calling the `xil_histogram_create()` function, whose function prototype is shown below.

```
XilHistogram xil_histogram_create(XilSystemState state,
    unsigned int nbands, unsigned int *nbins, float *low_value,
    float *high_value);
```

Table 13-1 lists and describes each of the parameters.

Table 13-1 Parameters to `xil_histogram_create()`

Parameter	Description
state	The system-state data structure returned when you initialized the library.
nbands	The number of bands in the histogram. This number must match the number of bands in the image whose histogram you want to take.
nbins	An array of `unsigned int`s, each element of which specifies the number of bins to be used for one band of the image. The number of elements in the array must match the number of bands in the image.
low_value	A pointer to an array of `float`s, each element of which is the midpoint of the first bin. (See "Determining low_value and high_value.") The number of elements in the array must match the number of bands in the image.
high_value	A pointer to an array of `float`s, each element of which is the midpoint of the last bin. See "Determining low_value and high_value.") The number of elements in the array must match the number of bands in the image.

Note – The total number of bins in the histogram is the product of the `nbins` values for all bands. Specifying too many bins for multi-band images could significantly degrade performance by using large amounts of memory. For example, if you create a histogram for a 3-band image with 256 bins for each

band, the histogram data structure would require approximately 16 million bins (64 MBytes). (You can generate an individual histogram for each band of an image by creating a child image.)

Determining `low_value` *and* `high_value`

The following are examples of how to determine `low_value` and `high_value` for three grayscale images.

First Image

8-bit grayscale, `XIL_BYTE`, range: `0` to `255`

4 bins: `[0, 63][64, 127] [128, 191] [192, 255]`

`low_value` = 32

`high_value` = 224

Second Image

floating point grayscale, `XIL_FLOAT`, range: `0.00` to `1.00`

4 bins: `[0.00, 0.25) [0.25, 0.50) [0.50, 0.75)[0.75, 1.00)`

`low_value` = 0.125

`high_value` = 0.875

Third Image

16-bit grayscale, `XIL_SHORT` , range: `-32768` to `32767`

4 bins: `[-32768 , -16385] [-16384, -1] [0, 16383][16384, 32767]`

`low_value` = -24576

`high_value` = +24576

Writing Level Information to the Histogram Structure

Once you have created an appropriate histogram structure, you generate the histogram for an image using the `xil_histogram()` function. The prototype for this function is shown below.

```
void xil_histogram(XilImage src, XilHistogram histogram,
    unsigned int skip_x, unsigned int skip_y);
```

The `src` parameter is a handle to the image whose histogram you want to take. The `histogram` parameter is a handle to the histogram structure you created earlier. The `skip_x` and `skip_y` parameters tell `xil_histogram()` whether it should count every pixel in the source image, or whether it can skip some pixels, either horizontally or vertically.

If `skip_x` is set to 1, `xil_histogram()` counts every pixel on a scanline; if it is set to 2, the function counts every other pixel; and so on. The value of `skip_y` has an analogous effect on whether `xil_histogram()` counts every pixel in the vertical direction.

Reading Data From a Histogram

Once you've generated histogram data for an image, you'll want to read the counts stored in the bins of the histogram structure. You read this data using the `xil_histogram_get_values()` function, whose prototype is shown below.

```
void xil_histogram_get_values(XilHistogram histogram,
    unsigned int *data);
```

The `histogram` parameter is a handle to your histogram structure, and `data` is a pointer to an array of `unsigned int`s in which `xil_histogram_get_values()` places the histogram data.

The following code fragment shows how you might retrieve the histogram data for a 3-band XIL_BYTE image. This code assumes that 32 bins were used for each band of the image.

```
#define BINS 32

XilHistogram histogram;
unsigned int *data;

data = (unsigned int *)malloc(BINS * BINS * BINS *
    sizeof(unsigned int));
xil_histogram_get_values(histogram, data);
```

The array data is populated and accessed as a 'C' multi-dimensional array, where the last index is that of the highest-numbered band.

The XIL library also contains functions to retrieve other information from histogram structures, to name histograms, and to create a copy of a histogram. These are listed and described in Table 13-2.

Table 13-2 Additional Histogram Functions

Function	Description
xil_histogram_get_nbands()	Returns the number of bands in the histogram
xil_histogram_get_nbins()	Fills a user-supplied array with values representing the number of histogram bins for each histogram band
xil_histogram_get_limits()	Fills one user-supplied array with floating-point numbers that represent the low values for the histogram's bands and a second array with numbers representing the high values
xil_histogram_get_info()	Returns information on the number of bands, the number of bins, and the low and high values in the histogram
xil_histogram_set_name()	Sets the name of a histogram

Table 13-2 Additional Histogram Functions *(Continued)*

Function	Description
xil_histogram_get_name()	Returns a copy of a histogram's name
xil_histogram_get_by_name()	Returns a handle to a histogram that has the name you specify
xil_histogram_create_copy()	Creates and returns a copy of the specified histogram

Destroying a Histogram

After reading the histogram data for your image, you should destroy the histogram structure if you don't intend to use it again. Destroying the histogram frees the memory that was allocated to store it.

You destroy a histogram structure by calling the xil_histogram_destroy() function, whose prototype is shown here.

```
void xil_histogram_destroy(XilHistogram histogram);
```

The only parameter to this function is a handle to the histogram you want to destroy.

Thresholding an Image

The XIL library's xil_threshold() function provides a simple mechanism for defining the boundaries of objects that appear on a contrasting background. The function's prototype is shown here.

```
void xil_threshold(XilImage src, XilImage dst, float *lowvalue,
    float *highvalue, float *mapvalue);
```

The src and dst parameters are handles to the source and destination images for the operation. The lowvalue parameter is a pointer to an array of floating-point numbers that define the lower bound for the threshold operation for each band of src. That is, the operation affects only values greater than or equal to lowvalue[0] in band 0, only values greater than or equal to lowvalue[1] in band 1, and so on. The highvalue parameter is a pointer to

an array of `float`s that define the upper bound for the threshold operation for each band of `src`. The final parameter, `mapvalue`, is also a pointer to an array of `float`s and is used as follows. For band *n* of an image, all values in the range `lowvalue[`*n*`]` to `highvalue[`*n*`]` inclusive are set to `mapvalue[`*n*`]`.

A standard way of arriving at the optimal values for the elements of the arrays `lowvalue` and `highvalue` is to create a histogram for the image. For information on XIL histograms, see "Producing a Histogram for an Image" on page 247.

Filling an Area in an Image

The XIL library contains two fill functions: `xil_fill()` and `xil_soft_fill()`. The first function performs boundary fills, and the second performs soft fills.

`xil_fill()`

The `xil_fill()` function fills a 4-connected region of pixels with a specified color. (One pixel is 4-connected to another if it is located directly above, below, to the left of, or to the right of that pixel.) The region begins at a starting pixel (the *seed*) and grows until it encounters a boundary color or the edge of the image.

Figure 13-1 shows a fill operation in which the boundary color is black, the fill color is dark gray, and the starting point for the fill is 3,2.

Starting point

Source image   Destination image

● Boundary color

● Fill color

Figure 13-1 Boundary Fill

The function prototype for `xil_fill()` is shown below.

```
void xil_fill(XilImage src, XilImage dst, float xseed,
     float yseed, float *boundary, float *fill_color);
```

The `src` and `dst` parameters are handles to the source and destination images for the operation. These images must have the same number of bands and the same data type. The `xseed` and `yseed` parameters represent the *x* and *y* coordinates of the starting point in the source image. The `boundary` and `fill_color` parameters are arrays of floating-point numbers. Both arrays must contain a number of elements equal to the number of bands in the source and destination images. The elements of `boundary` represent the boundary color, and those in `fill_color` represent the fill color.

xil_soft_fill()

The `xil_soft_fill()` function is designed to solve the problem of filling a region that does not have a distinct boundary, usually because the outline of the region has been antialiased. It also fills regions that are shaded or contain specular reflections without losing the special effects.

Here's how `xil_soft_fill()` works. Suppose, for example, that you want to fill a red object that appears on a blue background, and the object has been antialiased so that there's a transition at the edge of the object from red to magenta to blue. You want to fill the object with green. To get started, you specify:

- The foreground color, the region to be filled: red
- The background color: blue. (There may be more than one background color.)
- A fill color: green
- A starting point (seed) known to be inside the region to be filled

The soft-fill function then determines which pixels are inside the region and sets those pixels appropriately. To be considered part of the region to be filled, a pixel must be 4-connected to a pixel inside the region and must contain some fraction of the foreground color. That is, the value of each pixel in the region can be expressed as shown here.

```
Pixel value = (fraction * foreground_color) + ((1 - fraction) *
background_color)
```

When setting the values of pixels in the destination image, `xil_soft_fill()` uses this equation.

```
Pixel value = (fraction * fill_color) + ((1 - fraction) *
background_color)
```

As a consequence, where there was a transition from red to magenta to blue in the source, there is a transition from green to cyan to blue in the destination.

The prototype for `xil_soft_fill()` is shown below.

```
void xil_soft_fill(XilImage src, XilImage dst, float xseed,
    float yseed, float *fgcolor, unsigned int num_bgcolor,
    float *bgcolor, float *fill_color);
```

Table 13-3 lists and describes each of the parameters.

Table 13-3 Parameters to `xil_soft_fill()`

Parameter	Description
src	A handle to the source image for the operation
dst	A handle to the destination image. This image must have the same number of bands and the same data type as the source image
xseed	The *x* coordinate of the starting point in the source image for the operation
yseed	The *y* coordinate of the starting point in the source image for the operation
fgcolor	An array of `float`s containing a number of elements equal to the number of bands in the source and destination images. These elements specify the current color of the region to be filled
num_bgcolor	The number of background colors in the image
bgcolor	An array of `float`s containing a number of elements equal to the number of background colors times the number of bands in the source and destination images. If your RGB source image contained two background colors—black and red—your declaration of `bgcolor` might look like this: `float bgcolor[6] = {0.0,0.0,0.0,0.0,0.0,255.0};`
fill_color	An array of `float`s containing a number of elements equal to the number of bands in the source and destination images. These elements specify the color to be used in filling the region of interest

Filtering an Image

The XIL function for filtering is called `xil_convolve()`. Depending on the convolution filter, or *kernel*, that you specify as a parameter to this function, the function can perform a variety of tasks, including:

- Sharpening images
- Blurring images
- Highlighting the edges in images

See Color Plate 2 for a few examples of how different kernels can affect a source image.

Note – To detect edges, you can also use `xil_edge_detection()`, which uses the Sobel algorithm. For information on `xil_edge_detection()`, see "Detecting Edges in an Image" on page 264.

The kernel you supply as a parameter is a two-dimensional array of weighted values that the `xil_convolve()` function uses as follows.

To calculate the value of each pixel in the destination image, the function rotates the kernel 180 degrees about its key value. Then it lays the key value—usually the value at the center of the kernel—over the source-image pixel. This means that the other values in the kernel lie on top of neighbors of the source image pixel. Figure 13-2 shows a 3-by-3 kernel being laid on top of

the source image pixel at 1,1 and its neighbors. Note that, in this case, rotating the kernel has no effect in calculating the destination pixels, as the values surrounding the kernel key are symmetric.

Figure 13-2 Convolution Operation

The convolution function then multiplies the source-image pixel as well as each pixel in the neighborhood defined by the kernel by the appropriate kernel value. Finally, it sums the products of these multiplications. (The result of this sum becomes the destination pixel value.)

Three basic steps are involved in performing this type of convolution operation using the XIL library:

- Creating the convolution kernel
- Filtering your image
- Destroying the kernel

These tasks are discussed in detail in the next several sections.

Creating a Convolution Kernel

Convolution kernels are XIL objects of type `XilKernel` and are created with the `xil_kernel_create()` function. The prototype for this function is shown here.

```
XilKernel xil_kernel_create(XilSystemState state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, float *data);
```

The `state` parameter is a system-state data structure. You received a handle to this structure when you initialized the library. The `width` and `height` parameters are the width and height of the kernel in pixels. These values are usually odd numbers so that the kernel has a center value, but need not be. Common sizes for kernels are 3-by-3 and 5-by-5. The `keyx` and `keyy` parameters define the coordinates of the key value in the kernel. These coordinates are specified with respect to the upper-left value in the kernel (0,0). As mentioned earlier, the key value is usually in the center of the kernel. The last parameter, `data`, is a pointer to the floating-point values written to the kernel.

Note – Alternately as an optimization, you can create a separable convolution kernel of type `XilKernel`. For details, see "Creating a Separable Convolution Kernel" on page 262.

The code needed to create the kernel shown in Figure 13-2 would look something like this.

```
XilKernel high_pass_filter;
XilSystemState state;
unsigned int width, height, keyx, keyy;
float data[] = {0.0, -1.0, 0.0,
                -1.0, 5.0, -1.0,
                0.0, -1.0, 0.0};

width = height = 3;
keyx = keyy = 1;
high_pass_filter = xil_kernel_create(state, width, height, keyx,
    keyy, data);
```

The kernel created here is a high-pass filter, used for sharpening an image. Two other common high-pass filters are shown in Figure 13-3.

-1.0	-1.0	-1.0
-1.0	9.0	-1.0
-1.0	-1.0	-1.0

1.0	-2.0	1.0
-2.0	5.0	-2.0
1.0	-2.0	1.0

Figure 13-3   High-Pass Filters

Figure 13-4 shows several low-pass filters. These are useful for smoothing or blurring images.

0.1111	0.1111	0.1111
0.1111	0.1111	0.1111
0.1111	0.1111	0.1111

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

0.0625	0.1250	0.0625
0.1250	0.2500	0.1250
0.0625	0.1250	0.0625

Figure 13-4   Low-Pass Filters

Filtering an Image

Once you have created the convolution kernel you want to use for your filtering operation, call `xil_convolve()` to perform the actual filtering. The prototype for this function is as follows:

```
void xil_convolve(XilImage src, XilImage dst, XilKernel kernel,
    XilEdgeCondition edge_condition);
```

The `src` and `dst` parameters are handles to a source and destination image. These two images must have the same number of bands and contain the same type of data. The `kernel` parameter is a handle to the convolution kernel you created earlier. The final parameter, `edge_condition`, is an enumeration constant that indicates how `xil_convolve()` should handle pixels at, or near, the edge of the source image. Table 13-4 lists the three constants you can use here and explains what effect they have.

Table 13-4 Handling Edges in a Convolution Operation

Value for `edge_condition`	**Effect on Convolution Operation**
`XIL_EDGE_NO_WRITE`	The pixels at the edge of the destination image are not touched (that is, they retain the values they had prior to convolution).
`XIL_EDGE_ZERO_FILL`	Pixels at the edge of the destination image are set to zero.
`XIL_EDGE_EXTEND`	The convolution operator temporarily extends the width and height of the source image by replicating the pixels at the edge of the image. This strategy enables the operator to filter the pixels that were originally at the edge of the image.

Destroying a Convolution Kernel

After filtering your image, you should destroy the convolution kernel you used for the filtering if you don't intend to use it again. Destroying this kernel frees the memory that was allocated to store it.

You destroy a convolution kernel by calling the `xil_kernel_destroy()` function, whose prototype is shown here.

```
void xil_kernel_destroy(XilKernel kernel);
```

The only parameter to this function is a handle to the kernel you want to destroy.

Additional Kernel-Related Functions

The preceding sections have discussed the most frequently used XIL functions that affect kernels. However, the library also contains the kernel-related functions listed and described in Table 13-5. These functions enable you to make a copy of a kernel, read the values of kernel attributes, and assign a name to a kernel, among other things.

Table 13-5 Utility Functions for Convolution Kernels

Function	Description
`xil_kernel_create_copy()`	Creates a copy of a kernel
`xil_kernel_get_width()`	Gets the width of a kernel in pixels
`xil_kernel_get_height()`	Gets the height of a kernel in pixels
`xil_kernel_get_key_x()`	Gets the *x* coordinate of the kernel's key value
`xil_kernel_get_key_y()`	Gets the *y* coordinate of the kernel's key value
`xil_kernel_set_name()`	Assigns a name to a kernel
`xil_kernel_get_name()`	Returns a copy of a kernel's name
`xil_kernel_get_by_name()`	Gets a handle to a kernel by specifying the name of the kernel
`xil_kernel_get_values()`	Gets the values stored internally in the kernel

Creating a Separable Convolution Kernel

As an optimization, XIL supports creating a separable convolution kernel object of type `XilKernel` with the `xil_kernel_create_separable()` function. The prototype for this function is shown here.

```
XilKernel xil_kernel_create_separable(XilSystemState State,
                                      unsigned int width,
                                      unsigned int height,
                                      unsigned int keyx,
                                      unsigned int keyy,
                                      float *x_data,
                                      float *y_data);
```

A separable convolution kernel can be expressed as 2 one-dimensional vectors whose outer product produces a 2-dimensional kernel.

Example Separable Kernel

The code to create a simple convolution kernel (5 columns by 3 rows) expressed in separable form would look something like this.

```
XilSystemState state;
XilKernel kernel;
unsigned int width, height, keyx, keyy;
float x_data[5] = { 1.0/5.0, 1.0/5.0, 1.0/5.0, 1.0/5.0, 1.0/5.0 };
float y_data[3] = { 1.0/3.0, 1.0/3.0, 1.0/3.0 };

width = 6;
height = 3;
keyx = 2;
keyy = 1;

kernel = xil_kernel_create_separable(state, width, height, keyx,
keyy, x_data, y_data);
xil_convolve(source, destination, kernel, XIL_EDGE_ZERO_FILL);
```

The resulting kernel is shown in Figure 13-5.

		x_data				
		1.0/5.0	1.0/5.0	1.0/5.0	1.0/5.0	1.0/5.0
	1.0/3.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0
y_data	1.0/3.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0
	1.0/3.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0	1.0/15.0

Figure 13-5 Separable Kernel

Advantages of Separable Kernels

Creating a separable convolution kernel has two advantages:

- It increases performance. XIL is required to make fewer calculations.

- You can streamline the code you write. Instead of providing 15 array values, the above example only requires 8.

Note – XIL tests kernels created with the `xil_kernel_create()`function to determine if they are separable. If so, it uses separable convolution routines to perform the actual colvolution calculations with such kernels, thereby enhancing performance.

Separable Kernel Functions

Once a separable kernel is created, it can use all the functions of a standard convolution kernel. For details on these functions, see the following sections:

- "Filtering an Image" on page 260
- "Destroying a Convolution Kernel" on page 261
- "Additional Kernel-Related Functions" on page 262

Detecting Edges in an Image

The `xil_edge_detection()` function detects edges in a source image and writes the result to a destination image. The prototype for this function is shown here.

```
void xil_edge_detection(XilImage src, XilImage dst,
    XilEdgeDetection edge_detection_method);
```

The `src` and `dst` parameters are handles to a source and destination image. The two images must have the same number of bands and contain the same type of data. The `edge_detection_method` parameter is an enumeration constant describing the type of edge detection to perform.

Currently, `XIL_EDGE_DETECT_SOBEL` is the only edge detection method supported; to detect vertical and horizontal edges in an image, it uses the masks shown in Figure 13-6.

Vertical			Horizontal		
-0.5	0.0	0.5	-0.5	-1.0	-0.5
-1.0	0.0	1.0	0.0	0.0	0.0
-0.5	0.0	0.5	0.5	1.0	0.5

Figure 13-6 Filters Used by the `XIL_EDGE_DETECT_SOBEL` Algorithm

The `XIL_EDGE_DETECT_SOBEL` method detects edges by finding the gradient of an image as follows:

- It performs two correlation operations on the source image: one operation detects vertical edges using the vertical mask shown in Figure 13-6, the other detects horizontal edges using the horizontal mask. This yields two intermediate images: *a* and *b*.

- It squares all the pixel values in image *a* and then in image *b*, yielding the intermediate images a^2 and b^2.

- It forms the final destination image by taking the square root of $a^2 + b^2$.

Note – The correlation operations duplicate the source-image edges. This is similar to using the `XIL_EDGE_EXTEND` edge condition on `xil_convolve()`. For information on `xil_convolve()`, see "Filtering an Image" on page 257.

Note – For a discussion of the Sobel edge condition algorithm, consult Gonzalez and Wintz, *Digital Image Processing* (see Appendix E, "Bibliography").

Dilating or Eroding an Image

The XIL library includes two functions—`xil_dilate()` and `xil_erode()`—that enable you to dilate and erode regions within images. One of the main uses of these functions is to perform *trapping* on color images that are printed. This trapping involves creating some overlap where regions of different colors meet. Once a trap is in place, slight registration problems during the printing process should not result in gaps appearing between the edge of an object and its background.

Dilation and erosion are similar to convolution in that both calculate destination-image pixels by looking at a neighborhood of source-image pixels and the values in a matrix (called a *structuring element*). Like a convolution kernel, this structuring element contains a key value. As each pixel in the destination is calculated, this key value is laid over the corresponding pixel in the source; see Figure 13-7.

Figure 13-7 Dilating and Eroding Images

Note that a structuring element contains only 0's and 1's.

For each placement of the structuring element, the dilation function calculates a destination pixel value as follows. It looks at the values of the source-image pixels that correspond to a 1 in the structuring element and assigns the maximum of these values to the appropriate destination pixel. In the case of an `XIL_BIT` image, this means that if any of the source-image pixels associated with a 1 in the structuring element is set, the destination pixel is set.

Figure 13-8 shows a binary image before and after dilation of the image's white region. The dilation was performed using the structuring element shown in Figure 13-7.

Source image   Dilated image

Figure 13-8   Dilating an Image

The erosion function does the opposite of this. It looks at the values of the source-image pixels that correspond to a 1 in the structuring element and assigns the minimum of these values to the appropriate destination pixel. Figure 13-9 shows a binary image before and after erosion of the image's white region.

Source image   Eroded image

Figure 13-9   Eroding an Image

$\equiv 13$

Performing either a dilation or an erosion in an XIL program is a three-step process:

1. Create a structuring element.

2. Perform the dilation or erosion.

3. Destroy the structuring element.

These tasks are discussed in detail in the sections below.

Creating a Structuring Element

Structuring elements, which are XIL objects of type `XilSel`, are created with the `xil_sel_create()` function. The prototype for this function is shown here.

```
XilSel xil_sel_create(XilSystemState state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, unsigned int *data);
```

The `state` parameter is a system-state data structure. You received a handle to this structure when you initialized the library. The `width` and `height` parameters are the width and height of the structuring element in pixels. These values are usually odd numbers so that the structuring element has a center value, but need not always be odd. Common sizes for structuring elements are 3-by-3 and 5-by-5. The `keyx` and `keyy` parameters define the coordinates of the key value in the kernel. These coordinates are specified with respect to the upper-left value in the structuring element (0,0). The key value is usually in the center of the structuring element. The last parameter, `data`, is a pointer to the Boolean values that are written to the kernel.

The code needed to create the structuring element shown in Figure 13-7 would look something like this.

```
XilSel cross_sel;
XilSystemState state;
unsigned int width, height, keyx, keyy;
unsigned int data[] = {0, 1, 0,
                       1, 1, 1,
                       0, 1, 0};

width = height = 3;
keyx = keyy = 1;
cross_sel = xil_sel_create(state, width, height, keyx, keyy,
    data);
```

You can also assign names to structuring elements that you've created and then get handles to those structuring elements using the `xil_sel_get_by_name()` function. To assign a name to a custom structuring element, you use the `xil_sel_set_name()` function, whose prototype is as shown here.

```
void xil_sel_set_name(XilSel sel, char *sel_name);
```

You can determine whether a structuring element has a name and, if it does, get that name by using the `xil_sel_get_name()` function.

```
char *xil_sel_get_name(XilSel sel);
```

Dilating or Eroding an Image

Once you have created a structuring element, call `xil_dilate()` to perform a dilation. This function's prototype is shown here.

```
void xil_dilate(XilImage src, XilImage dst, XilSel sel);
```

The `src` and `dst` parameters are handles to the source and destination images for the operation. These images must have the same number of bands and the same data type. The `sel` parameter is a handle to the structuring element you created earlier.

To erode a region in an image, you call `xil_erode()`, which takes the same parameters as `xil_dilate()`.

```
void xil_erode(XilImage src, XilImage dst, XilSel sel);
```

Destroying a Structuring Element

After performing your dilation or erosion, you should destroy your structuring element if you don't intend to use it again. Destroying the structuring element frees the memory that was allocated to store that structure.

To destroy a structuring element, call `xil_sel_destroy()`. This function's prototype is shown below.

```
void xil_sel_destroy(XilSel sel);
```

The only parameter to this function is a handle to the structuring element you want to destroy.

Additional Structuring-Element Functions

The preceding sections discussed the most frequently used XIL functions that affect structuring elements. However, the library also contains the structuring-element-related functions listed and described in Table 13-6. These functions enable you to make a copy of a structuring element and read the values of structuring-element attributes.

Table 13-6 Utility Functions for Structuring Elements

Function	Description
`xil_sel_create_copy()`	Returns a copy of a structuring element
`xil_sel_get_width()`	Gets the width of a structuring element
`xil_sel_get_height()`	Gets the height of a structuring element
`xil_sel_get_key_x()`	Gets the *x* coordinate of the key value of a structuring element
`xil_sel_get_key_y()`	Gets the *y* coordinate of the key value of a structuring element
`xil_sel_get_values()`	Gets the values stored internally for a structuring element

Passing an Image Through a Lookup Table

XIL lookup tables provide a very general mechanism for modifying images. A lookup table enables you to 1) convert a single-band image of any data type to a single-band or multiband image of any data type, or 2) convert a multiband image of any data type to a multiband image of any data type. Each lookup table allows you to specify precisely the mapping from source image pixel values to destination image pixel values.

The general procedure for performing a lookup operation involves three steps:

1. Create a lookup table.

2. Pass a source image through the lookup table.

3. Destroy the table.

These tasks are discussed in detail in the sections that follow.

☰ *13*

Creating a Lookup Table

The steps for creating a lookup table vary slightly, depending on whether the input image is a single-band or multiband image. For a single-band input image, you create a single lookup table; for a multiband input image, you create a single lookup table for each band in the input image, then combine those tables into a *combined* lookup table. Regardless of whether you create a single lookup table or a combined lookup table, you pass the image through the table in the same way.

Lookup tables for single-band and multiband input images are discussed separately in the following sections.

Creating Lookup Tables for Single-Band Input Images

To create a lookup table for a single-band input image, you make a single call to the `xil_lookup_create()` function, whose prototype is shown here.

```
XilLookup xil_lookup_create(XilSystemState state,
    XilDataType input_data_type, XilDataType output_data_type,
    unsigned int output_nbands, unsigned int num_entries,
    short first_entry_offset, void *data);
```

This function returns a handle to an object of type `XilLookup`, which is the lookup table. Table 13-7 lists and describes the parameters to `xil_lookup_create()`.

Table 13-7 Parameters to `xil_lookup_create()`

Parameter	Description
`state`	A handle to the system-state data structure that was created when you initialized the XIL library
`input_data_type`	The data type of the pixel values in the source image. The four possible data types are `XIL_BIT`, `XIL_BYTE`, `XIL_SHORT`, and `XIL_FLOAT`
`output_data_type`	The data type of the pixel values in the destination image
`output_nbands`	The number of bands in the destination image

Table 13-7 Parameters to `xil_lookup_create()` *(Continued)*

Parameter	Description
num_entries	The number of entries in the lookup table. Each entry specifies the correspondence between a source-image pixel value and the value, or values, that define the corresponding pixel in the destination image
first_entry_offset	The source-image pixel value for the first entry in the table. The source-image pixel value for the last entry in the table is `first_entry_offset + num_entries - 1`
data	A pointer to the data to be stored in the table

You might use this function if, for example, you had an `XIL_BIT` image that you want to convert to a three-band `XIL_BYTE` image so that you can display it on a 24-bit display. Further, you want black pixels in the source to appear dark gray in the destination image, and white pixels to appear light blue. The code to create a lookup table for this operation is shown here.

```
XilLookup lookup_table;
XilSystemState state;
Xil_unsigned8 data[] = {79, 79, 47, 216, 216, 191};

lookup_table = xil_lookup_create(state, XIL_BIT, XIL_BYTE, 3, 2,
    0, data);
```

The lookup table this code would produce is shown in Figure 13-10.

0	79	79	47
1	216	216	191

Figure 13-10 Single Lookup Table

The numbers to the left of the bold line are `XIL_BIT` values to be looked up in the source image. The numbers to the right of the line are `XIL_BYTE` values to be written to the destination image. Because the destination has three bands, the table contains three values for each pixel.

A lookup table can also be used to convert an `XIL_BIT` image to an 8-bit `XIL_BYTE` image; however, it may be easier to use `xil_cast()` to cast the `XIL_BIT` data type to `XIL_BYTE`. The `xil_cast()` function casts the values 0 and 1 in the `XIL_BIT` image to indices 0 and 1 in the `XIL_BYTE` image. If you need different indices, convert the image by passing it through a lookup table.

After creating the lookup table, you must pass the image through it, as discussed in "Passing an Image Through the Table" on page 276.

Creating Lookup Tables for Multiband Input Images

To create a lookup table for a multiband input image, call `xil_lookup_create()` once for each band in the input image, then combine these single lookup tables into a *combined* lookup table. To create the combined lookup table, you call the `xil_lookup_create_combined()` function, whose prototype is shown here.

```
XilLookup xil_lookup_create_combined(XilSystemState state,
    XilLookup lookup_list[], unsigned int num_lookups);
```

This function returns a handle to an object of type `XilLookup`, which is the combined lookup table. The `state` parameter is a handle to the system-state data structure that was created when you initialized the XIL library, `lookup_list[]` is an array of type `XilLookup` that stores the single lookup tables created for each of the input image's bands, and `num_lookups` indicates how many lookup tables are stored in the `lookup_list[]` array.

To create the combined lookup table:

1. **Create an array variable of data type** `XilLookup`**.**
   The variable should have as many array elements as there are bands in the input image so you can store each band's values in a separate element. Store the values for band 0 in element 0, those for band 1 in element 1, and so on.

2. **Call** `xil_lookup_create()` **to create a single lookup table for each band in the input image.**
Each single lookup table defines values for only one band in the destination image; thus, you must pass a 1 for the `xil_lookup_create()` function's `output_nbands` argument. The single lookup tables must all have the same data type, but each can use a different offset.

For more information, see "Creating Lookup Tables for Single-Band Input Images" on page 272.

3. **Call** `xil_lookup_create_combined()` **to combine the single tables.**
Pass it the array variable you created in Step 1.

4. **Pass the input image through the combined lookup table.**
For detailed information, see "Passing an Image Through a Lookup Table" on page 271.

This code fragment shows how you might alter a 3-band `XIL_BYTE` input image whose green band is accented but whose red and blue bands are subdued.

```
XilImage image;
XilLookup lookup_tables[3];/* var to store 3 lookup components */
XilLookup combined_lookup_table;
Xil_unsigned8 red[256];     /* red component of lookup */
Xil_unsigned8 green[256];   /* green component of lookup */
Xil_unsigned8 blue[256];    /* blue component of lookup */
int i;

for(i=0; i<256; i++) {
   green[i] = (i + 20) < 255 ? i + 20 : 255;
    blue[i] = red[i] = (i - 10) < 0 ? 0 : i - 10;
}
/* create single lookup tables for each input band */
lookup_tables[0] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                    1, 256, 0, red);
lookup_tables[1] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                    1, 256, 0, green);
lookup_tables[2] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                    1, 256, 0, blue);
/*combine the tables and pass input image through combined
table*/
combined_lookup_table = xil_lookup_create_combined(state,
                                        lookup_tables, 3);
xil_lookup(image, image, combined_lookup_table);
```

Passing an Image Through the Table

Once you've created a lookup table, you call `xil_lookup()` to perform the actual lookup operation. The prototype for this function is shown below.

```
void xil_lookup(XilImage src, XilImage dst, XilLookup lookup);
```

The `src` parameter is a handle to your source image. The data type of this image must match the data type of the values on the input side of your lookup table. The `dst` parameter is a handle to your destination image. The data type of this image must match the data type of the values on the output side of the table. For a single-band input image, the number of bands in the destination

image must match the number of values per pixel on the output side of the table; for a multiband input image, the number of bands in the destination image must match the number of lookup tables that compose the combined lookup table. The final parameter, `lookup`, is the handle to the lookup table you received when you called `xil_lookup_create()` for a single-band input image, or `xil_lookup_create_combined()` for a multiband input image.

Destroying a Lookup Table

After performing your lookup operation, you should destroy your lookup table if you won't be using it in a subsequent operation. Destroying the table frees the memory that was allocated to store it. For multiband input images, be sure to destroy the tables created for each input band, as well as the combined table.

You destroy a lookup table by calling the `xil_lookup_destroy()` function, whose prototype is shown below.

```
void xil_lookup_destroy(XilLookup lookup);
```

The only parameter to this function is a handle to the lookup table you want to destroy.

Additional Lookup-Table Functions

The preceding sections discussed the most frequently used XIL functions that affect lookup tables. However, the library also contains other functions related to lookup tables. Table 13-8 lists and describes these functions. They enable you to make a copy of a lookup table, read the values of lookup-table attributes, assign a name to a lookup table, and so on.

≡ *13*

Note – Some of the functions cannot be used on a combined lookup table. However, you can extract a band from a combined lookup table and perform those functions on the extracted copy. You can then use the altered table for creating another combined lookup table.

Table 13-8 Additional Functions for Lookup Tables

Function	Description
`xil_lookup_create_copy()`	Creates and returns a copy of an existing lookup table (LUT).
`xil_lookup_get_input_datatype()`	Gets the data type of an LUT's input.
`xil_lookup_get_output_datatype()`	Gets the data type of an LUT's output.
`xil_lookup_get_input_nbands()`	Gets the number of bands in the LUT's input.
`xil_lookup_get_output_nbands()`	Gets the number of bands defined for the LUT's output.
`xil_lookup_get_num_entries()`	Gets the number of entries in the LUT. Cannot be used on a combined LUT.
`xil_lookup_get_offset()`	Gets the input value for the first entry in the LUT. Cannot be used on a combined LUT.
`xil_lookup_set_offset()`	Sets the input value for the first entry in the LUT. Cannot be used on a combined LUT.
`xil_lookup_get_band_lookup()`	Gets a particular lookup table out of a combined LUT.
`xil_lookup_get_values()`	Retrieves the values in an LUT. Cannot be used on a combined LUT.
`xil_lookup_set_values()`	Sets the values in an LUT. Cannot be used on a combined LUT.
`xil_lookup_get_version()`	Gets a unique identifier associated with an LUT.
`xil_lookup_convert()`	Creates an LUT that includes input data from one lookup table and output data from another. Cannot be used on a combined LUT.

Table 13-8 Additional Functions for Lookup Tables *(Continued)*

Function	Description
`xil_squeeze_range()`	Creates an LUT that maps a single-band image to a single-band image with contiguous values. Cannot be used on a combined LUT.
`xil_lookup_get_name()`	Reads the name of an LUT.
`xil_lookup_set_name()`	Assigns a name to an LUT.
`xil_lookup_get_by_name()`	Gets a handle to an LUT by specifying the name of the table.

Linear Combination of Image Bands

The `xil_band_combine()` function calculates each value in a destination image by performing a linear combination (matrix multiply) of the values of all the bands of a pixel in the source image. The nonimage values used in this calculation are stored in a matrix. For example, assume that `xil_band_combine()` is operating on the three-band single-pixel image shown in Figure 13-11 using the matrix shown in the figure and is writing its output to a one-band single-pixel image.

Figure 13-11 Interband Linear Combination

The equation used to calculate the value of the destination-image pixel would be:

dst = (255 * .25) + (157 * .5) + (28 * .75) + 1.0

As you can see, the number of columns in the matrix is equal to the number of bands in the source image plus one. This number of columns provides a multiplier for each band in the source image plus a constant that is to be added to the sum of the products obtained using the multipliers and the values in the source image. The number of rows in the matrix must equal the number of bands in the destination image. If the destination image had three bands, the values in the second row of the matrix would be used in calculating the values in the second band of the destination, and the values in the third row would be used in calculating the values in the third band.

Thus, the matrix shown in Figure 13-12 would operate on a three-band source image and would produce a three-band image identical to the source.

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0

Figure 13-12 Linear Combination Matrix

Performing a Linear Combination

Performing a linear combination is a two-step process. You first create an XIL kernel (an object of type `XilKernel`) in which to store a matrix. (For details on creating kernels, see "Creating a Convolution Kernel" on page 259.) Then, you call `xil_band_combine()` to perform the operation.

To create the kernel that holds the matrix, you call the `xil_kernel_create()` function. For instance, to create a kernel that describes the matrix shown in Figure 13-12, you could use this code.

```
XilSystemState state;
XilKernel matrix;
unsigned int width = 4, height = 3, keyx = 0, keyy = 0;
float data[] = {1.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 0.0,
                0.0, 0.0, 1.0, 0.0};

matrix = xil_kernel_create(state, width, height, keyx, keyy,
    data);
```

The values of `keyx` and `keyy`, which define the key value of the kernel, are irrelevant if the kernel is used in a linear-combination operation because `xil_band_combine()` ignores these values. For more information on key values, see "Filtering an Image" on page 257.

Note – When you've finished using this kernel, you should destroy it using the `xil_kernel_destroy()` function.

Once you've created this kernel, you can call `xil_band_combine()`, whose function prototype is shown below.

```
void xil_band_combine(XilImage src, XilImage dst, XilKernel
    matrix);
```

The `src` parameter is a handle to the source image, `dst` is a handle to the destination image, and `matrix` is a handle to the XIL kernel created earlier.

How to Use Linear Combinations

The `xil_band_combine()` function is a general-purpose function that can be used in a few ways.

One possibility is to use the function to convert an image from one color space to another. For example, you could use the matrix shown in Figure 13-13 to convert a three-band `XIL_BYTE` image from the RGB to the CMY color space.

0.0	0.0	-1.0	255.0
0.0	-1.0	0.0	255.0
-1.0	0.0	0.0	255.0

Figure 13-13 RGB-to-CMY Conversion Using `xil_band_combine()`

Note – You need to be aware of the ordering of bands for multiband images of different color spaces. See "Effect of Color Space on Storage Format" on page 79 for details.

Similarly, you could use the matrix shown in Figure 13-14 to convert a three-band `XIL_BYTE` RGB image to a one-band Y image.

0.114	0.587	0.299	0.0

Figure 13-14 RGB-to-Y Conversion Using `xil_band_combine()`

However, in most cases using `xil_band_combine()` is not the best way of performing color-space conversions using the XIL library. For information on the preferred method of performing these conversions, see "Color Space Manipulation" on page 178 .

Another possibility would be to use the matrix shown in Figure 13-15 to convert a three-band source image to a single-band destination of the same data type.

0.333	0.333	0.333	0.0

Figure 13-15 Calculating the Normalized Sum of an Image

Each pixel in the destination would equal the normalized sum of the three values of the corresponding pixel in the source.

Blending Images

The XIL library's blending function, `xil_blend()`, blends two images using an alpha mask. The function's prototype is shown below.

```
void xil_blend(XilImage src1, XilImage src2, XilImage dst,
    XilImage alpha);
```

The `src1` and `src2` parameters are handles to the two images to be blended, and `dst` is a handle to the destination image. The last parameter, `alpha`, is a handle to an XIL image that serves as an alpha mask for the blending operation. This alpha image must be a single-band image.

The following equation shows how `xil_blend()` uses the values in the mask as it calculates the values of pixels in the destination image:

```
dst = (((1 - normalized-alpha) * src1) + (normalized-alpha * src2))
```

Note – A normalized value is one that has been scaled into the range 0 to 1, where 0 corresponds to the minimum value for the image's data type and 1 corresponds to the maximum value. For example, in an `XIL_BYTE` image, a value of 127 would be normalized to approximately .5 since it falls about half way between 0 and 255.

Figure 13-16 illustrates the blending of two `XIL_BYTE` images, labeled `src1` and `src2`. The pixels in the dark areas of these images have values of 100, and the white pixels have values of 255. The alpha mask, `alpha`, has all its pixels set to 127, so normalized `alpha` is approximately .5. Thus, the equation used to calculate destination-image values in this example is:

```
dst = ((.5 * src1) + (.5 * src2))
```

There are only three distinct values in the destination:

- Where `src1` and `src2` were both white, the destination value is calculated as follows:

  ```
  (.5 * 255) + (.5 * 255) = 255
  ```

  These pixels appear white in the destination.

- Where `src1` and `src2` were both dark (pixel value of 100), the destination value is calculated as follows:

  ```
  (.5 * 100) + (.5 * 100) = 100
  ```

  These pixels appear dark gray in the destination.

- Where `src1` was white and `src2` was dark, or vice versa, the destination value is calculated as follows:

  ```
  (.5 * 255) + (.5 * 100) = 178
  ```

These pixels appear light gray in the destination.

Figure 13-16 Blending Images

To see an example of the blending of two color images, see Color Plate 3.

Painting on an Image

The library's paint operation requires you to provide a list of the pixels in the source image you want to paint by specifying their coordinates. You also specify a kernel that the operation uses as a brush. This kernel is like the kernel you use with `xil_convolve()`, except that all the values in the kernel must fall in the range 0.0 to 1.0. Finally, you specify a color that you want to use for your painting.

Note – For information on creating an XIL kernel, see "Creating a Convolution Kernel" on page 259.

The paint function uses the kernel you supply much as the convolution operation does. For each pixel that you want to paint, the paint function lays the key value of the kernel over the pixel. See Figure 13-17.

Figure 13-17 Painting on an Image

However, the paint function uses the kernel differently. In a paint operation, a neighborhood of pixels in the source does not contribute to the value of a singe pixel in the destination. Instead, for each placement of the kernel, each source-image pixel that lies under the kernel affects the value of the corresponding pixel in the destination. This part of the painting operation is similar to a blending operation. The pixels under the kernel are blended with the paint color, and the kernel serves as the equivalent of an alpha mask. Destination-image pixel values are determined by the following formula:

```
dst = (brush-value * color) + ((1.0 - brush-value) * src)
```

You perform a paint operation in an XIL program by calling `xil_paint()`, whose prototype is as follows:

```
void xil_paint(XilImage src, XilImage dst, float *color,
    XilKernel brush, unsigned int count, float *coord_list);
```

Table 13-9 lists and describes each of the parameters.

Table 13-9 Parameters to `xil_paint()`

Parameter	Description
src	A handle to the source image for the operation.
dst	A handle to the destination image. This image must have the same number of bands and the same data type as the source image.
color	An array of `float`s containing a number of elements equal to the number of bands in the source and destination images. The elements in this array define the color to be used for painting.
brush	A handle to the kernel used as the brush.
count	The number of pixels to be painted.
coord_list	An array of `float`s containing `2 * count` elements. These elements represent the *x* and *y* source coordinates of the pixels to be painted.

Setting and Getting the Values of Pixels in an Image

The XIL library contains three functions that set or retrieve the values of pixels in an image. Two of these functions—`xil_set_pixel()` and `xil_get_pixel()`—set or get the value of individual pixels. The third, `xil_set_value()`, sets all the pixels in an image.

`xil_set_pixel()` *and* `xil_get_pixel()`

The prototypes for these two functions are shown here.

```
void xil_set_pixel(XilImage image, unsigned int x,
    unsigned int y, float *values);

void xil_get_pixel(XilImage image, unsigned int x,
    unsigned int y, float *values);
```

In both prototypes, the `image` parameter is a handle to the image you're working with. The parameters `x` and `y` are the *x* and *y* coordinates of the pixel you want to read or write. These coordinates should describe the location of the pixel of interest with respect to the image's origin, which by default is in the upper-left corner of the image (0,0). The final parameter, `values`, is an array of floating-point numbers. The number of elements in this array must match the number of bands in the image.

When you read a pixel, the value of band 0 of that pixel is written to `values[0]`, the value of band 1 is written to `values[1]`, and so on. Similarly, when you write a pixel, `values[0]` is written to band 0 of the pixel, and so forth. Before the writing actually takes place, the floating-point numbers in `values` are converted to integers.

Note – If you attempt to write to a pixel a value that is out of range for the source image's data type, the value is clamped to the low or high limit for the data type. For example, an `XIL_BYTE` image can only accommodate values in the range 0 to 255. If you try to write a 400 to such an image, the function actually writes a 255.

`xil_set_value()`

The `xil_set_value()` function is useful for clearing an image by setting all the pixels in the image to the same color. The prototype for this function is shown here.

```
void xil_set_value(XilImage dst, float *values);
```

The `dst` parameter is a handle to the image you are working with, and `values` is an array of floating point numbers. The number of elements in this array must match the number of bands in the image `dst`. The first element in the array is rounded to an integer and is then written to all of band 0 of your image; the second element is rounded to an integer and written to band 1 of the image; and so on.

Note – The array elements are rounded if and only if the image `dst` data type is not `XIL_FLOAT`.

Note – If you attempt to write to a band a value that is out of range for the image's data type, the value is clamped to the low or high limit for the data type. For example, an `XIL_BYTE` image can only accommodate values in the range 0 to 255. If you try to write a 400 to a band, the function actually writes a 255.

Copying a Pattern to an Image

Besides `xil_copy()`, which copies a source image to a destination image, the XIL library includes a function, `xil_copy_pattern()`, that writes as many copies of the source image as possible to the destination image. The function prototype for `xil_copy_pattern()` is shown here.

```
void xil_copy_pattern(XilImage src, XilImage dst);
```

The `src` parameter is a handle to the source image, and `dst` is a handle to the destination image.

If your destination image is the same size as or is smaller than your source image and aligning image origins results in the source image completely covering the destination, using `xil_copy()` and `xil_copy_pattern()` produce the same result. If, however, your destination image is larger than your source image or aligning origins causes any portion of the destination image not to be covered by the source, `xil_copy()` writes one copy of the source image to the destination, while `xil_copy_pattern()` writes as many copies as possible. Some of these copies are partial copies if the dimensions of the destination are not multiples of the dimensions of the source. Figure 13-18

illustrates the difference between `xil_copy()` and `xil_copy_pattern()` when using each function to copy a source image to a destination image that is twice the height and width of the source.

xil_copy()   xil_copy_pattern()

Figure 13-18 Replicating a Source Image

Compressed Image Sequences *14*

Besides enabling you to perform image processing operations on XIL images, the XIL library also gives you the ability to compress one or more XIL images and to store the compressed images in an object called a compressed image sequence (CIS).

This chapter has the following main sections:

What is an XIL Compressed Image Sequence?

An XIL *compressed image sequence* (CIS) is an object of type `XilCis` that is a container for compressed images. You can compress image data stored in a CIS in one of several formats that may represent a series of still images, frames

from a movie, or pages from a document. You can also decompress the data stored in a CIS and write your output to one or more XIL images. The compression-decompression process is shown in Figure 14-1.

Figure 14-1   Compressing and Decompressing XIL Images

Uncompressed images can take up a lot of disk space and a long time to move over a network. For example, a 512-by-512 3-band image containing `XIL_BYTE` data takes up .75 Mbytes. A compressed version of this image might be anywhere from one-half to one-fiftieth this size, depending on the type of compressor used.

Compression and Decompression Modules

The XIL library contains several compression and decompression modules:

* JPEG baseline sequential and lossless codecs
* An H.261 decompressor
* An MPEG-1 decompressor
* CCITT Group 3 and Group 4 codecs
* Cell and CellB codecs

> **Note** – The current release of the XIL library includes modules for H.261 and an MPEG-1 decompressors only. However, compressor interfaces are defined for third parties who want to implement them.

JPEG Baseline Sequential and Lossless Codecs

The JPEG compression standards were developed by the Joint Photographic Experts Group to support the compression of still images, both grayscale and color. Although not specifically designed for the compression of sequences of images, or movies, JPEG compressors are also used frequently for that purpose. A *lossy* JPEG compressor compresses an image in such a way that when the compressed data is decompressed, the decompressed image and the original image may not match exactly. On the other hand, a *lossless* JPEG compressor processes an image so that the decompressed image matches the original image pixel for pixel.

H.261 Decompressor

The H.261 compression standard was developed by the International Telegraph and Telephone Consultative Committee (CCITT). The H.261 video encoder is intended to be used to compress video data that is sent over Integrated Services Digital Network (ISDN) lines. The H.261 codec is intended primarily for use in video telephony and videoconferencing applications.

MPEG Decompressor

The MPEG-1 video compression standard was developed by the Moving Picture Experts Group. The group's goal was to compress full-motion video and the associated audio at the rate of about 1.5 Mbits/s. This is approximately the rate at which data can be read from a 1 x speed CD-ROM, so MPEG-1 compressed video is a good choice for use in interactive multimedia applications.

☰ *14*

CCITT Group 3 and Group 4 Codecs

The CCITT Group 3 and Group 4 compression standards were developed by the International Telegraph and Telephone Consultative Committee to enable facsimile machines to compress and decompress digitized documents. Now, Group 3 and Group 4 codecs are also used for general document storage and retrieval.

Cell and CellB Codecs

The Cell image compression technology, which was developed by Sun, has been optimized for the rapid decompression and display of images, especially on indexed-color frame buffers. The initial focus of the Cell technology is on Sun-to-Sun communications, where the benefits of fast decoding outweigh the benefits of standards. Possible areas of application include media distributions on CD-ROM and multimedia mail.

The CellB codec, which derives from its Cell counterpart, is intended for use primarily in videoconferencing applications. It features greater balance between the time spent compressing and decompressing images than the Cell codec. The CellB codec's strengths include:

- Software compression at interactive rates
- Very fast decoding and display, especially on indexed-color frame buffers
- Low rates of CPU use
- Good quality output

Basic CIS Management

This section explains several basic CIS-related tasks:

- Creating and destroying a CIS
- Putting compressed data in a CIS
- Reading data from a CIS

If you have looked at the example programs presented in Chapter 15, "Compressing and Decompressing Sequences of Images," you already know something about performing these tasks. Don't skip the following sections, however, because they introduce a number of functions that are not used in the examples.

Creating and Destroying a CIS

You create a CIS by calling the `xil_cis_create()` function. This function takes two arguments: a handle to the system-state data structure that was created when you initialized the XIL library and a string that identifies the codec that is used to write compressed data to, or read compressed data from, the CIS. For example, the call to create a CIS might look like this.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "Jpeg");
```

The string you pass to the function—`Jpeg` in the code fragment above—can be any one of the following strings:

- `Jpeg` (a JPEG baseline sequential codec)
- `JpegLL` (a JPEG lossless codec)
- `H261` (an H.261 decompressor)
- `Mpeg1` (an MPEG-1 decompressor)
- `faxG3` (a CCITT Group 3 codec)
- `faxG4` (a CCITT Group 4 codec)
- `Cell` (a Cell codec)
- `CellB` (a CellB codec)

The return value from `xil_cis_create()` is a handle to the newly created CIS. You use this handle as an argument to all subsequent functions that affect the CIS.

Three primary actions occur as a result of the `xil_cis_create()` call:

- An `XilCis` object is created and initialized.

- The shared object (`.so` file) for the requested codec is loaded (if not done already).

- Structures to manage the buffering of frames are created.

A function related to `xil_cis_create()` is `xil_cis_reset()`. The latter function takes an existing CIS, which may contain compressed data, and returns it to its initial state. That is, it clears out all existing compressed data

and frees any memory the CIS has allocated to hold that data. The only argument to `xil_cis_reset()`is a handle to the CIS whose compressed data you want to clear.

To destroy a CIS, you call the `xil_cis_destroy()` function. This function deallocates any buffers allocated to hold compressed data and frees the memory used to hold other information about the CIS. The function's only argument is a handle to the CIS.

Putting Compressed Data into a CIS

There are two primary ways to get compressed data into a CIS. One way is to call the `xil_compress()` function to compress one or more XIL images and write the compressed data to the CIS. The second way is to move an existing compressed data stream from system memory to a CIS. This compressed data may have been compressed originally by an XIL or a compatible non-XIL compressor.

Both of these methods change the value of an internal CIS index so that the next operation that writes compressed data to the CIS appends its data to existing data in the CIS. For more information about this index, see "Start Frame, Read Frame, Write Frame" on page 306.

Using `xil_compress()`

The `xil_compress()` function reads an XIL image (or possibly more than one image), compresses it, and writes the compressed data to a CIS. The function takes two arguments. The first argument is a handle to a source image, and the second is a handle to a CIS. You do not have to specify a compressor or codec because each CIS is associated with a single compressor/decompressor when the CIS is created.

Note – This description of what `xil_compress()` does is somewhat oversimplified. Some compressors may prefer to compress groups of frames. Therefore, they read an image each time your application calls `xil_compress()`, but they may not actually compress the images and write data to the CIS until they have read *n* frames. If you need to make sure that a

compressor has compressed all the frames it has read so far, call the
`xil_cis_flush()` function which is defined on "Performing Any
Outstanding Compression Operations" on page 325.

Each compressor has certain requirements regarding the type of XIL image you
can compress using `xil_compress()`. These requirements are summarized in
Table 14-1. An N/A in a table box indicates that there is no requirement in a
particular area.

Table 14-1 Types of Images Supported by XIL Compressors

Compressor	Width and Height	Number of Bands	Data Type	Color Space
JPEG baseline sequential	N/A	1 to 255	`XIL_BYTE`	N/A
JPEG lossless	N/A	1 to 255	`XIL_BYTE` `XIL_SHORT`	N/A
H.261	176 x 144 (QCIF) or 352 x 288 (CIF)	3	`XIL_BYTE`	`ycc601`
MPEG-1	N/A	3	`XIL_BYTE`	`ycc601`
CCITT Group 3	N/A	1	`XIL_BIT`	N/A
CCITT Group 4	N/A	1	`XIL_BIT`	N/A
Cell	Multiple of 4	3	`XIL_BYTE`	`rgb709` `ycc601` `ycc709`
CellB	Multiple of 4	3	`XIL_BYTE`	`ycc601`

Note – For information on the Common Interchange Format (CIF) and
Quarter-CIF (QCIF) images listed in Table 14-1, see "How an H.261 Codec
Works" on page 380.

You can also obtain information about the type of XIL image that can be
written to a particular CIS using the `xil_cis_get_input_type()` function.
For more information about this function, see "Input and Output Image Type"
on page 304.

14

Putting a Compressed Datastream into a CIS

If you have an existing stream of compressed data in system memory, you can put that data into a CIS in one of two ways. (In both cases, the compressed data must have been compressed with a compressor of the same type as the compressor/decompressor associated with the CIS.)

First, you can copy the datastream into the CIS buffer using the `xil_cis_put_bits()` function, whose function prototype is shown below.

```
void xil_cis_put_bits(XilCis cis, int nbytes, int nframes,
    void *data);
```

This function copies `nbytes` of compressed data representing `nframes` of uncompressed data from the caller's buffer into the CIS `cis`. The `data` argument is a generic pointer to the data to be copied into the CIS.

The argument `nframes` normally indicates the number of frames represented by `nbytes` of compressed data. If you do not know the exact number of frames in the datastream, but know that there are no partial frames (frames that are not completely represented in the datastream), you should set `nframes` to -1. This value informs the CIS that the data being placed into it contains one or more complete frames and no partial frames. If you know that the first or last frame is not represented completely by the data in the stream, or you're not sure whether the datastream contains a partial frame, you should set `nframes` to 0. This value alerts the compressor that the datastream may contain a partial frame.

Note – Don't set `nframes` to 0 if you know that the datastream contains only complete frames. Doing so slows down the copy.

You can also put a compressed datastream into a CIS using the `xil_cis_put_bits_ptr()` function, whose prototype is shown here.

```
void xil_cis_put_bits_ptr(XilCis cis, int nbytes, int nframes,
    void *data, XIL_FUNCPTR_DONE_WITH_DATA done_with_data);
```

This function does not copy data into the CIS buffer, but gives the CIS a pointer to the datastream. If you use this function, your application is responsible for ensuring that the datastream remains valid.

You must also free the memory where the datastream is stored. One way to do this is to pass `xil_cis_put_bits_ptr()` a pointer to a function (`done_with_data`) that frees the memory. The `xil_cis_put_bits_ptr()` function calls this function if the CIS is destroyed, is reset, or no longer needs the data. You should define this function to return `void` and to take as its only argument a generic pointer to `data`. Thus, the prototype for the function should look like this.

```
void free_memory(void *data)
```

When you call `xil_cis_put_bits_ptr()`, put the function's name in the spot where `done_with_data` is shown in the function prototype.

```
xil_cis_put_bits_ptr(cis, nbytes, nframes, data, free_memory);
```

If you don't pass `xil_cis_put_bits_ptr()` a pointer to a function, pass `NULL` as the final argument. In this case, your program must determine when the memory holding the datastream is no longer needed.

The value of `nframes` has the same meanings when you're using `xil_cis_put_bits_ptr()` as it does when you're using `xil_cis_put_bits()`.

Reading Data From a CIS

As with putting data into a CIS, there are two basic ways to get data out of a CIS. First, you can use the `xil_decompress()` function to read one frame's worth of compressed data from the CIS, decompress the data, and write the decompressed data to an XIL image. Second, you can get a pointer to the compressed data in the CIS and write the data to a file using a function like `fwrite()`.

Both of these methods change the value of an internal CIS index, so the next operation that reads compressed data from the CIS begins reading with the first frame in the CIS following the last frame read during the just-completed operation. For more information about this index, see "Start Frame, Read Frame, Write Frame" on page 306.

Using `xil_decompress()`

The function prototype for `xil_decompress()` is shown below.

```
void xil_decompress(XilCis cis, XilImage dst);
```

The first argument to the function is a handle to your CIS, and the second is a handle to an XIL image into which a frame of video can be decompressed. The decompressor used for the operation is the one you specified when you created your CIS.

The XIL image into which you decompress a frame of data must have the same width, height, number of bands, and data type as the images stored in the CIS. If you do not know these attributes of the images in the CIS, you can determine them using the `xil_cis_get_output_type()` function. For more information on this function, see "Input and Output Image Type" on page 304.

Copying Compressed Data From a CIS

If you want to copy data from a CIS without decompressing it, you should follow this procedure:

1. Use the `xil_cis_get_bits_ptr()` function to get a generic pointer to the compressed data in the CIS.

2. Use a function like `fwrite()` to copy the data from the CIS buffer to a file.

The function prototype for `xil_cis_get_bits_ptr()` is shown below.

```
void *xil_cis_get_bits_ptr(XilCis cis, int *nbytes,
    int *nframes);
```

The first argument to this function is a handle to the CIS. The second argument is the address of a variable in which the function stores the number of bytes to which its return value (a pointer to `void`) points. The third argument is the address of a variable in which the function stores the number of frames represented by `nbtyes`.

Note that `nbytes` is frequently not the total number of bytes of compressed data stored in the CIS. This is true because the CIS's "buffer" may actually be a list of buffers. As a result, you may have to call `xil_cis_get_bits_ptr()` multiple times to read all the compressed data in a CIS. See "Reading Data From a CIS in a Loop" for information about controlling loops in which you're reading data from a CIS.

Reading Data From a CIS in a Loop

If you want to decompress all the frames in a CIS or copy all the compressed data in a CIS to a file, you need to decompress data or copy data repeatedly, until all the data in the CIS has been read. A good way to control the loop that decompresses or copies data is to use the `xil_cis_has_frame()` function, as shown below.

```
XilCis cis;
XilImage dst;

while (xil_cis_has_frame(cis) == TRUE){
    xil_decompress(cis, dst);
}
```

The `xil_cis_has_frame()` function returns `TRUE` as long as one complete frame of compressed data remains in the CIS. Its single argument is a handle to your CIS.

Note – A loop like the one shown above works because a call to `xil_decompress()` or `xil_cis_get_bits_ptr()` changes an attribute of the CIS called its *read frame*. (For complete information about this attribute, see "Start Frame, Read Frame, Write Frame" on page 306.) If you decompress a frame, the read frame is incremented by 1, and if you copy 50 frames, the read frame is increased by 50.

You can also control this type of loop using the `xil_cis_has_data()` and `xil_cis_number_of_frames()` functions. The first of these functions returns the number of unread bytes in the CIS, and the second returns the number of unread frames. Neither function, however, is as effective as `xil_cis_has_frame()` for normal loop control, because `xil_cis_has_data()` can return a nonzero value even after the last complete frame has been read (if the CIS contains a partial frame), and `xil_cis_number_of_frames()` may take longer to execute than `xil_cis_has_frame()`.

Nonsequential Reads

Normally, when you read data from a CIS, you begin reading at the last frame read plus 1. However, you can also begin reading at a different point. To do this, you call the `xil_cis_seek()` function to indicate the frame you would like to read next. The prototype for this function is shown below.

```
void xil_cis_seek(XilCis cis, int framenumber, int relative_to);
```

The first argument to this function is a handle to your CIS. The second and third arguments, taken together, determine which frame becomes the CIS's read frame (the frame that is read first the next time you read data from the CIS). The second argument, `framenumber`, is a number of frames and is to be construed as an offset from one of three locations. This location is specified by the value of `relative_to`. This argument is analogous to the `direction` argument to the UNIX `lseek(3)` function. A value of 0 indicates that `framenumber` is an offset from the beginning of the CIS (frame 0), a value of 1 that `framenumber` is an offset from the current read frame, and a value of 2 that `framenumber` is an offset from the last frame in the CIS.

It is an error to seek to a frame prior to the first frame currently in the CIS or to a frame beyond the end of the CIS. For information about determining the frame numbers of the first and last frames in the CIS, see "Start Frame, Read Frame, Write Frame" on page 306. It is also an error to seek a frame prior to the current read frame if the CIS's random-access attribute is set to 0. For a discussion of this attribute, see the section "Random Access Flag" on page 306.

General CIS Attributes

Up to this point, this chapter has presented the CIS as a buffer or database in which you store compressed data. However, you can also think of a CIS as a structure with members that contain information about the compressor/decompressor associated with the CIS, the type of the images in a CIS, and so on. These members are referred to as *attributes* of the CIS. The following list shows you the general CIS attributes:

Note – The term *general attribute* is used because CISs also have codec-specific attributes. These are covered in chapters Chapter 16, "JPEG Baseline Sequential Codec" through Chapter 22, "CellB Codec," where the different XIL compressors/decompressors are discussed.

- Compressor (or decompressor) associated with the CIS
- Type of the compressor
- Type of image that can be compressed and stored in the CIS
- Type of image that results when compressed data in the CIS is decompressed
- Flag indicating whether the data in the CIS can be accessed randomly
- Index to the first image currently in the CIS
- Index to the image that is read next
- Index to the image that is written next
- Maximum number of images the CIS can contain
- Number of already-read images the CIS should try to keep in its buffer
- Flag indicating whether the compressor should automatically recover from an error it knows how to handle
- Name of the CIS

These attributes are discussed in detail in the following sections.

Compressor and Compression Type

A CIS's compressor attribute is a string that identifies the compressor/decompressor that is used to compress or decompress data for the CIS. This attribute is set when you create the CIS. For example, if you create your CIS using this statement, the attribute is set to `Cell`.

```
cis = xil_cis_create(state, "Cell");
```

You cannot change the value of this attribute after creating the CIS. However, you can read its value using the `xil_cis_get_compressor()` function.

The compression-type attribute is a string that identifies a compressor's class. In this release of the XIL library, each class contains only one compressor/decompressor. However, as Sun and third parties write new XIL compressors/decompressors, this situation will change. For instance, if a third party writes a JPEG baseline sequential compressor, that compressor will have a unique compressor name, but a compression type of `JPEG`, which is the compression type of the XIL JPEG baseline sequential codec. Like the compressor attribute, this attribute is set when you create your CIS and cannot be changed afterwards. You can retrieve the value, however, using the `xil_cis_get_compression_type()` function.

Table 14-2 Compressors and Compressor Types

Compressor	Compression Type
Jpeg	JPEG
JpegLL	JPEGLL
H261	H261
Mpeg1	MPEG1
faxG3	FAXG3
faxG4	FAXG4
Cell	CELL
CellB	CELLB

Input and Output Image Type

A CIS's input-image-type attribute is a data structure of type `XilImageType` that defines the type of XIL image that may be compressed and written to the CIS. This structure contains information about an image's width, height, number of bands, and data type.

14

This information is available as soon as the CIS is created and may be retrieved with the `xil_cis_get_input_type()` function. Say, for example, you use this statement to create a CIS.

```
cis = xil_cis_create(state, "Cell");
```

The statement creates a CIS that is written to by the Cell compressor. If you then call `xil_cis_get_input_type()` to determine the type of image that can be compressed and written to this CIS, you find that the image must have three bands and contain `XIL_BYTE` data. The Cell compressor can only work with images that have these characteristics. At this point, the values of the image type's width and height members are 0, which indicates that there are currently no specific requirements for these characteristics.

After you have used `xil_compress()` to write compressed data to your CIS, additional information about the input image type becomes available. For instance, a call to `xil_cis_get_input_type()` returns nonzero values for the image type's width and height. These correspond to the width and height of the images that have already been compressed.

Note – Once images have been compressed into a CIS, all additional images must have the same size, band count, and data type, unless the CIS is reset.

A CIS's output-data-type attribute is also a structure of type `XilImageType`, but indicates the type of image that is produced when data stored in the CIS is decompressed. You get this image-type structure by calling `xil_cis_get_output_type()`.

It's an error to call `xil_cis_get_output_type()`before writing data to your CIS. However, once the CIS contains data, `xil_cis_get_output_type()` returns information about the width, height, number of bands, and data type of the images stored in the CIS. In general, programs that decompress data must call this function because `xil_decompress()` expects its second argument to be an XIL image to which it can write its output. Before you can create this XIL image, you must know the output image type.

Random Access Flag

You can always seek forward in a CIS using the `xil_cis_seek()` function. However, only certain compressors/decompressors allow you to seek backwards. To find out whether your compressor/decompressor allows this type of seek, call the `xil_cis_get_random_access()` function to determine the value of the CIS's random-access attribute. A return value of 1 indicates that backward seeks are supported, and a return value of 0 indicates that such seeks are not allowed.

Start Frame, Read Frame, Write Frame

These attributes are integer indexes to certain important images or frames stored in a CIS.
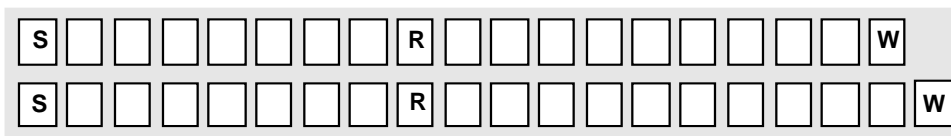
The start-frame attribute is an index to the first frame in the CIS that is currently accessible. For now, think of the start frame as the first frame written to the CIS (frame 0). To obtain the number of the current start frame, you call the `xil_cis_get_start_frame()` function.

The read-frame attribute is an index to the frame that is read the next time you read data from the CIS. If you use `xil_decompress()` to decompress a frame, the function decompresses the current read frame and then increments the read frame by 1. If you get a pointer to data you want to read using the `xil_cis_get_bits_ptr()` function, the read frame is incremented by the number of frames that you read from the CIS. You can determine the current read frame using the `xil_cis_get_read_frame()` function.

The write-frame attribute is an index to the frame that is written to the next time you put data into the CIS. (Thus, the index to the last frame currently in the CIS is the write-frame index minus 1.) When you add data to the CIS using `xil_compress()`, `xil_cis_put_bits()`, or `xil_cis_put_bits_ptr()`, this write-frame index is incremented by the number of frames added to the CIS. To read the current value of the write-frame attribute, use the `xil_cis_get_write_frame()` function.

The four diagrams below shed further light on how these indexes relate to one another. In each diagram, each row of rectangles represents a CIS buffer, and each rectangle in the row represents a frame of compressed data. Start frames, read frames, and write frames are labeled S, R, and W, respectively.

The first diagram shows a CIS buffer before and after a call to `xil_compress()`.

This second diagram illustrates the buffer before and after a call to `xil_decompress()`.

This third diagram shows the buffer before and after a call to `xil_cis_get_bits_ptr()` that returns a pointer to the data from the read frame to the write frame minus 1 inclusive. After obtaining this pointer, you can use a function like `fwrite()` to write the data to a file.

This final diagram shows the buffer before and after a seek operation. Only the read-frame index changes.

Maximum Frames and Keep Frames

The maximum-number-of-frames attribute specifies the maximum number of frames a CIS should buffer at any one time. You set this attribute using the `xil_cis_set_max_frames()` function, and you can read the value of the attribute using `xil_cis_get_max_frames()`.

Each CIS has a maximum-frames attribute whether or not you set it. If you don't set the value of the attribute, the value is a default value that depends on the codec associated with the CIS. You can set the value of the attribute by passing an integer greater than 0 to `xil_cis_set_max_frames()`. You can also pass a -1 to this function: this value indicates that there should be no limit on the number of frames that the CIS can buffer.

Note – In the case where you set the maximum frames attribute to an integer greater than 0, this setting is actually a suggestion rather than a requirement. Some codecs cannot function properly if a CIS's buffer is too small and will enforce a minimum value of maximum-frames.

The keep-frames attribute specifies the number of frames *prior to the current read frame* that a CIS should try to keep in its buffer. As with the maximum-frames attribute, the keep-frames attribute has a compressor-specific value if you do not set it. If you set the attribute, you can pass to `xil_cis_set_keep_frames()` an integer representing the number of frames to buffer or a -1, which means that there is no limit on the number of keep frames. You read the value of the keep-frames attribute using the `xil_cis_get_keep_frames()` function.

Note – Like the maximum-frames attribute, the keep-frames attribute is only a hint to the compressor/decompressor. Some decompression algorithms do not work if certain already-decompressed frames, such as key frames, aren't available.

The diagrams below illustrate how the maximum-frames and keep-frames attributes affect the state of the CIS buffer.

The first diagram shows how compressing frames affects a buffer that already contains the maximum number of frames. The first line in the illustration shows the initial state of the buffer. The second line shows how the buffer looks after a call to `xil_compress()`, the third line how the buffer looks after a second call to `xil_compress()`, and so on. If you lose the current read frame while adding new data to the buffer, an error occurs. An error also occurs if R minus S becomes less than the value of the keep-frames attribute. This error is reported only once, after the operation that first causes this condition.

Note – An error does not occur if you cause R minus S to become less than the value of the keep-frames attribute by seeking backwards in the buffer.

In the next two diagrams, R minus S equals the value of the keep-frames attribute in the initial view of the buffer.

The first diagram below shows the effect of decompressing a frame from the CIS. Note that S is incremented to prevent R minus S from exceeding the value of the keep-frames attribute. Frame S - 1 is no longer accessible.

This second diagram shows the CIS buffer before and after an operation that reads all the compressed data between the read frame and the write frame minus 1 inclusive. Note that R minus S equals the value of the keep-frames attributes in both depictions of the buffer.

Error-Recovery Flag

There are two types of XIL datastream errors: those that a codec or decompressor knows how to recover from and those that it can't recover from without help from the programmer. Here's an example of the former type of error. Some datastreams contain end-of-line markers. If a decompressor, while it is decoding data, finds a line that is longer or shorter than expected, it can (but doesn't have to) recover from this error by truncating or zero-filling the line. The value of a CIS's error-recovery flag determines whether a codec or decompressor automatically recovers from errors that it knows how t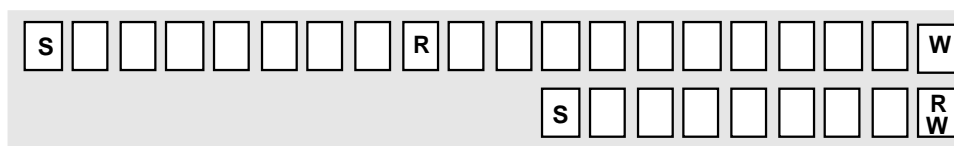o handle or doesn't. For information on recovering from errors that the module doesn't know how to handle, see "CIS Error Recovery" on page 313.

By default, a CIS's error-recovery flag is set to `FALSE`. This setting indicates that a codec or decompressor should not automatically recover from the type of error mentioned above.

Note – The library generates an error structure when any type of datastream error occurs and passes that structure to the currently installed error handler. The error handler can then deal with the error as it sees fit.

To set this flag to `TRUE` in order to enable automatic error recovery, you call the `xil_cis_set_autorecover()` function.

```
void xil_cis_set_autorecover(XilCis cis, Xil_boolean on_off);
```

The `cis` parameter is a handle to the CIS whose datastream is being accessed, and `on_off` is the enumeration constant `TRUE`. (You would use the constant `FALSE` to turn automatic recovery back off.)

The XIL library also includes a function that determines whether the error-recovery flag is currently set. This function is called `xil_cis_get_autorecover()`, whose prototype is shown below.

```
Xil_boolean xil_cis_get_autorecover(XilCis cis);
```

Note – If you set the error-recovery flag for a codec that cannot automatically recover from any datastream errors—like the Cell compressor—the flag has no effect.

CIS Naming

The library enables you to assign a name (`char *`) to a CIS. This type of naming is useful because it enables you to get a handle to a CIS later in your program by specifying the name of the CIS. The CIS naming functions are listed and described in Table 14-3.

Table 14-3 CIS Naming Functions

Function	Description
xil_cis_set_name()	Sets the name of a CIS
xil_cis_get_name()	Returns a copy of a CIS's name
xil_cis_get_by_name()	Returns a handle to the CIS that has the name you specify

CIS Error Recovery

This section discusses how you recover from datastream errors that a codec or decompressor does not know how to handle. For information on handling errors that such a module knows how to handle, see "Error-Recovery Flag" on page 311.

When a datastream error occurs that a codec or decompressor cannot recover from, two events take place:

- The CIS is marked invalid. If the datastream error occurs while data is being decompressed, the CIS is marked invalid for further reading. If the error occurs while data is being compressed, the CIS is marked invalid for further writing. To determine whether a CIS is invalid, you call either `xil_cis_get_read_invalid()` or `xil_cis_get_write_invalid()`.

- The XIL library generates an error structure and passes it to the currently installed error handler. If this is a user-installed error handler, it may take some action to deal with the datastream error.

Once a CIS has been marked invalid, there are three ways to make it valid again:

- Reset the CIS by calling `xil_cis_reset()`. This function clears all existing compressed data from the CIS.

- Seek a valid frame using `xil_cis_seek()`. If this seek fails, the library generates a seek error.

- Ask the library to recover from the error by calling `xil_cis_attempt_recovery()`. The remainder of this section explains how to use this function.

You use the `xil_cis_attempt_recovery()` function to try to recover from a datastream error that (1) occurred while you were decompressing a CIS and (2) made the CIS invalid for further reading. The prototype for this function is shown below.

```
void xil_cis_attempt_recovery(XilCis cis, unsigned int nframes,
    unsigned int nbytes);
```

The `cis` parameter is a handle to the CIS you're working with. The `nframes` parameter indicates the maximum number of frames the function should scan in its attempt to recover from the error. The `nbytes` parameter is the maximum number of bytes the function should scan. If both `nframes` and `nbytes` are set to 0, the function can search as far forward as necessary in order to recover.

If one of the last two parameters is zero and the other is nonzero, the function behaves as follows. If `nframes` is nonzero and `nbytes` is zero, the error recovery mechanism attempts to search `nframes` frames ahead, using its best guess as to exactly how many bytes those frames would contain. If `nframes` is zero, and `nbytes` is nonzero, the recovery routine scans `nbytes` bytes, regardless of how many frames those bytes represent.

When `xil_cis_attempt_recovery()` returns, you can check to see whether it was successful by calling the `xil_cis_get_read_invalid()` function. If you asked the recovery routine to scan a relatively few frames or bytes, it may be necessary to call the routine several times to recover from the error.

The example below shows `xil_cis_attempt_recovery()` being used inside a user-defined error handler. (For a complete discussion of writing error handlers, see Chapter 10, "Error Handling.")

```
Xil_boolean cis_error_handler(XilError error)
{
    XilCis cis;
    XilObject object;

    if ((xil_error_get_category(error) == XIL_ERROR_CIS_DATA) &&
        ((object = xil_error_get_object(error)) != NULL) &&
        (xil_object_get_type(object) == XIL_CIS)) {

        cis = (XilCis)object;
        if (xil_cis_get_read_invalid(cis)) {
            xil_cis_attempt_recovery(cis, 0, 0);
            if (!xil_cis_get_read_invalid(cis))
                return TRUE;
        }
    }
    return xil_call_next_error_handler(error);
}
```

If the error affects a CIS and the CIS has been marked read invalid, the error handler uses `xil_cis_attempt_recovery()` to try to recover from the error. The recovery routine has permission to scan the entire datastream if necessary. The error handler returns `TRUE` if the recovery is successful. Otherwise, it calls the next error handler.

Note – When `xil_cis_attempt_recovery()` is called from within an error handler, the call to `xil_decompress()` that produced the error fails even if the recovery is successful. The next call to `xil_decompress()` succeeds.

≡ 14

Compressing and Decompressing Sequences of Images   15 ≡

Before describing the details of specific compression modules (in a chapter dedicated to each), this guide provides an overview of how compression and decompression work by presenting the two example programs, `encode` and `xilcis_example`.

This chapter discusses these programs and has the following main sections:

Example Program Overviews

The first example program this chapter presents is called `encode`. This program builds a JPEG, Cell, or CellB movie. It reads a series of video frames from disk files, compresses each frame, and writes the compressed image data to a CIS. As frames are compressed, the program writes the compressed data in the CIS to an output file. This output file can serve as input to the second example program, a movie player.

The second example program, `xilcis_example`, plays back a JPEG, H.261, MPEG-1, Cell, or CellB datastream on an 8- or 24-bit frame buffer. It reads a datastream from a disk file and stores the compressed data in a CIS. Then, for each compressed image in the CIS, it takes the following actions. First, it

decompresses a frame of video from the CIS and stores the resulting image in a 3-band `XIL_BYTE` image. Then, it prepares each image for display on a particular frame buffer and writes the image to a display image. These steps result in a frame of video being displayed.

Creating a JPEG Datastream

The source files for the movie-making example can be found in the `/usr/openwin/demo/xil/ movie_maker_example` directory. Table 15-1 lists these source files and indicates what the code in each file does.

Table 15-1 Source Files for `Movie-Maker`

Source File	Description
encode.c	Contains `main()`, which prepares different types of source images to be copied to various types of display images (is the same as `example1.c` in the `example1` program)
fileio.c	Reads an image from a file and loads the image data into an XIL image

The `movie_maker_example` directory also contains the image files that the program uses as input and a `Makefile` you can use to build the program.

After briefly discussing how to build and run the example, this section walks you through the program's code to clarify how the program produces a JPEG datastream. (Cell and CellB bytestreams are produced in a very similar way.) The basic algorithm is shown below:

1. Create a CIS.

2. While all the video frames have not been compressed, perform the following steps:

   a. Load data from an image file into an XIL image.

   b. Convert the format of the XIL image if that's necessary or desirable. It is necessary if the image does not contain `XIL_BYTE` data because the JPEG compressor expects input of that type. If your image is not a YC_bC_r image, it is desirable to convert the image to that color space because YC_bC_r images are optimized for compression and decompression. The JPEG compressor works with images of any color space, but it typically produces superior compression with YC_bC_r images.

    c. Compress the image and write the compressed data to the CIS.

    d. Destroy the image in order to free the memory allocated to hold the image structure and the image data.

    e. Write any compressed data in the CIS to a disk file.

3. Perform any outstanding compression operations, and write any resulting compressed data to the output file.

Building and Running the Example

To build and run the `encode` example:

1. **Change your working directory to**
   `/usr/openwin/demo/xil/movie_maker_example`**.**

2. **Type** `make`**.**
   The `Makefile` in that directory builds the program.

3. **Execute the program using the following syntax:**

   ```
   % encode [-c | -cb] file-list [output-file]
   ```

By default, the program creates a JPEG bitstream. To produce a Cell bytestream instead, use the `-c` option, and to produce a CellB bytestream, use the `-cb` option.

The *file-list* argument must be the name of a file that contains a list of files (images) to be processed. The list of files supplied with this example is called `mifkin.list`.

The *output-file* argument is optional. If you supply a file name here, the example writes its output to a file of that name; otherwise, it writes its output to `out.jpg`, `out.cell`, or `out.cellb`, depending on which compressor you've used.

≡ *15*

Creating a CIS

Before a program can compress an image or sequence of images, it must create a CIS in which to store compressed data. The example program creates this data structure using code similar to this.

```
XilSystemState state;
XilCis cis;

cis = xil_cis_create(state, "Jpeg");
xil_cis_set_attribute(cis, "ENCODE_411_INTERLEAVED",
    (void *)TRUE);
xil_cis_set_max_frames(cis, 100);
```

You use the `xil_cis_create()` function to create the CIS, which is a data structure of type `XilCis`. The two arguments to this function are a system-state structure (this was returned by an earlier call to `xil_open()`) and the name of a compressor/decompressor. The example requests the JPEG baseline sequential codec (`Jpeg`), but could have requested another compressor/decompressor. (For information about the strings you use to create CISs associated with other compressors/decompressors, see "Creating and Destroying a CIS" on page 295.)

The next line sets a JPEG-specific attribute called `ENCODE_411_INTERLEAVED` to `TRUE`. If the images being compressed are 3-band YC_bC_r images, setting this attribute causes the codec to subsample the data in the color bands, so only one color value is encoded for each four color values in the original (4:1:1). This subsampling enables the compressor to achieve a much higher level of compression than it could otherwise.

The last line sets a CIS attribute that limits the number of frames the CIS buffers. In this case, the CIS maintains 100 frames in its buffer. If more than 100 frames are compressed, the oldest frames become inaccessible. This means that it is not possible to seek back more than 100 frames. It also means that 100 frames is the maximum number of unplayed (decompressed) frames that can be accumulated in the CIS. This attribute is set to keep the program from using too much memory in instances when there are many frames to be compressed.

CIS attributes are discussed in more detail in "General CIS Attributes" on page 303.

Compressing Video Frames and Writing Compressed Data to a File

After creating a CIS, the example begins compressing frames of video. For each frame, the program performs the steps described in the sections below.

Loading Data From a File into an XIL Image

To read an image from a disk file into an XIL image, the example calls the `load_example_image_file()` function. `load_example_image_file()` is very similar to the XIL file loading functions in previous examples except that it reads a color space from the image file header. After creating the XIL image into which it reads pixel values, it sets that XIL image's color-space attribute appropriately. See the following code fragment in `fileio.c`.

```
...
XilSystemState state;
...
XilImage image;
unsigned int width, height, nbands, depth;
XilDataType datatype;
char colorspace[80];
...
FILE* header_file;

fscanf(header_file, "%s", colorspace);
...
image = xil_create(state, width, height, nbands, datatype);
...
xil_set_colorspace(image, xil_colorspace_get_by_name(state,
    colorspace));
...
```

You check the color space of the input image because the JPEG compressor is optimized for compressing YC_bC_r images. The images supplied with the example are RGB images, so the example converts each image to the YC_bC_r color space before compressing it. This task is described in the next section.

Converting an XIL Image to the Proper Format

Compressors have certain requirements for the images they compress. (For a list of these requirements, see "Putting Compressed Data into a CIS" on page 296.) For example, the JPEG compressor used in this example requires `XIL_BYTE` images. In addition, the compressor would produce a bitstream that can be decompressed most rapidly if the input is YC_bC_r images.

The images provided for you to use as input to the example are RGB `XIL_BYTE` images. Thus, although the example does not have to convert the images in any way, it can produce a bitstream that can be played back the fastest if it converts the color space of the images. This conversion is handled in the `image_transform_colorspace()` function in `encode.c`.

```
XilImage
image_transform(XilSystemState state, XilImage image,
    int desired_nbands, char *desired_colorspace)
{
    XilImage new_image;
    unsigned int width, height, nbands;
    XilDataType datatype;

    xil_get_info(image, &width, &height, &nbands, &datatype);
    new_image = xil_create(state, width, height, desired_nbands,
        datatype);
    if (new_image == NULL) {
      /* XIL sends an error msg to stderr if image create fails */
        return(NULL);
    }
    xil_set_colorspace(new_image,
        xil_colorspace_get_by_name(state, desired_colorspace));
    xil_color_convert(image, new_image);
    xil_destroy(image);
    return(new_image);
}
```

This routine is passed a source image and a desired color space (`ycc601`). Given this data, the function:

- Creates a second XIL image, `new_image`, that has the same width, height, and data type as `image` and a number of bands that fits the desired color space. Because a YC_bC_r image must have three bands, `new_image` is a 3-band image.

- Uses the `xil_set_colorspace()` function to set `new_image`'s color space attribute to `ycc601`.

- Calls `xil_color_convert()` to convert the data in `image` from the RGB color space to the YC_bC_r color space. The converted data is stored in `new_image`.

- Destroys `image` since it is no longer needed.

A JPEG playback program runs fastest if the images it decompresses are multiples of 16 in width and height. For this reason, the example (`encode.c`) uses the following code to clip each source image to a multiple of 16 in width and height before compressing it.

```
XilImage src, child_clip;

child_clip = xil_create_child(src, 0, 0,
                              xil_get_width(src) & ~0xf,
                              xil_get_height(src) & ~0xf, 0, 3);
```

Compressing an XIL Image

Compressing the clipped YC_bC_r image requires a single function call.

```
XilImage src;
XilCis cis;

xil_compress(child_clip, cis);
```

The first argument to this function is the child YC_bC_r image `child_clip`, and the second is a handle to the program's CIS. The call both compresses the image and writes the compressed data to the CIS.

≡ *15*

Destroying an XIL Image

Once an image has been compressed, the example destroys that image to free the resources associated with it.

```
xil_destroy(child_clip);
xil_destroy(src);
```

Writing the Compressed Data to a File

After compressing each frame of video, the example calls a `write_file()` function to write any compressed data in the CIS to the program's output file. The basic algorithm for this routine is shown below.

While the CIS contains data that has not been read:

- Get a pointer to the first byte of compressed data that has not been read. After the first call to the function that gets this pointer, the pointer points to the first byte of data in the CIS.

- Using the pointer retrieved in the previous step, read data from the CIS and write it to a file.

Note – The loop is necessary because all of the compressed data in the CIS may not be in one contiguous buffer.

The following code implements this algorithm.

```
int nbytes, nframes;
XilCis cis;
Xil_unsigned8 *data;
FILE *out;
...
while (xil_cis_has_frame(cis) == TRUE) {
    data = (Xil_unsigned8 *)xil_cis_get_bits_ptr(cis, &nbytes,
        &nframes);
...
    fwrite((char *)data, sizeof(Xil_unsigned8), nbytes, out);
}
```

The `xil_cis_has_frame()` function determines whether there is at least one complete frame's worth of compressed data in the CIS that has not been read. If there is remaining data, the function returns TRUE, and the statements inside the loop are executed.

Note – The function must check for more than one frame's worth of data in the CIS because compressors are not required to write data to the CIS each time `xil_compress()` is called. Instead, they might read a number of frames of video and store them internally before doing any compression. Then, they might write a number of frames' worth of compressed data to the CIS at once. This type of strategy is necessary, for instance, when a compressor wants to do interframe compression.

The first call to `xil_cis_get_bits_ptr()` returns a generic pointer to the beginning of the compressed data in the CIS. It also returns the number of bytes the pointer points to (`nbytes`) and the number of video frames these bytes represent (`nframes`). In addition to returning these values, the function changes an attribute of the CIS called its *read frame* from 0 to `nframes`. (Frames are numbered beginning with 0.) Thus, a second call to `xil_cis_get_bits_ptr()` returns a pointer to the first byte of compressed data that is part of the frame numbered `nframes`. The arguments to this function are a handle to your CIS and the addresses of the variables `nbytes` and `nframes`.

The only other action taking place in the loop is that the example uses the `fwrite()` function to write the compressed data pointed to by `data` to the output file `out`.

Performing Any Outstanding Compression Operations

When the loop described in the preceding section ends, the compressor might have read frames of video that it has not yet compressed. To deal with this possibility, the example makes the following two calls.

```
XilCis cis;
FILE *out;

xil_cis_flush(cis);
write_file(cis, &total_nbytes, &total_nframes, out);
```

The `xil_cis_flush()` function instructs the compressor to compress any images it has read but not compressed and to write the last of its output to the CIS. The call to `write_file()` is the same as the one used in the loop discussed in the last section. It writes to the output file any compressed data written to the CIS as a result of the call to `xil_cis_flush()`.

Playing a JPEG Movie

This section discusses a movie-player example supplied with the XIL library. This program can display Cell, CellB, JPEG, H.261, and MPEG-1 movies on 8- and 24-bit displays.

The source files for the movie-player example can be found in the `/usr/openwin/demo/xil/ movie_player_example` directory. Table 15-2 lists these source files and indicates what the code in each file does.

Table 15-2 Source Files for `Movie-Player`

Source File	Description
`xilcis_example.c`	Contains `main()`, which displays either single frames or movie sequences of JPEG or CELL encoded images
`memmap.c`	Maps the contents of a file containing a movie to system memory
`xilcis_color.c`	Sets up the X colormap to display dithered frames

The `/movie_player_example` directory also contains a one-frame JPEG bitstream that the example can use as input and a `Makefile` that you can use to build the program. In addition, you can use as input to this program the output of the movie-maker example discussed in "Creating a JPEG Datastream" on page 318.

Note – The JPEG code in the program is optimized to handle movies containing YC_bC_r images. The JPEG movie supplied with the example contains this type of image.

After briefly discussing how to build and run the example, the section takes a look at the code the example uses to play back a JPEG movie on an 8-bit display. (If you look at the program, you'll notice that the CellB, H.261, and

MPEG-1 cases are handled very similarly to the JPEG case.) The code specific to the Cell case for 8-bit displays is covered in "Playing Cell Movies" on page 338.

The basic algorithm the example uses to play a JPEG movie on an indexed-color display is shown below.

1. Memory map the contents of the movie file (the compressed data) into your process's address space.

2. Create a CIS.

3. Give the CIS a pointer to the compressed data you memory mapped earlier. At this point, the CIS is all set up.

4. Create a display image in which to show the movie by following these steps:

   a. Determine the dimensions of the frames in the movie.

   b. Create an X window that is equal in width and height to the frames in the movie.

   c. Create an XIL display image from the X window.

5. Create an XIL image that holds frames as they are decompressed. This image must have the same width, height, number of bands, and data type as the frames in the movie.

6. Initialize the parameters the example uses when dithering each 24-bit XIL image (frame of video) so that it can be displayed on an 8-bit frame buffer.

7. Install an X colormap.

8. Play back the movie. While the CIS contains unread frames of compressed data:

   a. Decompress a frame of video and store the resulting image in the XIL image created above.

   b. Dither the XIL image, and write the dithered image to the display.

≡ *15*

Running the Movie Player

To run the movie player, change your working directory to `/usr/openwin/demo/xil/movie_player_example`, and then build the program in that directory by typing `make`. You should execute the example from this same directory, using a command line of the following form:

% **xilcis_example** [-i *filename*] [-h | -m | -c | -cb] [-s *width height*]

Table 15-3 Command-Line Options for `xilcis_example`

Option	Description
-i *filename*	Specifies the name of the movie (a file containing compressed movie frames) that you want to play back. If you omit this option, the program plays back a one-frame movie called `earth.jpg`.
-h	Instructs the movie player to play back an H.261 bitstream.
-m	Instructs the movie player to play back an MPEG-1 bitstream.
-c	Instructs the movie player to play back a Cell movie. By default, the player is set up to play a JPEG movie.
-cb	Instructs the movie player to play back a CellB bytestream.
-s *width height*	If you're decoding a CellB bytestream, you must use this option. The *width* and *height* arguments are the width and height in pixels of the images being decoded.

When the program runs, it shows the movie you have selected in an X window. To terminate the program, move your cursor into the program's window and press any mouse button.

Memory Mapping the Movie

When you run the example program, you use the -i *filename* option to pass it the name of a file containing a movie. Before the example can attach the compressed image data in the file to a CIS, it must map the contents of the file to system memory. You can find the code that handles this task in the source file `memmap.c`.

As you see in that source file, after opening the movie file, the example uses the `fstat(2V)` system call to determine the length of the movie in bytes and the `mmap(2)` system call to memory map the file and get a pointer to the

beginning of the datastream. Following these calls, the `memfile->mstart` structure member is a pointer to the beginning of the datastream, and `memfile->mlen` is the number of bytes in the datastream.

Creating a CIS

After mapping the compressed movie data to memory, the example creates a CIS to hold that data. The following code creates this `XilCis` object.

```
XilCis cis;
XilSystemState state;
char *cis_type = "Jpeg";

cis = xil_cis_create(state, cis_type);
if (!cis) {
    /* XIL sends error message to stderr if xil_cis_create fails */
     exit(1);
}
```

The arguments to `xil_cis_create()` are a handle to a system-state data structure returned by an earlier call to `xil_open()` and the name of the codec or decompressor to decompress frames from the datastream, in this case "`Jpeg`". The function returns a handle to the newly created CIS.

Putting Compressed Data in a CIS

Now that the CIS exists and the example has a pointer to the JPEG bitstream, the example can set the data-pointer member of the CIS to point to the beginning of the JPEG bitstream. It does this using a call to `xil_cis_put_bits_ptr()`.

```
XilCis cis;
int frame_count = -1;

xil_cis_put_bits_ptr(cis, memfile->mlen, frame_count,
    memfile->mstart, NULL);
```

Table 15-4 Arguments to `xil_cis_put_bits_ptr()`

Argument	Description
`cis`	A handle to the CIS whose data pointer is being set.
`memfile->mlen`	The length in bytes of the JPEG bitstream.
`frame_count`	The number of video frames the JPEG bitstream represents is unknown (because `frame_count` is set to -1). If the number of frames in the movie were known, `frame_count` would be set to that number.
`memfile->mstart`	A pointer to the beginning of the JPEG bitstream.
`NULL`	The example has assumed responsibility for freeing the memory in which the JPEG bitstream is stored. Alternatively, the example could have passed a pointer to a function used to free or unmap the bitstream memory. In the latter case, the function passed to `xil_cis_put_bits_ptr()` would be called if the CIS were destroyed or reset.

After the call to `xil_cis_put_bits_ptr()` has been made, the CIS is ready for use. Before the example can begin decompressing frames of video, however, it must create two XIL images: an 8-bit display image in which to display the video and a 24-bit memory image to serve as a destination image for the decompression function.

Creating a Display Image

There are three steps involved in creating the 8-bit display image in which the movie is shown.

1. Determine the dimensions of the images stored in the CIS.
The code used to get these dimensions is shown below.

```
XilImageType outputtype;
XilCis cis;
unsigned int cis_xsize, cis_ysize, cis_nbands;
XilDataType cis_datatype;

outputtype = xil_cis_get_output_type(cis);
xil_imagetype_get_info(outputtype, &cis_xsize, &cis_ysize,
     &cis_nbands, &cis_datatype);
```

The first function called, `xil_cis_get_output_type()`, takes a handle to the CIS as its only argument and returns an image type. This image type, an object of type `XilImageType`, contains information about the images stored in the CIS, such as their width, height, number of bands, and data type. For more information the image type of an XIL image, see "Image Type" on page 59.

The following call to `xil_imagetype_get_info()` takes the image type as its first argument and returns the width, height, number of bands, and data type of the image type in its remaining arguments. The data type, stored in `cis_datatype`, is one of the following enumeration constants: `XIL_BIT`, `XIL_BYTE`, or `XIL_SHORT`. (In this case, the value is `XIL_BYTE` because the movie supplied with the XIL release was made from 3-band `XIL_BYTE` images.)

Note – Currently none of the XIL codec modules supports `XIL_FLOAT` data.

---

2. **Create an X window that matches the width and height of the images in the CIS.**
In this example, the X window is created with a call to
`XCreateSimpleWindow()`. When the example is run on a system with an
8-bit display, this window is 8 bits deep by default.

```
int screen_num;
Display *display;
Window window;

screen_num = DefaultScreen(display);
window = XCreateSimpleWindow(display,
    RootWindow(display, screen_num), 0, 0, cis_xsize,
    cis_ysize, 0, BlackPixel(display, screen_num),
    WhitePixel(display, screen_num));
```

3. **Create a display image based on the X window created above.**
This display image is the destination image to which frames are written in
order to display them. To create the display image, the example calls
`xil_create_from_window()`.

```
XilImage displayimage = NULL;
XilSystemState state;
Display *display;
Window window;

displayimage = xil_create_from_window(state, display, window);
if (!displayimage) {
    /* XIL err msg to stderr if xil_create_from_window fails */
    exit(1);
}
```

For a complete discussion of display images, see "Display Images" on
page 45.

Creating an Image to Hold Decompressed Frames

If you have loaded the JPEG movie supplied with the XIL release (`earth.jpg`)
or a JPEG movie you created with the example movie maker into your CIS, the
images in the CIS are YC_bC_r images (24 bits deep). Because they cannot be

decompressed directly into the program's display image (8 bits deep), the example must create an XIL memory image whose purpose is to hold those decompressed frames. That is, for each frame in the movie, the example must perform the two-step sequence shown in Figure 15-1.
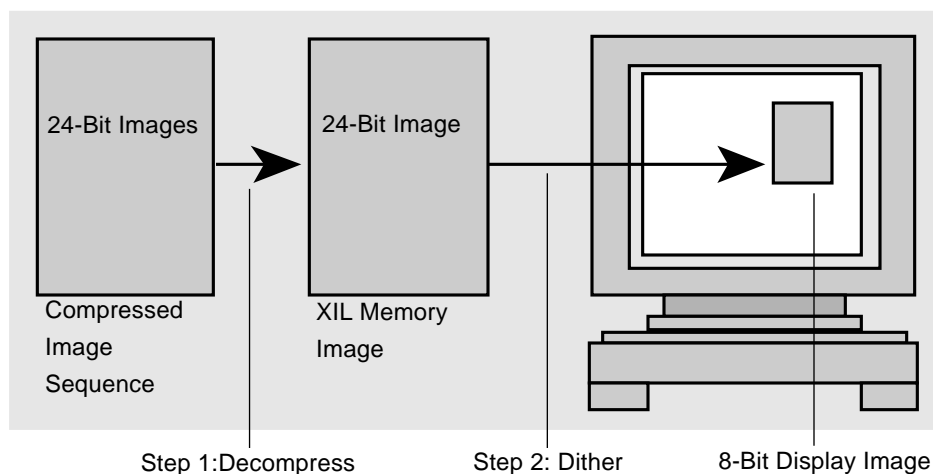

Figure 15-1   Decompressing and Dithering a Frame of Video

The code used to create the XIL memory image is shown below.

```
XilSystemState state;
XilImage imageYCC = NULL;
unsigned int cis_xsize, cis_ysize, cis_nbands;
XilDataType cis_datatype;

imageYCC = xil_create(state, cis_xsize, cis_ysize, cis_nbands,
    cis_datatype);
```

The `cis_xsize, cis_ysize, cis_nbands,` and `cis_datatype` variables were set earlier by a call to `xil_imagetype_get_info()`, which returned the width, height, number of bands, and data type of the compressed images in the CIS. Thus, the image returned here, `imageYCC`, is identical to the images stored in the CIS in both dimensions and data type.

Initializing Parameters to Be Used With the Dither Function

Next, the example program produces the colorcube and dither mask that is need later to dither images to the display. To get the colorcube, the example calls `xil_lookup_get_by_name()`.

```
XilLookup colorcube;

colorcube = xil_lookup_get_by_name(state, "cc855");
```

This call returns a handle to a special colorcube that the library creates when it is initialized. This colorcube has dimensions of 8, 5, and 5 and is designed for dithering the values in a YC_bC_r image to 200 colors.

The program then creates a dither mask.

```
XilDitherMask dmask;

dmask = xil_dithermask_get_by_name(state, "dm443");
```

This call returns a handle to a special dither mask the library creates when it is initialized. The mask is 4 pixels high and wide and 3 bands deep.

For more information on dithering, see "Dithering an Image" on page 161.

Installing an X Colormap

At this point, the program calls the `create_cmap()` function, from the `xilcis_color.c` source file, to set up the X colormap that is used in displaying the dithered frames. This function contains primarily Xlib code, but does get some information from the XIL library to get started.

First, the function is passed an XIL lookup table called `yuv_to_rgb`. This is a standard lookup table that is useful for converting the values in a dithered YC_bC_r image to RGB values.

Note – The YC_bC_r images must have been dithered using the standard colorcube `cc855`.

This lookup table specifies which color values should be written to which colorcells in the X colormap.

The `create_cmap()` function then calls `xil_lookup_get_num_entries()` to determine the number of entries in the colormap.

```
int cmapsize;

cmapsize = xil_lookup_get_num_entries(yuvtorgb);
```

Once the function knows how many colorcells it needs to allocate, it creates an X colormap and allocates the colorcells it needs to hold the information stored in `yuvtorgb`. Note that to avoid colormap flashing, the function avoids allocating the last two colorcells in the X colormap if possible and also leaves free as many colorcells as possible at the beginning of the X colormap.

At this point, `create_cmap()` allocates a buffer called `data` and then uses the `xil_lookup_get_values()` function to copy colormap entries from the lookup table `yuvtorgb` to the buffer `data`.

```
Xil_unsigned8 *data;

data = (Xil_unsigned8 *)malloc(sizeof(Xil_unsigned8) * cmapsize
    * 3);
xil_lookup_get_values(yuvtorgb,
    xil_lookup_get_offset(yuvtorgb), cmapsize, data);
```

The second argument to `xil_lookup_get_values()` enables you to start copying entries from the lookup table at an entry other than 0, and here is set to the offset of `yuvtorgb`.

Once `create_cmap()` has copied the colormap information from `yuvtorgb` to the buffer `data`, the function has direct access to the color values it needs to store in the X colormap, and it goes on to store those values. The only remaining XIL call in the `create_cmap()` function changes the offset of the colorcube that `main` uses to dither decompressed images to match the offset at which `create_cmap()` began writing color values to the X colormap.

```
xil_lookup_set_offset(colorcube, (unsigned int)pixels[0]);
```

The first argument to `xil_lookup_set_offset()` is a handle to the colorcube—`main` passed this handle to `cmap_create()`—and the second represents the pixel value associated with the first colorcell in which `cmap_create()` stored color values. This call ensures that all the pixel values in the dithered images created in `main` map to the correct spot in the X colormap.

Playing the Movie

After returning from `create_cmap()`, the program is ready to play the movie. The calls it uses to do this are shown below.

```
while (xil_cis_has_frame(cis)) {
    xil_decompress(cis, imageYCC);
    xil_rescale(imageYCC, imageYCC, scale, offset);
    xil_ordered_dither(imageYCC, displayimage, colorcube,
        dmask);
}
```

The loop is controlled by the return value of `xil_cis_has_frame()`. As long as the CIS contains at least one complete frame of video, the function returns 1, and the statements in the loop are performed.

The call to `xil_decompress()` decompresses an image from the CIS and stores it in the intermediate image `imageYCC`. Since this is a CCIR Rec. 601 YC_bC_r image, the values in the Y band of the image can range from 16 to 235, and those in the color bands can range from 16 to 240. Before such an image is dithered, the values in each band should be scaled to fall in the range 0 to 255.

This rescaling is handled by the call to `xil_rescale()`. Since `imageYCC` is both the source and destination image for the operation, the rescale is performed in place. The scale and offset arguments were defined earlier as shown here.

```
float scale[3], offset[3];

scale[0] = 255.0 / (235.0 - 16.0);
scale[1] = 255.0 / (240.0 - 16.0);
scale[2] = 255.0 / (240.0 - 16.0);
offset[0] = -16.0 * scale[0];
offset[1] = -16.0 * scale[1];
offset[2] = -16.0 * scale[2];
```

Finally, `xil_ordered_dither()` dithers the 3-band YC_bC_r image in `imageYCC` using the colorcube and dither mask created earlier and writes the result to the `displayimage` display image. This results in a frame of the movie being shown in the X window.

Note – When a JPEG movie made from YC_bC_r images is displayed on an 8-bit frame buffer, as in this example, the exact sequence of calls you use to play the movie has a dramatic effect on the speed with which the movie is shown. The reason for this is that by using the deferred-execution scheme explained in Chapter 23, "Acceleration in XIL Programs," the XIL library can look for a certain sequence of functions at runtime and, if it finds that sequence, replace all the functions in the sequence with a single, highly optimized routine. In this example, the sequence `xil_decompress()`, `xil_rescale()`, `xil_ordered_dither()` is such a sequence. Thus, when you play a JPEG movie using this program, these functions are not executed, but are replaced by an optimized function (*molecule*) that performs the jobs of all three functions. For a complete list of decompression molecules, see "Video Decompression Molecules" on page 478.

☰ *15*

Playing Cell Movies

The example movie player performs the same steps through the point where it creates a display image, regardless of whether its input is a JPEG movie or a Cell movie. From that point on, however, it treats the JPEG and Cell cases differently to get the maximum playback speed for each case. The algorithm for the Cell case is summarized below:

1. Install an X colormap.

2. Create an XIL image to hold images as they are decompressed.

3. Decompress frames from the movie and display them.

Each of these steps is considered in more detail in the following sections.

Installing an X Colormap

As you look at this part of the example program, keep in mind that the Cell playback code may update the X colormap many times as it plays back a movie. In fact, the decompressor may use a different colormap for each frame of a movie. This need to modify the X colormap makes colormap handling a bit more complex than it is in the JPEG case.

The first thing the example does in this section is to call the `create_cmap()` function, which is defined in the `xilcis_color.c` file.

```
XilLookup xil_cmap;
Colormap x_cmap;
XilIndexList *ilist;

xil_cmap = create_cmap(state, cis, display, window,
  DefaultScreen(display), &x_cmap, CELL, ilist, NULL, NULL);
```

This routine returns an XIL lookup table that serves as the colormap for the Cell decompressor. The `&x_cmap` argument is the address of an X `Colormap`, and `ilist` is a structure that is defined (in an XIL header file) as shown here.

```
typedef struct
{
    Xil_unsigned32 *pixels;
    Xil_unsigned16 ncolors;
} XilIndexList;
```

The `pixels` array can hold up to `ncolors` colormap index values, and these values later determine which X colormap entries the Cell decompressor can modify.

The first thing the `create_cmap()` function does is to look at the CIS's `DECOMPRESSOR_MAX_CMAP_SIZE` attribute to determine how many colorcells to allocate in the X colormap.

```
int cmapsize;

xil_cis_get_attribute(cis, "DECOMPRESSOR_MAX_CMAP_SIZE",
    (void **)&cmapsize);
```

The function then creates an X colormap and allocates `cmapsize` colorcells in that colormap.

The next section of code sets up the `ilist` structure used to determine which X colormap entries the Cell decompressor can modify. In this example, `ilist` is set up so that the decompressor can modify any of the colorcells allocated by `create_cmap()`.

```
ilist->pixels = (Xil_unsigned32 *)malloc(sizeof(Xil_unsigned32)
    * cmapsize);
ilist->ncolors = cmapsize;
for (i = 0; i < cmapsize; i++)
    ilist->pixels[i] = (Xil_unsigned32)pixels[i];
```

Next, the `cmap_create()` function creates the XIL colormap that it passes back to `main`. Because the XIL function that creates this colormap requires as one of its arguments an array of bytes containing the color values to be stored in the colormap, `cmap_create()` first reads the color values in the default X colormap into a buffer called `data`. It then uses `xil_lookup_create()` to create the XIL colormap.

```
data = (Xil_unsigned8 *)malloc(sizeof(Xil_unsigned8) * cmapsize
    * 3);
for (i = 0; i < cmapsize; i++)
    cdefs[i].pixel = i + pixels[0];
XQueryColors(display, DefaultColormap(display, screen), cdefs,
    cmapsize);
for (i = 0, j = 0; i < cmapsize; i++, j += 3) {
    data[j] = cdefs[i].blue >> 8;
    data[j + 1] = cdefs[i].green >> 8;
    data[j + 2] = cdefs[i].red >> 8;
}
lut = xil_lookup_create(state, XIL_BYTE, XIL_BYTE, 3, cmapsize,
    (int)pixels[0], data);
```

The lookup table created contains three bands on the output side and `cmapsize` entries. Also, note that color values are entered in the table in BGR order.

After `create_cmap()` returns the XIL lookup table to `main`, the example gets the version number of this colormap (`xil_cmap`) as shown here.

```
XilVersionNumber lu_version;

lu_version = xil_lookup_get_version(xil_cmap);
```

This version number, `lu_version`, changes whenever the lookup table's contents are changed by the Cell decompressor. Therefore, after decompressing each movie frame, the program can test the version number to determine whether it needs to update the X colormap.

The example then sets two important attributes of the Cell decompressor: `DECOMPRESSOR_COLORMAP` and `RDWR_INDICES`.

```
xil_cis_set_attribute(cis, "DECOMPRESSOR_COLORMAP", xil_cmap);
xil_cis_set_attribute(cis, "RDWR_INDICES", ilist);
```

The `DECOMPRESSOR_COLORMAP` attribute specifies the XIL lookup table that serves as the Cell decompressor's XIL colormap. Because colormap information is stored in the Cell bytestream along with pixel values, any time a frame is decompressed, the contents of the decompressor's colormap can change. When the contents of the colormap change, a subset of its entries are written to the X colormap before the next video frame is displayed.

By default, however, the decompressor's colormap is read only. To make an entry writable, you register its index in `RDWR_INDICES` (in this case, `ilist->pixels`). Only the entries whose indexes are stored in this array can change. For example, in this program, only the entries whose indexes correspond to those of the X colorcells allocated earlier can be written to.

Note – If you want to play Cell movies using a fixed colormap (for example, to avoid colormap flashing between windows), don't set the `RDWR_INDICES` attribute. In this case, the Cell compressor does its best using the colors stored in the lookup table `xil_cmap`.

Creating an Image to Hold Decompressed Frames

The program now creates a 3-band `XIL_BYTE` image to hold frames as they are decompressed. This task is handled with a call to `xil_create()`as shown here.

```
XilImage image24 = NULL;

image24  = xil_create(state, cis_xsize, cis_ysize, 3, XIL_BYTE);
```

Later in the program, these images are converted to 8-bit images for display.

Playing the Movie

Like the JPEG player, the Cell player plays a movie by executing a few statements in a loop that terminates when the function call `xil_cis_has_frame(cis)` returns 0.

First, the example decompresses a frame of data using the `xil_decompress()` function and checks the version number of the decompressor's XIL colormap as shown here.

```
xil_decompress(cis, image24);
if (lu_version != xil_lookup_get_version(xil_cmap))
{
    cell_install_cmap(x_cmap, displayimage, xil_cmap, ilist);
    lu_version = xil_lookup_get_version(xil_cmap);
}
```

The version number of the original XIL colormap was stored in `lu_version`. If this colormap has changed, then `xil_cmap`'s version number has changed, and the program calls `cell_install_cmap()` to update the X colormap. The program then saves `xil_cmap`'s new version number in `lu_version` so that it can check again to see if the colormap has changed after the next call to `xil_decompress()`.

Finally, the example converts the RGB image stored in `image24` to an 8-bit image and copies the 8-bit image to `displayimage`. This is done using the `xil_nearest_color()` function.

```
xil_nearest_color(image24, displayimage, xil_cmap);
```

For each set of RGB values in `image24`, this function performs a pixel-by-pixel search for the nearest matching color in the supplied colormap (`xil_cmap`) and sets the destination image pixel value to the appropriate colormap index.

Note – When a Cell movie is displayed on an 8-bit frame buffer, as in this example, the exact sequence of calls you use to play the movie has a dramatic effect on the speed with which the movie is shown. The reason for this is that by using the deferred-execution scheme explained in Chapter 23, "Acceleration in XIL Programs," the XIL library can look for a certain sequence of functions and, if it finds that sequence, replace all the functions in the sequence with a single highly optimized routine.

In this example, the sequence `xil_decompress()`, `xil_nearest_color()` is such a sequence. Thus, when you play a Cell movie using this program, these functions are not executed, but are replaced by an optimized function (*molecule*) that performs the jobs of both functions. For a complete list of Cell-related molecules, see "Cell Molecules" on page 455.

JPEG Baseline Sequential Codec 16 ≡

The JPEG baseline sequential codec is one of the digital-image codecs that has been specified by the Joint Photographic Experts Group, a joint ISO and CCITT technical committee. The JPEG lossless compressor, discussed in Chapter 17, "JPEG Lossless Codec," is another of these codecs. Taken together, the JPEG codecs provide a standard means of compressing continuous-tone (grayscale and color) still images.

The JPEG standard was originally developed for use in areas such as desktop publishing, graphic arts, medical imaging, and document imaging, where the archiving of still images is important. However, the introduction of high-performance hardware capable of coding and decoding JPEG images in real-time has enabled the development of full-motion video applications based on JPEG.

This chapter has the following main sections:

How the JPEG Baseline Sequential Codec Works	*page 346*
Creating a JPEG Baseline Sequential CIS	*page 350*
JPEG Baseline Sequential Codec Attributes	*page 350*
JPEG Molecules	*page 367*

≡ *16*

How the JPEG Baseline Sequential Codec Works

The JPEG baseline sequential codec is one of the DCT-based compressors, which also include the MPEG-1 and H.261 compressors. Figure 16-1 shows the basic steps the compressor uses to compress an image.
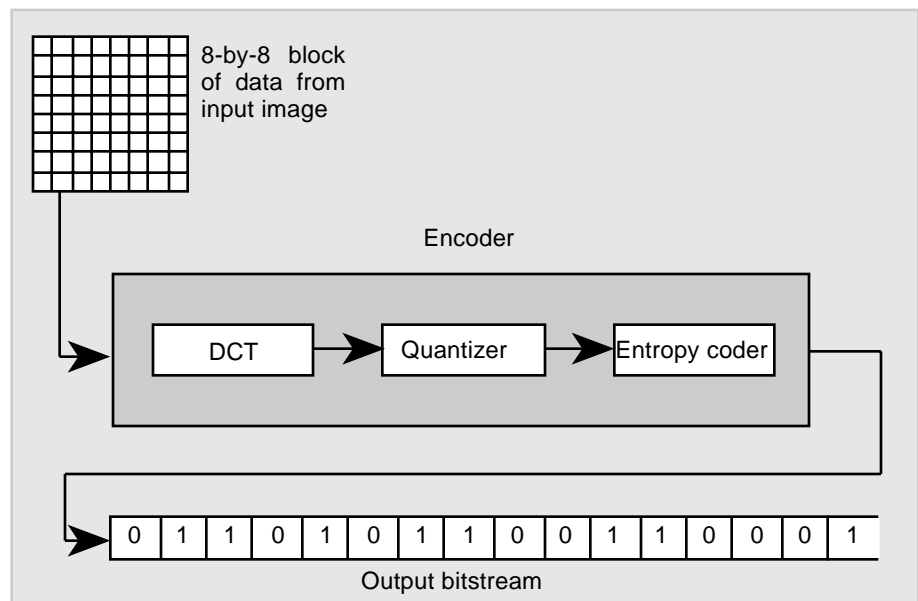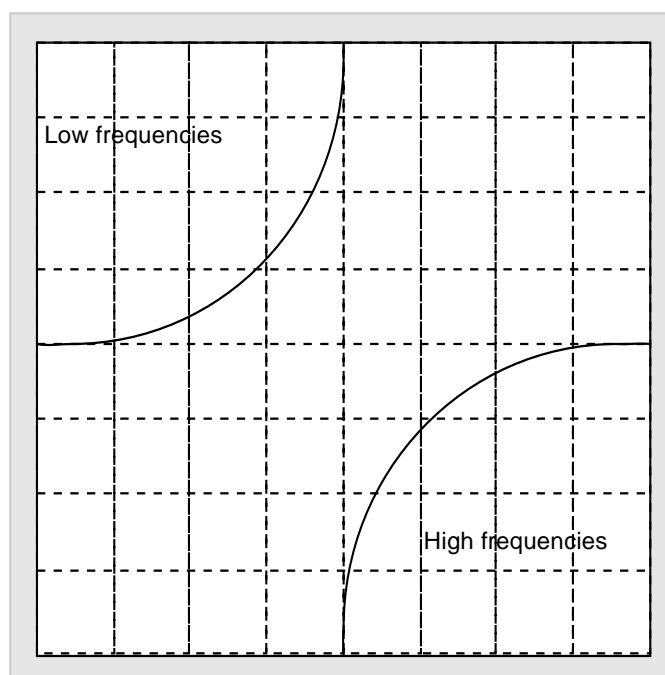

Figure 16-1 JPEG Baseline Sequential Compressor

As the figure indicates, the input to the encoder is an 8-by-8 block of samples from the image being compressed. The compressor encodes each block of data in an image by:

- Performing a Discrete Cosine Transform (DCT) on the 8-by-8 block of data
- Quantizing the results of the DCT
- Entropy coding the results of the quantization step

Each of these steps is discussed in more detail in the following sections.

Discrete Cosine Transform (DCT)

The DCT is a mathematical operation that takes a block of image samples as its input and converts the information in that input from the spatial domain to the frequency domain. For example, in JPEG, the input to the DCT is an 8-by-8 matrix whose values represent brightness levels at particular x, y coordinates, and the output is an 8-by-8 matrix whose values represent relative amounts of the 64 spatial frequencies that make up the input data's spectrum. In the output matrix, information about the lowest frequencies is stored in the upper-left corner, and information about the highest frequencies is stored in the lower-right corner. See Figure 16-2.

Low frequencies

Representation of the input data in the frequency domain

High frequencies

Figure 16-2   Output of the Discrete Cosine Transform

This transformation provides a strong basis for compression because in a typical block of input, low spatial frequencies far outweigh high spatial frequencies. As a result, most of the values in the output matrix, outside of those in the upper-left corner, have values close to 0 and end up not being encoded.

☰ *16*

Quantization

Quantization is the simplest step in the encoder's algorithm. It simply involves dividing the DCT output (each matrix value) by the corresponding value in a quantization table.
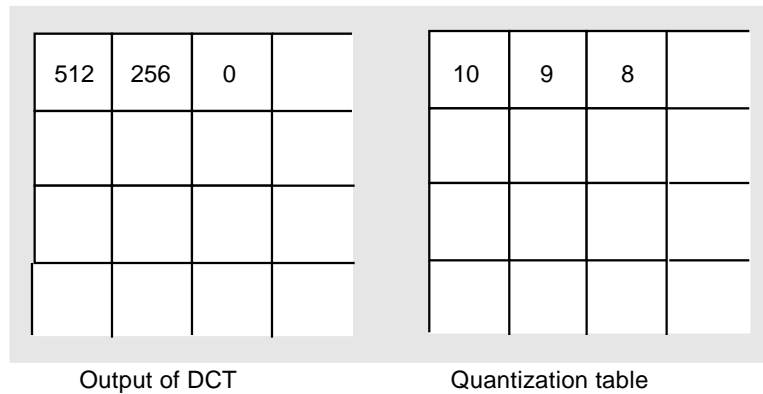
512	256	0	

10	9	8	

Output of DCT   Quantization table

Figure 16-3 Quantization in the JPEG Encoder

Figure 16-3 shows part of an 8-by-8 block of values output by the DCT and part of a table to be used in quantizing this data. In this example, the quantizer divides 512 by 10 and then rounds off 51.2 to 51. Likewise, it divides 256 by 9 and 0 by 8. During quantization, any value in the matrix on the left that is divided by a number greater than itself times 2 goes to 0 and is not encoded in the JPEG bitstream. The quantizer is the part of the JPEG baseline sequential encoder that causes the encoder to be a lossy one.

Entropy Coding

Once a block of data has been quantized, it enters an entropy coder, which creates the actual JPEG bitstream. The entropy coder first looks at the values in the quantized block of data in the zigzag sequence shown in Figure 16-4. Using this pattern produces a sequence in approximately increasing frequency order. This increases the likelihood of long runs of zero-valued quantized coefficients.

Figure 16-4 Zigzag Sequencing in JPEG Encoder

As the entropy encoder moves through the values in this order, it records three pieces of information each time it encounters a nonzero value: the number of 0's it passed over before finding the nonzero value, the number of bits it will take to encode the nonzero value, and the value itself. The first two pieces of information are considered a pair. For example, if the encoder skips over 7 zeros and then finds a 3, the pair would be 7/2 (2 is the number of bits required to encode the value 3). The encoder then consults a Huffman table to find the variable-length bit sequence that represents the pair 7/2 and writes this bit sequence to the JPEG bitstream. It then appends additional bits to represent the actual value 3. The encoder continues this process until all the remaining values in the block are 0's, at which point, it writes a special end-of-block bit sequence to the bitstream.

≡ *16*

Creating a JPEG Baseline Sequential CIS

Before you can use the JPEG codec to compress images or decompress a JPEG bitstream, you must create a JPEG CIS. To do this you pass the compressor name `Jpeg` to the `xil_cis_create()` function. See the code fragment below.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "Jpeg");
```

JPEG Baseline Sequential Codec Attributes

As discussed in "General CIS Attributes" on page 303, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for CISs attached to a JPEG baseline sequential codec. You set codec-specific attributes using the `xil_cis_set_attribute()` function, and you read them using the `xil_cis_get_attribute()` function.

The JPEG baseline sequential attributes can be broadly grouped into those that affect compression and those that affect decompression. The attributes are discussed under these headings below.

Compression Attributes

Setting any of the following attributes affects how the JPEG baseline sequential compressor compresses images. Attributes are identified as read-only (R), write-only(W), or read/write (R/W).

BAND_HUFFMAN_TABLE *(W)*

This attribute enables you to associate a particular Huffman table with a particular band of your image. As shown in Table 16-2 on page 358, the JPEG compressor supports four tables (0 to 3) for both types of DCT coefficients (DC and AC). Table 16-1 shows the default relationship between image bands and Huffman tables.

Table 16-1 Image Bands and Huffman Tables

Table	Type	Band
0	DC	Encodes the DC coefficient in band 0
1	DC	Encodes the DC coefficients in all bands except band 0
2	DC	Not used
3	DC	Not used
0	AC	Encodes the AC coefficients in band 0
1	AC	Encodes the AC coefficients in all bands except band 0
2	AC	Not used
3	AC	Not used

Note – Tables 2 and 3 for both DC and AC coefficients can be used for JPEG extended-baseline-sequential mode encoding. Keep in mind, though, that many JPEG decompressors are unable to decompress the extended-mode bitstream.

Before calling `xil_cis_set_attribute()` to change one of these relationships, you must declare and assign values to the members of a structure of type `XilJpegBandHTable`.

```
typedef struct {
    int band;
    int table;
    XilJpegHTableType type;
} XilJpegBandHTable;
```

The member `band` can have a value in the range 0 to 255 and represents the image band to be associated with a different Huffman table. (The first band in an image is band 0.) The member `table` can be 0 to 3, and the member `type` can be set to `DC` or `AC` (which are enumeration constants of type `XilJpegHTableType`). The setting of `type` determines whether the table you specify will be used to encode DC or AC coefficients in the band you selected.

The code fragment below associates table 0 (type `DC`) with band 1 of the image to be encoded.

```
XilCis cis;
XilJpegBandHTable table_for_band = {1, 0, DC};

xil_cis_set_attribute(cis, "BAND_HUFFMAN_TABLE",
    (void *)&table_for_band);
```

BAND_QUANTIZER *(W)*

The `BAND_QUANTIZER` attribute enables you to specify the quantization table that will be used in encoding a particular band of an image. Before setting this attribute, you must declare and assign values to the members of a structure of type `XilJpegBandQTable`.

```
typedef struct {
    int band;
    int table;
} xilJpegBandQTable;
```

The integer `band` can have a value in the range 0 to 255. In a YC_bC_r image, the Y band is band 0, the C_b band is band 1, and the C_r band is band 2.

The integer `table` can have a value in the range 0 to 3. By default, table 0 contains the values shown in Table K.1 of Annex K of the ISO JPEG specification, table 1 contains the values shown in Table K.2 of Annex K; and tables 2 and 3 are not loaded. You can load tables 2 and 3 or change the values in tables 0 and 1 using the `QUANTIZATION_TABLE` attribute. See "QUANTIZATION_TABLE (W)" on page 363.

By default, band 0 is associated with table 0, and all other bands are associated with table 1. The example code below shows band 2 being associated with table 2.

```
XilCis cis;
XilJpegBandQTable band_to_table;

band_to_table.band = 2;
band_to_table.table = 2;
xil_cis_set_attribute(cis, "BAND_QUANTIZER",
    (void *)&band_to_table);
```

BYTES_PER_FRAME *(R)*

The BYTES_PER_FRAME attribute is a read-only attribute that tells you the number of bytes in the last compressed image written to the CIS. The value of this attribute is 0 if no data has been written to the CIS.

BYTES_PER_FRAME is useful for helping you select an appropriate setting for the COMPRESSION_QUALITY attribute. (See "COMPRESSION_QUALITY (W)" on page 354.) If the number of bytes being used to store a compressed image is too high, you can lower it by lowering the value of COMPRESSION_QUALITY, and vice versa.

The code fragment below reads the value of the BYTES_PER_FRAME attribute.

```
XilCis cis;
int bytes_per_frame;

xil_cis_get_attribute(cis, "BYTES_PER_FRAME",
    (void **)&bytes_per_frame);
```

COMPRESSED_DATA_FORMAT *(W)*

The COMPRESSED_DATA_FORMAT attribute is a set-only attribute that defines the format of the JPEG compressor's output. It can be set to either INTERCHANGE or ABBREVIATED_FORMAT.

Setting the attribute to `INTERCHANGE` causes the compressor to produce output in JPEG interchange format. In this format, the quantization tables and Huffman tables required by the decompressor are included in each compressed frame.

The default value, `ABBREVIATED_FORMAT`, causes the compressor to produce output in JPEG abbreviated format. In this format, quantization tables and Huffman tables are not included in a compressed frame if the tables needed to decompress that frame have been defined in a previous frame in the sequence. If a table changes during the course of the sequence, the new table definition is included in the first compressed frame that uses the new table values.

Note – The compressor does not enable you to produce abbreviated-table output in which frames contain only table specifications. However, the decompressor accepts bitstreams in this format.

The code fragment below shows `COMPRESSED_DATA_FORMAT` being set to `INTERCHANGE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSED_DATA_FORMAT",
    (void *)INTERCHANGE);
```

COMPRESSION_QUALITY *(W)*

The `COMPRESSION_QUALITY` attribute tells the compressor how it should handle the trade-off between image quality and compression ratio. The attribute can be set to any value between 1 and 100. Setting the attribute to 100 is a request that the compressor produce very high quality images, even though this means a lower compression ratio. A setting of 1 tells the compressor to increase its compression ratio, even though the result is lower image quality. By default, this attribute is set to 50.

Studies by the JPEG committee indicate that for color images of average complexity, the following relationships exist between level of compression and image quality:

- 0.25-0.5 bits/pixel: moderate to good quality, sufficient for some applications

- 0.5-0.75 bits/pixel: good to very good quality, sufficient for many applications

- 0.75-1.5 bits/pixel: excellent quality, sufficient for most applications

- 1.5-2.0 bits/pixel: usually indistinguishable from the original, sufficient for the most demanding applications[1]

Also, see Color Plate 4. It shows an image compressed at a high-quality setting (95) and one compressed at a low-quality setting (5).

The setting of the `COMPRESSION_QUALITY` attribute controls image quality and the compression ratio by determining a scaling factor the compressor uses in creating scaled versions of its quantization tables. These scaled tables are used during compression. A `COMPRESSION_QUALITY` value of 50 results in a scaling factor of 1; that is, the scaled tables are identical to the original tables. A `COMPRESSION_QUALITY` value greater than 50 results in a scaling factor that is less than 1, and a value less than 50 results in a scaling factor greater than 1.

Note – The trade-off between image quality and compression ratio can also be affected by the values stored in the compressor's quantization tables. See "QUANTIZATION_TABLE (W)" on page 363.

The code fragment below shows the `COMPRESSION_QUALITY` attribute being set to 100.

```
XilCis cis;
int quality = 100;

xil_cis_set_attribute(cis, "COMPRESSION_QUALITY",
    (void *)quality);
```

---

1. G. K. Wallace, "The JPEG Still Picture Compression Standard," *Communications of the ACM*, April 1991, p. 35.

`ENCODE_INTERLEAVED` *(W)*

The `ENCODE_INTERLEAVED` attribute is a set-only attribute that determines whether the compressor produces an interleaved bitstream when compressing multiband images. If the attribute is set to `TRUE`, the default value, the compressor produces an interleaved bitstream, and if the attribute is set to `FALSE`, the compressor produces a noninterleaved bitstream. Interleaving refers to how the DCT blocks from the different bands are ordered. In a noninterleaved bitstream, all blocks from band 1 precede blocks from band 2.

The `ENCODE_INTERLEAVED` attribute is ignored if the image being compressed has more than four bands, because the bitstream for such images is never interleaved. Also, note that the `ENCODE_411_INTERLEAVED` attribute takes precedence over this attribute if the image being compressed has three bands.

In the example below, `ENCODE_INTERLEAVED` is being set to `FALSE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "ENCODE_INTERLEAVED", (void *)FALSE);
```

`ENCODE_411_INTERLEAVED` *(W)*

The `ENCODE_411_INTERLEAVED` attribute is a set-only attribute that affects how the compressor handles 3-band images and is intended specifically for use with YC_bC_r images (not RGB images). If the input to the compressor is an image of a different type, this attribute should be set to `FALSE` (its default value).

Note – If the input images are not 3-band images and this attribute is set to `TRUE`, the compressor operates as if the attribute were set to `FALSE`.

If the input is a 3 band image (presumably YC_bC_r) and `ENCODE_411_INTERLEAVED` is set to `TRUE`, the compressor:

- Subsamples the data in the color bands so that only one color value is encoded for each four color values in the original (4:1:1)

- Interleaves the data for the three image bands on a macroblock basis, where a macroblock is defined as a 16-by-16 block of pixels

---

Note – This is the only method the XIL library provides for producing JPEG bitstreams with subsampled bands.

If you are making a movie from YC_bC_r images, `ENCODE_411_INTERLEAVED` should be set to `TRUE` because the XIL library's accelerated routines for playing back JPEG movies require input in the format just described. The `ENCODE_411_INTERLEAVED` attribute tells the JPEG compressor to subsample the C_b and C_r data instead of using the evenly sampled bands that the XIL software produces. The output of a decompression (and later export) is still an evenly sampled 3-banded image with the C_b and C_r duplicated.

If the attribute is set to `FALSE`, the setting of the `ENCODE_INTERLEAVED` attribute determines the format of the JPEG bitstream.

In the following example, `ENCODE_411_INTERLEAVED` is being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "ENCODE_411_INTERLEAVED",
    (void *)TRUE);
```

HUFFMAN_TABLE *(W)*

The `HUFFMAN_TABLE` attribute enables you to supply the compressor with a Huffman table to use in encoding quantized DC or AC coefficients. Before calling `xil_cis_set_attribute()` to set this attribute, you must store information about the Huffman table in a structure of type `XilJpegHTable`.

```
typedef struct {
    int table;
    XilJpegHTableType type;
    XilJpegHTableValue *value;
} XilJpegHTable;
```

The `table` member must be set to 0 to 3, and the `type` member must be set to `DC` or `AC`. In a Huffman table of type `DC`, the array `value` contains 16 elements. In a table of type `AC`, this array contains 256 elements.

The default contents of the eight Huffman tables are shown in Table 16-2.

Table 16-2 Default Huffman Tables

Table	Type	Default Contents
0	DC	Contains the values specified in Table K.3 of the ISO JPEG specification. These values are useful for encoding the DC coefficients of the luminance band of 8-bit YC_bC_r images.
1	DC	Contains the values specified in Table K.4 of the ISO JPEG specification. These values are useful for encoding the DC coefficients of the chrominance bands of 8-bit YC_bC_r images.
2	DC	Empty.
3	DC	Empty.
0	AC	Contains the values specified in Table K.5 of the ISO JPEG specification. These values are useful for encoding the AC coefficients of the luminance band of 8-bit YC_bC_r images.
1	AC	Contains the values specified in Table K.6 of the ISO JPEG specification. These values are useful for encoding the AC coefficients of the chrominance bands of 8-bit YC_bC_r images.
2	AC	Empty.
3	AC	Empty.

Note – Tables 2 and 3, for both DC and AC coefficients, can be used for JPEG extended-baseline-sequential mode encoding. Keep in mind, though, that many JPEG decompressors are unable to decompress the extended-mode bitstream.

The final member of the XilJpegHTable structure, value, is an array of structures of type XilJpegHTableValue.

```
typedef struct {
    int bits;
    int pattern;
} XilJpegHTableValue;
```

Each structure of this type defines a pair of values to be written to the Huffman table. The `bits` member is the length of a Huffman code in bits, and `pattern` contains the code itself (the `bits` least significant bits of `pattern` are the code).

The examples below indicate how the structures referred to above are used in setting up a Huffman table. This first example shows table 0 (type `DC`) being loaded with values suitable for encoding DC coefficients. Entry `huffman_values[k]` is the huffman code for DC size category *k*.

```
XilCis cis;
XilJpegHTable huffman_table;

XilJpegHTableValue huffman_values[16] = {
    {2, 0x0000}, {3, 0x0002}, {3, 0x0003}, {3, 0x0004},
    {3, 0x0005}, {3, 0x0006}, {4, 0x000e}, {5, 0x001e},
    {6, 0x003e}, {7, 0x007e}, {8, 0x00fe}, {9, 0x01fe},
    {0, 0x0000}, {0, 0x0000}, {0, 0x0000}, {0, 0x0000}
};

huffman_table.table = 0;
huffman_table.type = DC;
huffman_table.value = huffman_values;
xil_cis_set_attribute(cis, "HUFFMAN_TABLE",
    (void *)&huffman_table);
```

Note that there are only twelve meaningful value pairs being loaded into the table. The last four are there to fill out `huffman_table.value`, which is an array of sixteen structures.

This next example shows table 0 (type `AC`) being loaded with values suitable for encoding AC coefficients. Entry `huffman_values[k]` is the huffman code for *k* = run_length * 16 + (AC size category).

```
XilCis cis;
XilJpegHTable huffman_table;

XilJpegHTableValue huffman_values[256] = {
    {4, 0x000a}, {2, 0x0000}, {2, 0x0001}, {3, 0x0004},
    {4, 0x000b}, {5, 0x001a}, {7, 0x0078}, {8, 0x00f8},
    {10, 0x03f6}, {16, 0xff82}, {16, 0xff83}, {0, 0x0000},
    {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},
```

```
{0, 0x0000}, {4, 0x000c}, {5, 0x001b}, {7, 0x0079},
{9, 0x01f6}, {11, 0x07f6}, {16, 0xff84}, {16, 0xff85},
{16, 0xff86}, {16, 0xff87}, {16, 0xff88}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {5, 0x001c}, {8, 0x00f9}, {10, 0x03f7},
{12, 0x0ff4}, {16, 0xff89}, {16, 0xff8a}, {16, 0xff8b},
{16, 0xff8c}, {16, 0xff8d}, {16, 0xff8e}, {0, 0x0000},
{0, 0x0000},{0, 0x000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {6, 0x003a}, {9, 0x01f7}, {12, 0x0ff5},
{16, 0xff8f}, {16, 0xff90}, {16, 0xff91}, {16, 0xff92},
{16, 0xff93}, {16, 0xff94}, {16, 0xff95}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {6, 0x003b}, {10, 0x03f8}, {16, 0xff96},
{16, 0xff97}, {16, 0xff98}, {16, 0xff99}, {16, 0xff9a},
{16, 0xff9b}, {16, 0xff9c}, {16, 0xff9d}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},


{0, 0x0000}, {7, 0x007a}, {11, 0x07f7}, {16, 0xff9e},
{16, 0xff9f}, {16, 0xffa0}, {16, 0xffa1}, {16, 0xffa2},
{16, 0xffa3}, {16, 0xffa4}, {16, 0xffa5}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {7, 0x007b}, {12, 0x0ff6}, {16, 0xffa6},
{16, 0xffa7}, {16, 0xffa8}, {16, 0xffa9}, {16, 0xffaa},
{16, 0xffab}, {16, 0xffac}, {16, 0xffad}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {8, 0x00fa}, {12, 0x0ff7}, {16, 0xffae},
{16, 0xffaf}, {16, 0xffb0}, {16, 0xffb1}, {16, 0xffb2},
{16, 0xffb3}, {16, 0xffb4}, {16, 0xffb5}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {9, 0x01f8}, {15, 0x7fc0}, {16, 0xffb6},
{16, 0xffb7}, {16, 0xffb8}, {16, 0xffb9}, {16, 0xffba},
{16, 0xffbb}, {16, 0xffbc}, {16, 0xffbd}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

{0, 0x0000}, {9, 0x01f9}, {16, 0xffbe}, {16, 0xffbf},
{16, 0xffc0}, {16, 0xffc1}, {16, 0xffc2}, {16, 0xffc3},
{16, 0xffc4}, {16, 0xffc5}, {16, 0xffc6}, {0, 0x0000},
{0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},
```

```
     {0, 0x0000}, {9, 0x01fa}, {16, 0xffc7}, {16, 0xffc8},
     {16, 0xffc9}, {16, 0xffca}, {16, 0xffcb}, {16, 0xffcc},
     {16, 0xffcd}, {16, 0xffce}, {16, 0xffcf}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

     {0, 0x0000}, {10, 0x03f9}, {16, 0xffd0}, {16, 0xffd1},
     {16, 0xffd2}, {16, 0xffd3}, {16, 0xffd4}, {16, 0xffd5},
     {16, 0xffd6}, {16, 0xffd7}, {16, 0xffd8}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

     {0, 0x0000}, {10, 0x03fa}, {16, 0xffd9}, {16, 0xffda},
     {16, 0xffdb}, {16, 0xffdc}, {16, 0xffdd}, {16, 0xffde},
     {16, 0xffdf}, {16, 0xffe0}, {16, 0xffe1}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

     {0, 0x0000}, {11, 0x07f8}, {16, 0xffe2}, {16, 0xffe3},
     {16, 0xffe4}, {16, 0xffe5}, {16, 0xffe6}, {16, 0xffe7},
     {16, 0xffe8}, {16, 0xffe9}, {16, 0xffea}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

     {0, 0x0000}, {16, 0xffeb}, {16, 0xffec}, {16, 0xffed},
     {16, 0xffee}, {16, 0xffef}, {16, 0xfff0}, {16, 0xfff1},
     {16, 0xfff2}, {16, 0xfff3}, {16, 0xfff4}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000},

     {11, 0x07f9}, {16, 0xfff5}, {16, 0xfff6}, {16, 0xfff7},
     {16, 0xfff8}, {16, 0xfff9}, {16, 0xfffa}, {16, 0xfffb},
     {16, 0xfffc}, {16, 0xfffd}, {16, 0xfffe}, {0, 0x0000},
     {0, 0x0000},{0, 0x0000},{0, 0x0000},{0, 0x0000}
};

huffman_table.table = 0;
huffman_table.type = AC;
huffman_table.value = huffman_values;
xil_cis_set_attribute(cis, "HUFFMAN_TABLE",
     (void *)&huffman_table);
```

Note that `huffman_values` is an array of 256 structures of type `XilJpegHTableValue`, and that the structures are organized as 16 groups of 16 structures. Because there are not 16 bit lengths (`bits`) for each of the 16 run lengths, each group of 16 structures contains 5 meaningless structures at the end.

OPTIMIZE_HUFFMAN_TABLES *(W)*

You can use the OPTIMIZE_HUFFMAN_TABLE attribute to tell the compressor to generate optimal Huffman tables instead of using the default tables included in the ISO specification or tables you loaded earlier. When the compressor creates optimal Huffman tables, its Huffman tables can vary from image to image, and you should see higher rates of compression.

If OPTIMIZE_HUFFMAN_TABLES is set to FALSE, its default value, the compressor uses fixed Huffman tables for each image in a sequence. If the attribute is set to TRUE, the compressor uses optimal Huffman tables.

Note – If an application sets OPTIMIZE_HUFFMAN_TABLES to TRUE and then later sets it to FALSE, the compressor loads a default set of tables. It does not restore the tables it was using when the attribute was first set to TRUE.

The following code fragment shows OPTIMIZE_HUFFMAN_TABLES being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis, "OPTIMIZE_HUFFMAN_TABLES",
    (void *)TRUE);
```

Note – Setting this attribute to TRUE will cause increased compression time and memory usage, as a two-pass appproach is required.

QUANTIZATION_TABLE *(W)*

The `QUANTIZATION_TABLE` attribute enables you to load values into one of four quantization tables that the compressor may use to quantize DCT coefficients. Prior to actually setting the attribute you must declare and assign values to the members of an `XilJpegQTable` structure.

```
typedef struct {
    int table;
    int (*value)[8];
} XilJpegQTable;
```

The integer `table` must have a value between 0 and 3. This value indicates which of the four quantization tables you want to load. By default, table 0 contains the values shown in Table K.1 of Annex K of the ISO JPEG specification. These values are designed to be used with the luminance band of 8-bit YCC images. Likewise, table 1 contains the values shown in Table K.2 of Annex K. The values in this table are designed to be used with the chrominance bands of 8-bit YCC images. By default, tables 2 and 3 are not loaded.

You use the pointer `value` (a pointer to an array of 8 integers) to point to the 64 quantization values that you want to load into the table.

The code fragment below illustrates 64 quantization values being loaded into table 2.

```
Xilcis cis;
XilJpegQTable quantization;
int quantization_array[8][8] = {
    {16, 11, 10, 16, 24, 40, 51, 61},
    {12, 12, 14, 19, 26, 58, 60, 55),
    {14, 13, 16, 24, 40, 57, 69, 56},
    {14, 17, 22, 29, 51, 87, 80, 62},
    {18, 22, 37, 56, 68, 109, 103, 77},
    {24, 35, 55, 64, 81, 104, 113, 92},
    {49, 64, 78, 87, 103, 121, 120, 101},
    {72, 92, 95, 98, 112, 100, 103, 99}
};

quantization.table = 2;
quantization.value = quantization_array;
xil_cis_set_attribute(cis, "QUANTIZATION_TABLE",
    (void *)&quantization);
```

TEMPORAL_FILTERING *(R/W)*

The TEMPORAL_FILTERING attribute controls whether the JPEG compressor filters 3-band images before it encodes them. If the attribute is set to FALSE, its default value, the compressor does not filter the images, and if it is set to TRUE, the compressor does filter them. This filtering can help in cases where the images being encoded contain a lot of noise; however, the filtering can also have the undesirable side effect of producing ghosting artifacts.

The example below shows the attribute being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis, "TEMPORAL_FILTERING", (void *)TRUE);
```

And this fragment demonstrates how to read the attribute's value.

```
XilCis cis;
Xil_boolean filtering_enabled;

xil_cis_get_attribute(cis, "TEMPORAL_FILTERING",
    (void **)&filtering_enabled);
```

Decompression Attributes

Setting the following attribute affects how the JPEG baseline sequential codec decompresses images.

DECOMPRESSION_QUALITY *(W)*

You use the DECOMPRESSION_QUALITY attribute to provide a hint to the compressor concerning how it should handle the trade-off between the quality of decompressed images and the speed of decompression. The decompressor is free to ignore this hint.

The attribute's value must be an integer in the range of 1 to 100. A setting of 100 is a request for a high level of image quality, and a setting of 1 is a request from a high playback speed. By default, the attribute is set to 100.

Note – The JPEG compressor increases playback speed by decreasing the number of quantized coefficients it uses in reconstructing an image. It drops high-frequency coefficients first.

The example code below shows DECOMPRESSION_QUALITY being set to 75.

```
XilCis cis;
int quality = 75;

xil_cis_set_attribute(cis, "DECOMPRESSION_QUALITY",
    (void *)quality);
```

`IGNORE_HISTORY` *(W)*

The setting of the `IGNORE_HISTORY` attribute affects what happens when you seek backward or forward in a JPEG CIS whose random-access flag is not set. If your CIS's random-access flag is set, the value of `IGNORE_HISTORY` is irrelevant: you can always seek backward or forward and get correct results. To determine whether the random-access flag is set, call the `xil_cis_get_random_access()` function.

Note – When you create a JPEG CIS, its random-access flag is always set. This flag remains set if the quantization and Huffman tables needed to decode the entire JPEG bitstream are encoded with the first image or if those tables are encoded with each image in the sequence. Otherwise, the decoder clears the flag as soon as it discovers that the bitstream cannot be accessed randomly.

If your CIS's random-access bit is not set and `IGNORE_HISTORY` is `FALSE` (its default value), the following rules apply:

- Seeking backward is illegal

- Seeking forward is successful; that is, the image you decode after the seek will be reconstructed properly. However, seeking forward with `IGNORE_HISTORY` set to `FALSE` may be slower than one with the attribute set to `TRUE`.

If your CIS's random-access bit is not set and `IGNORE_HISTORY` is `TRUE`, these rules apply:

- Seeking backward is legal. However, it is the responsibility of your application to seek to an image that can be decoded correctly. That is, the image you seek to must either define its own tables or depend on the tables that were most recently loaded into the decoder.

- You can also seek forward. However, your application should not seek forward past images that contain table definitions if those definitions are needed to decode the image you're seeking to. The decoder does not ensure that these table definitions are loaded.

The code below shows `IGNORE_HISTORY` being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "IGNORE_HISTORY", (void *)TRUE);
```

JPEG Molecules

The XIL library includes a series of *molecules* that accelerate the playback of JPEG baseline sequential bitstreams. These molecules are optimized routines that perform the jobs of two or more functions from the XIL API. You do not call such an optimized routine directly; rather, the library calls a molecule when your program calls a predefined sequence of XIL functions, sometimes with specific arguments.

For example, if your program calls `xil_decompress()` to decode an image stored in a JPEG CIS and then calls `xil_ordered_dither()` to convert the decoded image from a 24- to an 8-bit image, the library may not call these two functions. Instead, it may call a molecule that performs the decompression and the dithering in an optimized way.

For more information about accelerating the playback of JPEG bitstreams, see Chapter 23, "Acceleration in XIL Programs." For information about the JPEG molecules that are available and information about how to call those molecules, see "XIL Molecules" on page 477.

≡ *16*

JPEG Lossless Codec 17≡

Like the JPEG baseline sequential compressor discussed in Chapter 16, "JPEG Baseline Sequential Codec, the JPEG lossless compressor is designed to compress continuous-tone still images. The difference between the two is that the baseline sequential compressor is a lossy compressor: once an image has been compressed, you cannot recover the original image samples. The JPEG lossless compressor does allow you to recover these samples; however, the lossless compressor produces much lower compression ratios than the lossy compressor. The compression ratio for the lossless compressor is about 2:1.

One other difference between the two compressors is that while the baseline sequential compressor works only with `XIL_BYTE` images, the lossless compressor can work with both `XIL_BYTE` and `XIL_SHORT` images.

This chapter has the following main sections.

How the JPEG Lossless Codec Works

The JPEG lossless compressor is not based on the DCT like the baseline sequential encoder. Instead, it uses a predictive method, as shown in Figure 17-1.

Figure 17-1 JPEG Lossless Compressor

Each sample in the source image is encoded as follows: the predictor makes a guess at the sample's value based on its knowledge of the values of neighboring samples and then subtracts the predicted value of the sample from its actual value. The difference calculated by the predictor is then passed on to the entropy coder, which does a lookup in a Huffman table and writes encoded data to the bitstream.

The prediction and entropy-coding steps in the encoding process are discussed in more detail in the following sections.

Note – The XIL library's JPEG lossless encoder lets you perform an operation called a point transformation on your source image before the encoding begins. The point transformation can be done on a band-by-band basis. If you supply a point-transform value other than 0 for a band of an image, each sample in the band is divided by 2 raised to the power *x*, where *x* is the point-transform

value. This operation leads to a higher compression ratio because smaller values will need to be encoded; however, it is not actually part of the lossless-encoding process because it can result in the loss of some data.

Prediction

The predictor predicts the value of an image sample based on its knowledge of the values of neighboring samples. The neighboring samples it can take into consideration are shown in Figure 17-2 and are labeled A, B, and C. The sample whose value is being predicted is labeled P.

Figure 17-2 Predicting Values in the JPEG Lossless Compressor

Seven prediction methods are defined. These are shown below:

1. P = A

2. P = B

3. P = C

4. P = A + B - C

5. P = A + ((B - C) $/$ *2)*

6. P = B + ((A - C) $/$ *2)*

7. P = (A + B) $/$ 2

By default, the compressor uses prediction method 1 for all bands in an image. However, you can select a different prediction method for a particular band using the CIS attribute, `LOSSLESS_BAND_SELECTOR`. See "LOSSLESS_BAND_SELECTOR (W)" on page 376.

After the predictor has predicted the value of a sample, it subtracts the predicted value of the sample from its actual value. It is this difference that is encoded by the entropy coder.

Entropy Coding

The entropy coder takes as input a difference, the difference between a predicted and an actual sample value. On the output side, the entropy coder writes a sequence of bits to the JPEG bitstream.

To produce its output, the entropy coder uses the number of bits (k) required to represent a difference as an index into a Huffman table. At the proper location in the table, the encoder reads a code word—a sequence of bits—and then writes this code word to the bitstream. Immediately after the code word, the encoder writes to the bitstream the difference itself. The encoder performs these steps for each sample in the image (modulo 2^{k-1}).

JPEG Lossless Compressor Attributes

The attributes discussed in this section affect the way that the JPEG lossless compressor works. Attributes are identified as read-only(R), write-only(W), or read/write (R/W).

BAND_HUFFMAN_TABLE *(W)*

With the `BAND_HUFFMAN_TABLE` attribute you can specify that a particular Huffman table be used in encoding a particular band of your image. Before calling `xil_cis_set_attribute()` to set this attribute, you must declare and assign values to the members of a structure of type `XilJpegBandHTable`.

```
typedef struct {
    int band;
    int table;
    XilJpegHTableType type;
} XilJpegBandHTable;
```

The `band` member can have a value in the range 0 to 255 and represents the image band to be associated with a particular Huffman table. The `table` member can have a value between 0 and 3 and specifies one of four possible Huffman tables, and the `type` member must be set to `DC`.

The following code fragment associates table 0 (type `DC`) with band 1 of the image to be encoded.

```
XilCis cis;
XilJpegBandHTable table_for_band = {1, 0, DC};

xil_cis_set_attribute(cis, "BAND_HUFFMAN_TABLE",
    (void *)&table_for_band);
```

By default, table 0 is used in encoding band 0 of an image, and table 1 is used in encoding all other bands (1 to 255).

COMPRESSED_DATA_FORMAT *(W)*

The COMPRESSED_DATA_FORMAT attribute is a set-only attribute that defines the format of the JPEG lossless compressor's output. It can be set to either INTERCHANGE or ABBREVIATED_IMAGE.

Setting the attribute to INTERCHANGE causes the compressor to produce output in JPEG interchange format. In this format, the Huffman tables required by the decompressor are included in each compressed frame.

The default value, ABBREVIATED_IMAGE, causes the compressor to produce output in JPEG abbreviated format. In this format, Huffman tables are not included in a compressed frame if the tables needed to decompress that frame have been defined in a previous frame in the sequence. If a table changes during the course of the sequence, the new table definition is included in the first compressed frame that uses the new table values.

The code fragment below shows COMPRESSED_DATA_FORMAT being set to INTERCHANGE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSED_DATA_FORMAT",
    (void *)INTERCHANGE);
```

ENCODE_INTERLEAVED *(W)*

The ENCODE_INTERLEAVED attribute is a set-only attribute that determines whether the compressor produces an interleaved bitstream when compressing multiband images. If the attribute is set to TRUE, the default value, the compressor produces an interleaved bitstream, and if the attribute is set to FALSE, the compressor produces a noninterleaved bitstream. Band interleaving is performed on a pixel-by-pixel basis.

This attribute is ignored if the image being compressed has more than four bands, or if different point transformations or different predictors are being used in encoding different bands. In these instances, the compressor never produces an interleaved bitstream.

In this example, `ENCODE_INTERLEAVED` is being set to `FALSE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "ENCODE_INTERLEAVED", (void *)FALSE);
```

HUFFMAN_TABLE *(W)*

With the `HUFFMAN_TABLE` attribute you can supply the compressor with a Huffman table to use in encoding difference values. By default, the compressor's four Huffman tables are loaded as follows: Tables 0 and 2 contain the values specified in Table K.3 of the ISO JPEG specification (extended to 17 entries). These values are useful for encoding pixel differences for the luminance band of 8-bit YC_bC_r images. Tables 1 and 3 contain the values specified in Table K.4 of the ISO JPEG specification (again extended to 17 entries). These values are useful for encoding pixel differences for the chrominance bands of 8-bit YC_bC_r images.

Before calling `xil_cis_set_attribute()` to set this attribute, you must store information about the Huffman table in a structure of type `XilJpegHTable`.

```
typedef struct {
    int table;
    XilJpegHTableType type;
    XilJpegHTableValue *value;
} XilJpegHTable;
```

The `table` member must be set to a value between 0 and 3, and the `type` member must be set to `DC`. The final member of the `XilJpegHTable` structure, `value`, is an array of 17 structures of type `XilJpegHTableValue`.

```
typedef struct {
    int bits;
    int pattern;
} XilJpegHTableValue;
```

Each structure of this type defines a pair of values to be written to the Huffman table. The `bits` member is the length of a Huffman code in bits, and `pattern` contains the code itself (the `bits` least significant bits of `pattern` are the code).

This example shows table 0 (type `DC`) being loaded with values suitable for encoding difference values.

```
XilCis cis;
XilJpegHTable huffman_table;

XilJpegHTableValue huffman_values[17] = {
    {2, 0x0000}, {3, 0x0002}, {3, 0x0003}, {3, 0x0004},
    {3, 0x0005}, {3, 0x0006}, {4, 0x000e}, {5, 0x001e},
    {6, 0x003e}, {7, 0x007e}, {8, 0x00fe}, {9, 0x01fe},
    {10, 0x03fe},{11, 0x07fe}, {12, 0x0ffe}, {13, 0x1ffe},
    {14, 0x3ffe}
};

huffman_table.table = 0;
huffman_table.type = DC;
huffman_table.value = huffman_values;
xil_cis_set_attribute(cis, "HUFFMAN_TABLE",
    (void *)&huffman_table);
```

LOSSLESS_BAND_SELECTOR *(W)*

As mentioned in "Prediction" on page 371, there are seven methods the predictor can use to predict the values of an image sample. This attribute enables you to specify which of these methods the compressor should use in encoding a particular band of an image.

Before calling `xil_cis_set_attribute()` to set this attribute, you must declare and assign values to the members of a structure of type `XilJpegLLBandSelector`.

```
typedef struct {
    int band;
    XilJpegLLBandSelectorType selector;
} XilJpegLLBandSelector;
```

The `band` structure member must be set to a value between 0 and 255 (where band 0 is the first band in an image). The `selector` member can have any of the values shown in the following enumeration.

```
typedef enum {
    ONE_D1=1, ONE_D2, ONE_D3, TWO_D1, TWO_D2, TWO_D3, TWO_D4
} XilJpegLLBandSelectorType;
```

The correspondence between these enumeration constants and the prediction methods described earlier is shown in Table 17-1.

Table 17-1 JPEG Lossless Prediction Methods

Constant	Prediction Method
ONE_D1	P = A
ONE_D2	P = B
ONE_D3	P = C
TWO_D1	P = A + B - C
TWO_D2	P = A + ((B - C) / *2)*
TWO_D3	P = B + ((A - C) / *2)*
TWO_D4	P = (A + B) / 2

The following code fragment shows the prediction formula
P = A + ((B - C) / *2)* being selected for band 2 of an image.

```
XilCis cis;
XilJpegLLBandSelector predictor_band = {2, TWO_D2};

xil_cis_set_attribute(cis, "LOSSLESS_BAND_SELECTOR",
    (void *)&predictor_band);
```

LOSSLESS_BAND_PT_TRANSFORM *(W)*

With the `LOSSLESS_BAND_PT_TRANSFORM` attribute you can request that a point transformation be performed on a particular band of an image before the image is encoded. If you request this operation, each sample in the band is divided by 2 raised to power of *x*, where *x* is the prediction-transformation value. The default prediction-transformation value for all bands is 0.

Before calling `xil_cis_set_attribute()` to set this attribute, you must declare and assign values to the members of a structure of type `XilJpegLLBandPtTransform`.

```
typedef struct {
    int band;
    int PtTransform;
} XilJpegLLBandPtTransform;
```

The `band` structure member must be set to a value between 0 and 255 (where band 0 is the first band in an image), and the `PtTransform` member must be set to an integer in the range 0 to 15 that represents the power of two by which you want to divide all the samples in a band.

The code below requests that the samples in band 2 of an image be divided by 16 (2^4) before the lossless encoding process takes place.

```
XilCis cis;
XilJpegLLBandPtTransform pt_transform = {2, 4};

xil_cis_set_attribute(cis, "LOSSLESS_BAND_PT_TRANSFORM",
    (void *)&pt_transform);
```

H.261 Codec 18 ≡

The H.261 codec takes its name from the title of the recommendation in which the H.261 codec is specified: Recommendation H.261 published by the CCITT. This recommendation defines a video encoder that is intended to be used to compress video data sent over Integrated Services Digital Network (ISDN) lines.

The H.261 codec is intended primarily for use in video telephony and videoconferencing applications. Video telephony, in which generally a picture of the speaker's face against a stationary background is transmitted, is possible when only one or two ISDN channels (each capable of carrying 64Kb/s of information) are available. If more channels are available, more complex images can be sent.

Note – This chapter discusses both H.261 compression and decompression. However, the current release of the XIL library includes only an H.261 decompressor. The compressor interface is defined for third parties who want to implement XIL H.261 compressors.

This chapter has the following main sections:

How an H.261 Codec Works	*page 380*
Creating an H.261 CIS	*page 388*
H.261 Codec Attributes	*page 389*
H.261 Molecules (accelerating the playback of H.261 bitstreams)	*page 400*

$\equiv$ *18*

How an H.261 Codec Works

This section presents an overview of how an H.261 codec works. It discusses:

- The format of the images that can be used as input to the encoder
- The basic encoding scheme
- Methods of controlling the size of the bitstream produced by the encoder
- How the codec supports multipoint conferencing

Source Images

The images supplied as input to an H.261 compressor must meet both color space and size (width and height) requirements. In terms of color space, the images must be YC_bC_r images that conform to the standard set forth in CCIR Recommendation 601. In terms of size, the images must adhere to either the Common Interchange Format (CIF) or the Quarter-CIF (QCIF) format. Table 18-1 indicates the widths and heights defined by these formats.

Table 18-1 Sizes of CIF- and QCIF-Format Images

	Width	Height
CIF images	352	288
QCIF images	176	144

All H.261 encoders must be able to compress QCIF images. The ability to compress CIF images is optional.

Given an image of the appropriate format, the H.261 compressor subsamples the chrominance values so that there is one C_b value and one C_r value for each two-by-two block of luminance values. It then processes the image in segments called macroblocks. Each macroblock consists of a 16-by-16 block of luminance values and the chrominance values associated with those luminance values; see Figure 18-1.

Figure 18-1 Macroblock

The encoder performs some operations on 8-by-8 blocks of values. Each macroblock contains six blocks: four blocks of luminance values, one block of C_b values, and one block of C_r values.

≡ *18*

Basic Encoding Scheme

The flow chart in Figure 18-2 illustrates the procedure the H.261 encoder uses to encode a macroblock.

Figure 18-2 Flow Diagram for H.261 Encoding

Intraframe Versus Interframe Encoding

For each macroblock that it encodes, the H.261 encoder can perform intraframe or interframe compression. In intraframe mode, the compressor encodes the actual YC_bC_r values in the macroblock. In interframe mode, the compressor:

- Looks at the YC_bC_r values in the macroblock it is encoding.

- Calculates the difference between the predicted values for the macroblock and the actual values in the macroblock. The predicted values are taken from the most recently compressed image, which is stored in a history buffer.

- Encodes the difference values if they are significant.

In general, the H.261 compressor relies very heavily on interframe encoding because this type of encoding leads to greater rates of compression than intraframe compression. However, Recommendation H.261 requires that the encoder intraframe encode each macroblock at least once every 132 frames. This requirement ensures that if you join a videoconference in progress or your videoconference is disrupted by data transmission problems, all macroblocks will be updated properly within a few seconds.

Prediction

As noted in the last section, in interframe-encoding mode, the encoder calculates the difference between YC_bC_r values in the macroblock it is encoding currently and the values in the corresponding macroblock in the preceding picture. Before performing this calculation, the encoder may perform either a motion-compensation operation or a motion-compensation operation followed by a loop-filter operation. Both of these operations are optional.

When the encoder performs motion compensation, it compares the YC_bC_r values in the current macroblock not only with those in the spatially corresponding macroblock in the preceding picture, but also with the values in macroblocks that neighbor the spatially corresponding macroblock in the preceding picture; see Figure 18-3.

Figure 18-3 Motion Compensation in H.261

The neighboring macroblocks being examined can be offset from the matching macroblock by a maximum of ±15 pixels in both the *x* and *y* directions. The macroblock in the history image that best matches the macroblock in the current image is used in calculating the difference values to encode.

If the macroblock in the history image used to calculate difference values is not the matching macroblock, the encoder must record the number of pixels by which the former is offset from the latter. These *x* and *y* offsets are written to a motion vector, which is later variable-length coded and written to the H.261 bitstream along with any encoded difference values.

The second operation that the encoder may perform before calculating the difference between macroblocks is a filter of the macroblock of interest in the history image. This filtering operation is designed to remove high-frequency information from the macroblock. Generally, this filtering leads to smaller differences between the macroblocks and, thus, to a more compact bitstream.

Encoding YC_bC_r or Difference Values

Whether the encoder is encoding the actual values in a macroblock from the current picture or difference values calculated during the prediction step, it uses the procedure depicted in Figure 18-4.

Figure 18-4   Encoding of YC_bC_r or Difference Values in H.261

Each of the six blocks in a macroblock (four blocks of luminance values and two blocks of color values) is encoded separately. The values for each block are:

- Transformed from the spatial to the transform domain using a Discrete Cosine Transform (DCT)

- Linearly quantized

- Encoded with variable length codes or, for less frequently occurring values, with 20-bit codes

This encoding scheme is very similar to the one used in the JPEG still-image compression standard. For a more detailed discussion of the steps involved in this type of encoding, see "How the JPEG Baseline Sequential Codec Works" on page 346.

Bit-Rate Control

As mentioned earlier, the H.261 codec is intended primarily for use in videophone and videoconferencing applications. Because these applications need to send data at a constant rate over a network, the encoder must use a constant number of bits to encode, say, a second's worth of video. The encoder can achieve this constant bit rate using any combination of the following techniques:

- Altering the criterion that determines whether a macroblock that is to be interframe encoded actually needs to be encoded.

  The macroblock needs to be encoded only if its luminance values differ from those in the corresponding macroblock in the preceding picture by a certain amount. By increasing this amount, the encoder decreases the number of macroblocks it must compress.

- Changing the values in the quantizer.

  To produce a lower bit rate, the encoder can increase the size of the values in the quantizer. This strategy results in quantized coefficients with relatively low values, which can be encoded with relatively short code words.

- Using the loop filter operation described in "Prediction" on page 384.

Provisions for Multipoint Conferencing

The CCITT's specification of the H.261 codec includes several features designed to facilitate multipoint conferencing. In a multipoint conference, the receiver may elect to switch between two or more sources of video. These features include:

- A freeze-picture request

  This request is an external signal that causes the decoder to stop updating the currently displayed picture. The picture remains frozen until the decoder sees a freeze-picture-release flag in the bitstream or until a timeout period of six seconds or more has elapsed.

- A fast-update request

  This request is an external signal that causes the encoder to compress the next picture using intraframe encoding exclusively. The encoder must compress this frame without overflowing its output buffer.

- A freeze-picture release

  When an encoder receives a free-picture release request, it sets a bit in the header of the next picture it encodes. This bit tells a decoder that has frozen its display to resume displaying pictures in the normal way.

Creating an H.261 CIS

Before you use the H.261 decompressor to decompress an H.261 bitstream, you must create an H.261 CIS (and write an H.261 bitstream to the CIS). You create this CIS by passing the decompressor name `H261` to the `xil_cis_create()` function. See the following code fragment.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "H261");
```

H.261 Codec Attributes

As discussed in "General CIS Attributes" on page 303, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for CISs attached to an H.261 compressor or decompressor. You set compressor-specific attributes using the `xil_cis_set_attribute()` function, and you read compressor-specific attributes using the `xil_cis_get_attribute()` function.

The H.261 attributes can be broadly grouped into those that affect compression, see "Compression Attributes," and those that affect decompression, see "Decompression Attributes" on page 395. Attributes are identified as read-only (R), write-only, or read/write (R/W).

Compression Attributes

Setting any of the following attributes affects how an H.261 compressor compresses images.

`COMPRESSOR_BITS_PER_IMAGE` *(R/W)*

This attribute controls the number of bits the encoder uses to encode pictures. Normally, you arrive at this value by taking the number of bits per second you can move over a network and dividing that number by the number of pictures you're encoding each second. For example, if you have 2 ISDN channels available for transporting data (2 * 65536 == 131,072 bits per second) and you want to encode 30 pictures a second, you would set `BITS_PER_IMAGE` to 4369. This setting would cause the compressor to encode a QCIF picture using about 0.17 bits per pixel.

The following code shows the bits-per-image attribute being set to 4369.

```
XilCis cis;
int bits_per_image = 4369;

xil_cis_set_attribute(cis, "COMPRESSOR_BITS_PER_IMAGE",
    (void *)bits_per_image);
```

COMPRESSOR_BITS_PER_IMAGE can be set to any int greater than or equal to 0. Its default value is 5069, the number of bits needed to encode a QCIF picture at 0.2 bits per pixel.

You can read the value of this attribute using code similar to the following.

```
XilCis cis;
int bits_per_image;

xil_cis_get_attribute(cis, "COMPRESSOR_BITS_PER_IMAGE",
    (void **)&bits_per_image);
```

COMPRESSOR_DOC_CAMERA *(R/W)*

The setting of this attribute determines whether the encoder sets the document-camera bit in the picture header of the pictures it encodes. If the attribute is set to FALSE, the encoder does not set this bit, and if it is set to TRUE, the encoder does set the bit. By default, the attribute is set to FALSE.

The following code fragment shows the attribute being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSOR_DOC_CAMERA",
    (void *)TRUE);
```

You can also read the value of this attribute using code similar to the following.

```
XilCis cis;
Xil_boolean doc_camera_status;

xil_cis_get_attribute(cis, "COMPRESSOR_DOC_CAMERA",
    (void **)&doc_camera_status);
```

COMPRESSOR_ENCODE_INTRA *(R/W)*

When the COMPRESSOR_ENCODE_INTRA attribute is set, a call to
xil_compress() causes the encoder to compress a whole picture in *intraframe*
mode. The encoder must compress this picture without causing its output
buffer to overflow.

In general, this attribute should be set to FALSE, its default value. An
application should set the attribute to TRUE only upon receiving a fast-update
request and should set the attribute back to FALSE after compressing one
frame.

The following code shows this attribute being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSOR_ENCODE_INTRA",
    (void *)TRUE);
```

You can read the value of this attribute using code similar to the following.

```
XilCis cis;
Xil_boolean encode_intra_status;

xil_cis_get_attribute(cis, "COMPRESSOR_ENCODE_INTRA",
    (void **)&encode_intra_status);
```

COMPRESSOR_FREEZE_RELEASE *(R/W)*

As mentioned in "Provisions for Multipoint Conferencing" on page 388, an
H.261 decoder can, in response to an external signal, freeze the picture
currently being displayed. This freeze remains in effect until a timeout period
has elapsed or until the decoder sees a freeze-release bit set in the bitstream.

Setting the encoder's COMPRESSOR_FREEZE_RELEASE attribute to TRUE causes
the encoder to set the freeze-release bit in the header of the subsequent pictures
it encodes. If a decoder has been in freeze-display mode, it exits that mode
when it reads the freeze-release bit. By default,
COMPRESSOR_FREEZE_RELEASE is set to FALSE.

The following code shows `FREEZE_RELEASE` being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSOR_FREEZE_RELEASE",
    (void *)TRUE);
```

You can also read the value of this attribute, as is done in the following code.

```
XilCis cis;
Xil_boolean freeze_release_status;

xil_cis_get_attribute(cis, "COMPRESSOR_FREEZE_RELEASE",
    (void **)&freeze_release_status);
```

`COMPRESSOR_IMAGE_SKIP` *(R/W)*

The `COMPRESSOR_IMAGE_SKIP` attribute does not control the number of pictures the compressor skips between compressed pictures: the application controls that. However, if the application is skipping pictures, the attribute should be set to the number of pictures being skipped. This value figures in a calculation the compressor performs before filling in the 5-bit temporal-reference field in the header of each compressed picture. The compressor sets the value of this field to the value of the temporal-reference field in the last compressed picture plus the value of `COMPRESSOR_IMAGE_SKIP` plus 1.

The attribute can have a value in the range 0 to 31, and its default value is 0. The following code shows `IMAGE_SKIP` being set to 1. This is the appropriate setting if your application is compressing every other frame of video input.

```
XilCis cis;
int images_skipped = 1;

xil_cis_set_attribute(cis, "COMPRESSOR_IMAGE_SKIP",
    (void *)images_skipped);
```

You can also read the value of this attribute using code similar to the following.

```
XilCis cis;
int images_skipped;

xil_cis_get_attribute(cis, "COMPRESSOR_IMAGE_SKIP",
    (void **)&images_skipped);
```

COMPRESSOR_LOOP_FILTER *(R/W)*

When an H.261 encoder is interframe encoding a macroblock and is using motion compensation, the encoder has the option of filtering the macroblock in the history image that is being used in the operation. (For more information about interframe encoding in H.261, see "Prediction" on page 384.) The setting of the COMPRESSOR_LOOP_FILTER attribute provides a hint to the encoder concerning whether it should filter the macroblock in the history image or not: a setting of TRUE suggests that the encoder perform the filtering if necessary, and a setting of FALSE suggests that it not use the filter.

The default value for COMPRESSOR_LOOP_FILTER is TRUE. This setting gives the encoder the flexibility to produce the most compact bitstream. Use the FALSE setting when you need to minimize compression and decompression time.

The following code shows COMPRESSOR_LOOP_FILTER being set to FALSE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSOR_LOOP_FILTER",
    (void *)FALSE);
```

You can also read the value of this attribute. Use code similar to the following.

```
XilCis cis;
Xil_boolean loop_filter_status;

xil_cis_get_attribute(cis, "COMPRESSOR_LOOP_FILTER",
    (void **)&loop_filter_status);
```

COMPRESSOR_MV_SEARCH_RANGE *(R/W)*

As discussed in "Prediction" on page 384, when an H.261 encoder is interframe encoding a macroblock, it may use a technique called motion compensation in determining which macroblock in the history image to use for the operation. This macroblock in the history image may be offset by up to ±15 pixels in both the *x* and *y* directions from the macroblock in the history image that matches exactly the position of the macroblock that is being encoded. This means that the encoder may have to look at the contents of 961 (31 * 31) macroblocks to find the best match for the macroblock it is encoding. The COMPRESSOR_MV_SEARCH_RANGE attribute enables you to limit the extent of this search and, consequently, to speed up the encoding process.

Before setting the value of this attribute, you must declare and assign values to the members of an XilH261MVSearchRange structure.

```
typedef struct {
    int x;
    int y;
} XilH261MVSearchRange;
```

The value of x determines the extent of the search horizontally; that is, if x is set to 5, the encoder can only search five pixels to the right or left. The value of y determines the extent of the search vertically. Setting both values to 0 means that the encoder should not perform motion compensation at all.

Note – The value of this attribute is actually only a hint to the encoder and could be ignored.

The following code shows COMPRESSOR_MV_SEARCH_RANGE being set so that the encoder searches only 7 pixels horizontally and vertically.

```
XilCis cis;
XilH261MVSearchRange search_range;

search_range.x = 7;
search_range.y = 7;
xil_cis_set_attribute(cis, "COMPRESSOR_MV_SEARCH_RANGE",
    (void *)&search_range);
```

You can also read the value of this attribute. See the following code fragment.

```
XilCis cis;
XilH261MVSearchRange search_range;

xil_cis_get_attribute(cis, "COMPRESSOR_MV_SEARCH_RANGE",
    (void **)&search_range);
```

COMPRESSOR_SPLIT_SCREEN *(R/W)*

The setting of the COMPRESSOR_SPLIT_SCREEN attribute determines whether
the encoder sets the split-screen bit in the picture header of the pictures it
encodes. If the attribute is set to FALSE, the encoder does not set this bit, and if
it is set to TRUE, the encoder sets the bit. By default, the attribute is set to
FALSE.

The following code fragment shows the attribute being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COMPRESSOR_SPLIT_SCREEN",
    (void *)TRUE);
```

You can also read the value of this attribute using code similar to the following.

```
XilCis cis;
Xil_boolean split_screen_status;

xil_cis_get_attribute(cis, "COMPRESSOR_SPLIT_SCREEN",
    (void **)&split_screen_status);
```

Decompression Attributes

Of the decompression attributes, only IGNORE_HISTORY affects the behavior
of the decoder. The remaining attributes simply enable you to read values from
the header of the most recently decompressed picture.

`IGNORE_HISTORY` *(R/W)*

The `IGNORE_HISTORY` attribute affects your ability to seek forward and backward in an H.261 bitstream. These seeks are somewhat problematic in H.261 because the codec relies so heavily on interframe encoding and does not require periodic key frames.

By default, `IGNORE_HISTORY` is set to `FALSE`. In this case, seeking backward is illegal because there is no way to seek backward and still have a valid history frame to use in decoding a picture. Seeking forward is possible, but to ensure that the decoder can properly decode the frame you seek to, the decoder must actually decode all the frames you "skip." This is the only way the decoder can maintain the correct data in its history buffer.

If you set `IGNORE_HISTORY` to `TRUE`, you're telling the decoder that it's OK if, after a seek, it does not have the correct data in its history buffer to decode the next picture. In this case, seeking backward is legal, and seeking forward is faster than they would be otherwise. The drawback to ignoring the history image is that you may have to decode as many as 132 frames after a seek before you get a properly reconstructed picture.

Note – There is one exception to this problem. If `IGNORE_HISTORY` is set to `TRUE` and you seek backward by one frame and then call `xil_decompress()`, you get a good picture. The decompressor can copy this picture from its history buffer.

The following code shows `IGNORE_HISTORY` being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "IGNORE_HISTORY", (void *)TRUE);
```

You can also read the value of this attribute. To do this, use code similar to the following.

```
XilCis cis;
Xil_boolean ignore_history_status;

xil_cis_get_attribute(cis, "IGNORE_HISTORY",
    (void **)&ignore_history_status);
```

DECOMPRESSOR_DOC_CAMERA *(R)*

Reading the DECOMPRESSOR_DOC_CAMERA attribute tells you whether the document-camera bit was set in the header of the most recently decompressed picture. If the value of the attribute is TRUE, the bit was set, and if the value is FALSE, it was not set.

The following code fragment illustrates how to read the attribute.

```
XilCis cis;
Xil_boolean doc_camera_status;

xil_cis_get_attribute(cis, "DECOMPRESSOR_DOC_CAMERA",
    (void **)&doc_camera_status);
```

DECOMPRESSOR_FREEZE_RELEASE *(R)*

The DECOMPRESSOR_FREEZE_RELEASE attribute enables you to read the value of the freeze-release bit for the most recently decompressed picture. If the bit is set, the value of the attribute is TRUE; otherwise, it is FALSE.

Here's how an application uses this information: If the application has frozen the display in response to a freeze-picture request and DECOMPRESSOR_FREEZE_RELEASE is set to FALSE, the application does not display the picture it just decoded (unless a timeout period has elapsed). On the other hand, if the attribute is set to TRUE, it displays that picture, and all ensuing pictures until it receives another freeze-picture request.

The following code shows how to read the value of
DECOMPRESSOR_FREEZE_RELEASE.

```
XilCis cis;
Xil_boolean stop_freeze;

xil_cis_get_attribute(cis, "DECOMPRESSOR_FREEZE_RELEASE",
    (void **)&stop_freeze);
```

DECOMPRESSOR_SOURCE_FORMAT *(R)*

The DECOMPRESSOR_SOURCE_FORMAT attribute enables you to determine
whether the most recently decompressed picture was in CIF (352 by 288 pixels)
or QCIF (176 by 144 pixels) format. The value of the attribute is one of the
following enumeration constants.

```
typedef enum {
    QCIF, CIF
} XilH261SourceFormat;
```

The following code shows how you might read the value of
DECOMPRESSOR_SOURCE_FORMAT.

```
XilCis cis;
XilH261SourceFormat source_format;

xil_cis_get_attribute(cis, "DECOMPRESSOR_SOURCE_FORMAT",
    (void **)&source_format);
```

DECOMPRESSOR_SPLIT_SCREEN *(R)*

Reading the DECOMPRESSOR_SPLIT_SCREEN attribute tells you whether the
split-screen bit is set in the header of the most recently decompressed picture.
If the value of the attribute is TRUE, the bit is set, and if the value is FALSE, it
is not set.

The following code fragment illustrates how to read the attribute.

```
XilCis cis;
Xil_boolean split_screen_status;

xil_cis_get_attribute(cis, "DECOMPRESSOR_SPLIT_SCREEN",
    (void **)&split_screen_status);
```

DECOMPRESSOR_TEMPORAL_REFERENCE *(R)*

The DECOMPRESSOR_TEMPORAL_REFERENCE attribute enables you to retrieve the value of the temporal-reference field in the header of the last picture decoded. Since this is a 5-bit field, the value is in the range 0 to 31.

By looking at the temporal reference fields of the last two decompressed frames, an application can determine how many pictures the encoder skipped between these pictures. In most instances, the number of skipped pictures equals:

temporal reference for $picture_x$ - temporal reference for $picture_{x-1}$ - 1

However, if the temporal reference value for $picture_x$ is less than the temporal reference for $picture_{x-1}$, the calculation becomes:

31 + temporal reference for $picture_x$ - temporal reference for $picture_{x-1}$

The following code shows how to read the value of DECOMPRESSOR_TEMPORAL_REFERENCE.

```
XilCis cis;
int temp_ref;

xil_cis_get_attribute(cis, "DECOMPRESSOR_TEMPORAL_REFERENCE",
    (void **)&temp_ref);
```

≡ *18*

H.261 Molecules

The XIL library includes a series of molecules that accelerate the playback of H.261 bitstreams. These molecules are optimized routines that perform the jobs of two or more functions from the XIL API. You do not call such an optimized routine directly; rather, the library calls a molecule when your program calls a predefined sequence of XIL functions, sometimes with specific arguments.

For example, if your program calls `xil_decompress()` to decode an image stored in an H.261 CIS and then calls `xil_ordered_dither()` to dither the decoded image from a 24- to an 8-bit image, the library may not call these two functions. Instead, it may call a molecule that performs the decompression and the dithering in an optimized way.

For more information about accelerating the playback of H.261 bitstreams, see Chapter 23, "Acceleration in XIL Programs." For information about the H.261 molecules that are available and information about how to call those molecules, see "XIL Molecules" on page 477.

MPEG-1 Codec 19≡

This chapter discusses the MPEG-1 video codec specified by the Moving Picture Experts Group, an ISO working group. This group has produced a standard that is similar to the H.261 standard developed by the CCITT (see Chapter 18, "H.261 Codec"), but places less emphasis on low bit rates. By accepting a higher bit rate—for example, 1.5 Mbits per second—an MPEG-1 codec is able to recreate very high-quality pictures and to produce a bitstream that is easily editable.

Note – This chapter discusses both MPEG-1 compression and decompression. However, the current release of the XIL library includes only an MPEG-1 decompressor. The compressor interface is defined for third parties who want to implement XIL MPEG-1 compressors.

The rate of 1.5 Mbits/s makes the MPEG-1 codec especially viable in applications that read compressed data from CD-ROMs because even older (1x) CD-ROM readers can read data at this speed. Thus, putting an MPEG-1 bitstream on a CD-ROM is an effective way to distribute movies, business presentations, and training videos.

This chapter has the following main sections:

How an MPEG-1 Codec Works	*page 402*
Creating an MPEG-1 CIS	*page 407*
MPEG-1 Codec Attributes	*page 408*
MPEG-1 Molecules	*page 433*

≡ *19*

How an MPEG-1 Codec Works

Since the work done by the Motion Picture Experts Group grew out of the work done by the CCITT in developing the H.261 video codec, this section compares and contrasts the MPEG-1 codec with the H.261 codec to show how the MPEG-1 codec works. (If you're unfamiliar with how an H.261 codec works, see "How an H.261 Codec Works" on page 380.) The present section also includes a subsection that describes the organizational structure that MPEG-1 imposes on a video sequence.

Similarities Between MPEG-1 and H.261

These are the key similarities between MPEG-1 and H.261:

- Both compressors work with YC_bC_r pictures in which the information in the chroma channels has been subsampled so that there is one C_b and one C_r value for each 2-by-2 block of luma values.

- Like the H.261 compressor, the MPEG-1 compressor can compress a macroblock (a 16-by-16 block of pixels in a picture) by encoding the actual Y, C_b, and C_r values in the macroblock (intraframe encoding) or by encoding the differences between values in the current block and values in the corresponding macroblock in the previous picture (forward prediction). In addition, when using the forward-prediction encoding method, the MPEG-1 compressor, like the H.261 compressor, can employ motion compensation.

- Both compressors encode 8-by-8 blocks of pixel values or difference values using the same method. The compressors first perform a DCT on the 8-by-8 block of values. This operation transforms the values in the block from the spatial to the transform domain. Second, the compressors quantize the coefficients produced by the DCT. Finally, the compressors use entropy coding to encode the quantized coefficients.

Differences Between MPEG-1 and H.261

Although the Moving Picture Experts Group drew heavily on the work of the CCITT, there are also significant differences between the MPEG-1 and H.261 compressors. Most of these differences provide for random access to the MPEG-1 bitstream and make the bitstream easily editable.

I Pictures and P Pictures

In H.261, the ideas of intraframe encoding and predictive encoding are applied for the most part at the macroblock level. MPEG-1, on the other hand, includes the notion of intraframe-encoded pictures and predicted pictures.

In an intraframe-encoded picture, or *I picture*, all macroblocks are intraframe encoded. This means that the decoder needs no information from a preceding picture to decode an I picture. For this reason, seeking to I frames can be performed very quickly.

In a predicted picture, or *P picture*, each macroblock can be intraframe encoded or encoded using the forward-prediction method. This type of picture is very similar to an H.261 picture.

Typically, an MPEG-1 bitstream contains more P pictures than I pictures because the P picture can be encoded using fewer bits. Encoding difference values generally requires fewer bits than encoding pixel values. Also, the encoder does not have to encode macroblocks that are very similar to their counterparts in the preceding I or P picture.

B Pictures

Based on the previous discussion, you might picture an MPEG-1 bitstream as containing periodic I pictures followed by a number of P pictures. See Figure 19-1.

| I | P | P | P | I | P | P | P | I | P | P | P | I |

Figure 19-1 An MPEG-1 Bitstream Containing I and P Pictures

This does constitute a legal MPEG-1 bitstream. However, the bitstream may also contain one or more bidirectionally predicted pictures, or B pictures, between any pair of I or P pictures. In a B picture, each macroblock may be:

- Intraframe encoded

- Forward predicted from the nearest preceding I or P picture

- Backward predicted from the nearest succeeding I or P picture

- Bidirectionally predicted from the nearest preceding I or P picture and the nearest succeeding I or P picture

Intraframe encoding and forward prediction are discussed in "I Pictures and P Pictures" on page 403.

Backward prediction is strictly analogous to forward prediction. A backward-predicted macroblock is encoded with respect to the values in the corresponding macroblock in a picture that follows its own picture in the video sequence. This option may seem unintuitive at first because it results in a situation where pictures are not transmitted in the order in which they will be displayed; see Figure 19-2.

Images that are to
be displayed in
this order...

must be transmitted
in this order.

| I | B | P | | I | P | B |

Figure 19-2 MPEG-1 Display Order Versus Decoding Order

However, backward prediction is an option because it can lead to a more compact bitstream in instances where the macroblocks in a B picture match their counterparts in the closest succeeding I or P picture more closely than they match their counterparts in the closest preceding I or P picture.

A bidirectionally predicted macroblock is coded with respect to the corresponding macroblock in both the closest preceding and succeeding I/P pictures; see Figure 19-3.

Figure 19-3 Bidirectional Prediction in MPEG-1

In this case, the encoder averages the values in macroblocks A and C and then encodes the difference between values in macroblock B and the averaged macroblock. Bidirectional prediction is effective because the difference mentioned above is often so small that it does not need to be encoded.

For MPEG-1, the behavior of the `xil_cis_get_bits_ptr()` function (defined on page 300) differs from its usual behavior. For a bitstream with out-of-order frames (that is, a bitstream with B frames), the actual number of frames in the data returned by `xil_cis_get_bits_ptr()` might not equal the value of its `nframes` parameter. The value of `nframes` is the number of frames the CIS read frame has been advanced by the `xil_cis_get_bits_ptr()` call. A backward seek of `nframes` called by `xil_cis_seek()` (defined on page 302) restores the read position to the original read frame (before the `xil_cis_get_bits_ptr()` call). This is useful for an application that uses a preview mode, where the compress and write-to-file is followed by a decompress.

The number of frames reported may not represent the actual number of frames. Therefore, if the `chunk` is subsequently used for an `xil_cis_put_bits()` (defined on page 298) or `xil_cis_put_bits_ptr()` call (defined on page 298), the `nframes` parameter must be set to -1, which indicates an unknown number of frames.

≡ *19*

Groups of Pictures

Another concept used in MPEG-1 that does not apply in H.261 is that of a group of pictures. This is a series of consecutive pictures from a video sequence. Generally, the group of pictures provides a unit of the bitstream that can be removed without destroying the integrity of the bitstream and a unit that can be decoded independently of the rest of the bitstream. As you'll see in a moment, however, the group of pictures does not always have these characteristics.

By definition, a group of pictures (considered in display order) must begin with an I picture or with one or more B pictures followed by an I picture. It must end with an I or a P picture. A closed group of pictures can be decoded without any information from the preceding group of pictures. Thus, a closed group is one that begins with an I picture or one that begins with one or more B pictures whose macroblocks have been encoded using only intraframe encoding and backward prediction. An open group of pictures begins with one or more B pictures, at least one of which contains macroblocks encoded using forward or bidirectional prediction. This type of group can be decoded only if the preceding group of pictures is available. If that group is not available—for example, if an MPEG-1 bitstream editing program has removed the group—a broken-link bit must be set in the header for the open group of pictures.

How MPEG-1 Organizes a Video Sequence

Some of the organizational units used in MPEG-1, such as the group of pictures and the macroblock are mentioned in "Groups of Pictures." This section provides a complete overview of these units. This information should be useful to you as you read about MPEG-1 CIS attributes later in this chapter because many of these attributes are associated with a particular unit.

The largest unit MPEG-1 defines is the video sequence. You might think of this unit as an entire movie or presentation. Each MPEG-1 bitstream includes a sequence header, which defines several attributes for the entire sequence. These include the pixel aspect ratio for the sequence, the picture rate for the sequence in pictures per second, and the bit rate of the data channel over which the compressed sequence will be moved. An additional sequence header is written to the bitstream each time one of the encoder's quantization tables is changed.

Each sequence is divided into a series of groups of pictures. See "Groups of Pictures" on page 406 for a definition of this unit. If you're compressing video using an XIL-compliant MPEG-1 encoder, you can control the makeup of each group of pictures using the COMPRESSOR_PATTERN attribute.

Each group of pictures consists of pictures, which are individual frames of video. The major attributes associated with each picture are a picture type and a temporal reference. The major picture types are intraframe-encoded pictures, forward-predicted pictures, and bidirectionally predicted pictures. The temporal reference is an integer identifying a picture's place within a group of pictures.

The largest subdivision of a picture is called a slice, which consists of a series of consecutive macroblocks. Slices within a picture may vary in size, but each macroblock in a picture must be part of a slice. This unit is designed primarily to help a decoder recover from a bitstream error. If a decoder detects an error, one way to recover is to skip to the next slice header.

Slices are built from macroblocks that are a 16-by-16 block of pixels. The macroblock is the level at which motion compensation is performed. Also, the type of encoding used—intraframe, forward prediction, backward prediction, or bidirectional prediction—can change from macroblock to macroblock.

Finally, each macroblock is divided into six 8-by-8 blocks. Four of these blocks contain luma values, one contains C_b values, and one contains C_r values. This is the level at which the DCT is performed.

Creating an MPEG-1 CIS

Before you can use the MPEG-1 decompressor to decompress an MPEG-1 bitstream, you must create an MPEG-1 CIS (and write an MPEG-1 bitstream to the CIS). You create this CIS by passing the decompressor name Mpeg1 to the xil_cis_create() function; see the following code fragment.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "Mpeg1");
```

$\equiv$ *19*

MPEG-1 Codec Attributes

As discussed in "General CIS Attributes" on page 303, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for CISs attached to an MPEG-1 compressor or decompressor. You set codec-specific attributes using the `xil_cis_set_attribute()` function, and you read codec-specific attributes using the `xil_cis_get_attribute()` function.

The MPEG-1 attributes can be broadly grouped into those that affect compression, discussed in "Compression Attributes," and those that affect decompression, discussed in "Decompression Attributes" on page 427.

Compression Attributes

Setting any of the following attributes affects how an MPEG-1 compressor compresses images. It is also possible to read the value of each of these attributes. Attributes are identified as read-only (R), write-only (W), or read/write (R/W).

`COMPRESSOR_BITS_PER_SECOND` *(R/W)*

You use the `COMPRESSOR_BITS_PER_SECOND` attribute to tell the encoder how many bits it can use to encode one second's worth of pictures. This value is written to the sequence header for a picture sequence and is assumed to remain constant for the duration of the sequence. You cannot change the value of this attribute after the encoder compresses the first picture in the sequence.

The value you assign to the attribute must fall in the range 1 to 104,856,800. Before inserting this value in the sequence header, the encoder rounds it up to the nearest multiple of 400. The default value of the attribute is 1,152,000.

The following code shows how you might set the value of `COMPRESSOR_BITS_PER_SECOND` to 1,856,000.

```
XilCis cis;
int bits_per_second = 1856000;

xil_cis_set_attribute(cis, "COMPRESSOR_BITS_PER_SECOND",
    (void *)bits_per_second);
```

To read the value of the attribute, you might use code similar to this.

```
XilCis cis;
int bits_per_second;

xil_cis_get_attribute(cis, "COMPRESSOR_BITS_PER_SECOND",
    (void **)&bits_per_second);
```

COMPRESSOR_INSERT_VIDEO_SEQUENCE_END *(R/W)*

Normally, the value of the COMPRESSOR_INSERT_VIDEO_SEQUENCE_END attribute is FALSE. If you set it to TRUE, any time a subsequent call is made to xil_cis_flush() (defined on page 325), the encoder writes a video-sequence-end (eos) code to the last frame in the CIS. This action is added to the normal actions taken by the flush routine. When set to FALSE, this attribute doesn't affect the normal actions of the flush routine.

Setting COMPRESSOR_INSERT_VIDEO_SEQUENCE_END *to* TRUE

The following code shows the attribute being set to TRUE.

```
XilCis cis;

xil_cis_set_attribute(cis,
    "COMPRESSOR_INSERT_VIDEO_SEQUENCE_END", (void *)TRUE);
```

Note – The library prevents multiple eos codes from being written to the same frame. Thus, when the write-frame's number doesn't change and COMPRESSOR_INSERT_VIDEO_SEQUENCE_END is TRUE, multiple calls to xil_cis_flush() result in only one eos code in the frame; this code is written after the first call to xil_cis_flush().

You can read the value of this attribute using code similar to this.

```
XilCis cis;
Xil_boolean sequence_end_status;

xil_cis_get_attribute(cis,
    "COMPRESSOR_INSERT_VIDEO_SEQUENCE_END",
    (void **)&sequence_end_status);
```

Table 19-1 shows two sequences of function calls that result in the same CIS pattern; these sequences assume the call to `xil_cis_set_attribute()` sets the `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END` attribute to `TRUE`.

Note – The term *CIS pattern* refers to the order that frames appear in the bitstream. Don't confuse this with the `COMPRESSOR_PATTERN` attribute (see page 419), which lets you specify the makeup of a group of pictures. For example, the `COMPRESSOR_PATTERN IBBPBBP` results in the bitstream CIS pattern `IPBBPBB`.

Table 19-1 Two Function Call Sequences: CIS Pattern = IPBB

Function Call Sequence 1	Function Call Sequence 2
`xil_compress()`	`xil_cis_set_attribute()`
`xil_compress()`	`xil_compress()`
`xil_compress()`	`xil_compress()`
`xil_compress()`	`xil_compress()`
`xil_cis_flush() /* normal */`	`xil_compress()`
`xil_cis_set_attribute()`	`xil_cis_flush() /* normal + eos */`
`xil_cis_flush() /* normal + eos */`	

For the two sequences shown in Table 19-1, the resulting CIS sequence is:

`vsh.I0.P3.B1.B2.eos`

> where `vsh` represents the video sequence header information, and `eos` represents the end-of-sequence code. Between the `vsh` and the `eos` code are the frame types and their display id's; among the frame types, `I` = interframe encoding, `P` = predicted pictures, and `B` = bidirectionally predicted pictures.

From the XIL point of view, the `vsh` is bundled with the first frame, `I0`, and the `eos` is bundled with the last frame, `B2`.

Handling Multiple Video Sequence Headers and Sequences

There may be multiple vsh components associated with one eos, since the vsh changes as certain CIS attributes change (within the XIL limitations that there are no width/height changes).

In addition, there may be multiple sequences within a bitstream; thus,

```
vsh-----eos.vsh----eos.vsh----eos
```

constitutes three sequences.

If frames are compressed into the CIS after the call to `xil_cis_flush()`, it's the compressor's responsibility to provide the vsh per sequence. Before the application changes an attribute that would result in a new sequence header, it must first output an eos for the current sequence by calling `xil_cis_flush()` with `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END = TRUE`. For example, before you can change the `COMPRESSOR_BITS_PER_SECOND` attribute (see page 408), you must first set `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END` to `TRUE` and call `xil_cis_flush()` to write the eos code to the bitstream.

An MPEG-1 sequence isn't valid without the eos code; therefore, the last frame in the sequence must be followed by the eos code. Since it cannot be predicted when an application will end a sequence, the MPEG-1 codec *reserves* the last frame or subgroup of frames in the CIS so that you can write an eos to that frame or subgroup. The reserved frame or subgroup must be released before it can be retrieved with `xil_cis_get_bits_ptr()` or `xil_decompress()`.

Releasing Reserved Frames

Note – The reserved frame or subgroup affects the logic you need for calling `xil_cis_get_bits_ptr()` and `xil_decompress()`. For example, typically you call `xil_decompress()` from within a loop that executes only when `xil_cis_has_frame()` returns `TRUE`. When decompressing an MPEG-1 CIS, you must first release the reserved frames before you can expect a loop control condition that depends on the return value of `xil_cis_has_frame()` to evaluate to `TRUE`.

The following paragraphs tell you how to release the reserved frames. Beginning on page 414, the discussion moves to some example function-call sequences and shows how these sequences would affect both the return value of `xil_cis_has_frame()` and `xil_cis_get_bits_ptr()`.

A frame or subgroup is released when:

- It is followed by an eos, thus providing a valid sequence. This is done by setting the `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END` attribute to `TRUE` and then calling `xil_cis_flush()`, as described above.

- It is followed by another frame or subgroup. For an all I bitstream or a mixed I-P bitstream, this means that another frame is added to the CIS. Since there are no out-of-order frames, a subgroup for this type of CIS is a frame.

The following sequences release a frame:

```
I0.I1          I1 releases I0
I0.P1          P1 releases I0
```

For any bitstream with B frames, releasing a frame or subgroup is more complicated. B frames have a future-predictive frame that appears in the bitstream *previous* to the B frame, and multiple B frames may share the same predictive frame. So you can get the following (in these examples, any of the I's can be replaced by a P):

```
I2.B1

I3.B1.B2

I4.B1.B2.B3

I6.B1.B2.B3.B4.B5

I9.B1.B2.B3.B4.B5.B6.B7.B8
```

Sequence Subgroups

A sequence is considered a complete *subgroup* when its predictive frame has the display id N, and the predictive frame is followed by one or more B frames, the last one's display id is N-1. Thus, all of the above bitstream fragments are complete subgroups.

The following are examples of bitstreams with incomplete subgroups:

```
I3.B1          missing B2
I6.B1.B2.B3    missing B4,B5
```

When there are incomplete subgroups, the compressor must buffer the compressed frames until the buffer contains a complete subgroup, at which point the compressor adds the completed subgroup to the bitstream. In this manner, the compressor can handle interruptions to the compression-frame

sequence as necessary. For example, if, halfway through the IPBBB group, the video player's Stop/Output button is pressed, the compressor must be able to take the resulting I0.P4.B1 bitstream and rework it into a legal output form.

Once the subgroup is complete, it cannot be released until it is followed by an eos, or it is followed by another valid subgroup. The following sequences show a subgroup being released:

```
I0.I2.B1.eos            I2.B1 released by eos
I0.I2.B1.I4.B3          I2.B1 released by I4.B3
I0.I3.B1.B2.P5.B4       I3.B1.B2 released by P5.B4
I0.I3.B1.B2.P5.B4.eos   P5.B4 released by eos
```

Table 19-2, Table 19-3, and Table 19-4 show how subgroups are affected by a particular sequence of function calls. In each of these tables:

- Column heading **WF** is an abbreviation for Write Frame.

- Column heading `has_frame` is an abbreviation for the `xil_cis_has_frame()` function and the column text shows what the return value for that function would be.

- Column heading `get_bits_ptr` is an abbreviation for the `xil_cis_get_bits_ptr()` function and the column text shows the frames that would be returned by a call to that function.

- Column heading **RF** is an abbreviation for Read Frame and the column text shows what the read frame would be after the call to `xil_cis_get_bits_ptr()`.

- A call to `xil_cis_set_attribute()` in the sequence sets the `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END` attribute to `TRUE`.

In Table 19-2, the CIS pattern is all I frames and the result is that subgroup I2 is released by an appended eos. Notice that the first `xil_cis_flush()` call has no affect on the subgroup since the default setting for the `COMPRESSOR_INSERT_VIDEO_SEQUENCE_END` attribute is `FALSE`.

Table 19-2 Releasing a Frame: CIS Pattern = All I Frames

Function	WF	has_frame	get_bits_ptr	RF	CIS
xil_compress()	1	FALSE	NULL	0	I0
xil_compress()	2	TRUE	I0	1	I0.I1
xil_compress()	3	TRUE	I1	2	I0.I1.I2
xil_cis_flush()	3	FALSE	NULL	2	I0.I1.I2
xil_cis_set_attribute()					
xil_cis_flush()	3	TRUE	I2.eos	3	I0.I1.I2.eos

In Table 19-3, the CIS pattern is IPB and the result is that subgroup P2.B1 is released by the appended `eos`.

Table 19-3 Releasing a Frame: CIS Pattern = IPB

Function	WF	has_frame	get_bits_ptr	RF	CIS
xil_compress()	1	FALSE	NULL	0	I0
xil_compress()	2	FALSE	NULL	0	I0
xil_compress()	3	TRUE	I0	1	I0.P2.B1
xil_cis_set_attribute()					
xil_cis_flush()	3	TRUE	P2.B1.eos	3	I0.P2.B1.eos

Finally, in Table 19-4, the CIS pattern is IPBBPBB and the result is that subgroup P3.B1.B2 is released by the next subgroup, P6.B4.B5.

Table 19-4 Releasing a Frame: CIS Pattern = IPBBPBB

Function	WF	has_frame	get_bits_ptr	RF	CIS
xil_compress()	1	FALSE	NULL	0	I0
xil_compress()	2	FALSE	NULL	0	I0
xil_compress()	3	FALSE	NULL	0	I0

Table 19-4 Releasing a Frame: CIS Pattern = IPBBPBB

Function	WF	has_frame	get_bits_ptr	RF	CIS
xil_compress()	4	TRUE	I0	1	I0.P3.B1.B2
xil_compress()	5	FALSE	NULL	1	I0.P3.B1.B2
xil_compress()	6	FALSE	NULL	1	I0.P3.B1.B2
xil_compress()	7	TRUE	P3.B1.B2	4	I0.P3.B1.B2.P6.B4.B5

COMPRESSOR_INTRA_QUANTIZATION_TABLE *(R)*

The COMPRESSOR_INTRA_QUANTIZATION_TABLE attribute enables you to specify an 8-by-8 matrix of values to be used in quantizing the DCT coefficients for blocks in intraframe-encoded macroblocks. If you set this attribute, the table is written to your bitstream's sequence header. If you don't set it, the encoder uses a default implementation-specific table.

If you supply your own table, it must meet these requirements:

- The first value in the table must be an 8, the fixed quantization level of the DC coefficient for the block.

- The remaining values must fall in the range 1 to 255.

- The values must be listed in the zigzag order shown in Figure 19-4.

Figure 19-4 Zigzag Ordering of Quantization Table Values

The following code shows how to specify a table. This particular table is the one that is used as an example in the MPEG-1 standard and may well be the encoder's default table.

```
XilCis cis;
Xil_unsigned8 intra_quant_table[64] = {
    8, 16, 19, 22, 26, 27, 29, 34,
    16, 16, 22, 24, 27, 29, 34, 37,
    19, 22, 26, 27, 29, 34, 34, 38,
    22, 22, 26, 27, 29, 34, 37, 40,
    22, 26, 27, 29, 32, 35, 40, 48,
    26, 27, 29, 32, 35, 40, 48, 58,
    26, 27, 29, 34, 38, 46, 56, 69,
    27, 29, 35, 38, 46, 56, 69, 83 };

xil_cis_set_attribute(cis,
    "COMPRESSOR_INTRA_QUANTIZATION_TABLE",
    (void *)intra_quant_table);
```

You can read the value of the attribute using code similar to this.

```
XilCis cis;
Xil_unsigned8 *intra_quant_table;

xil_cis_get_attribute(cis,
    "COMPRESSOR_INTRA_QUANTIZATION_TABLE",
    (void **)&intra_quant_table);
```

If, after this call, `intra_quant_table` is a null pointer, the default table is in use; otherwise, `intra_quant_table` is a pointer to a user-supplied quantization matrix. In the latter case, your application must free the matrix pointed to by `intra_quant_table` when it is no longer needed.

COMPRESSOR_NON_INTRA_QUANTIZATION_TABLE *(R/W)*

The `COMPRESSOR_NON_INTRA_QUANTIZATION_TABLE` attribute enables you to specify an 8-by-8 matrix of values to be used in quantizing the DCT coefficients for blocks in nonintraframe-encoded macroblocks. If you set this attribute, the table is written to your bitstream's sequence header. If you don't set it, the encoder uses an implementation-specific default table.

If you supply an alternate table, it must meet these requirements:

- The values in the table must fall in the range 1 to 255.

- The values must be listed in the zigzag order shown in Figure 19-4 on page 417.

The following code is an example of how you might supply a custom table. This table, which contains all 16's, is used as an example in the MPEG-1 specification.

```
XilCis cis;
Xil_unsigned8 non_intra_quant_table[64];
int i;


for (i = 0; i < 64; i++)
    non_intra_quant_table[i] = 16;

xil_cis_set_attribute(cis,
    "COMPRESSOR_NON_INTRA_QUANTIZATION_TABLE",
    (void *)non_intra_quant_table);
```

You can read the value of the attribute using code similar to the following.

```
XilCis cis;
Xil_unsigned8 *non_intra_quant_table;

xil_cis_get_attribute(cis,
    "COMPRESSOR_NON_INTRA_QUANTIZATION_TABLE",
    (void **)&non_intra_quant_table);
```

COMPRESSOR_PATTERN *(R)*

The COMPRESSOR_PATTERN attribute enables you to specify the makeup of each group of pictures. You dictate the number of pictures in each group, the types of the pictures in the group, and the display order of those pictures. Your specification must conform to the following rules:

- A group of pictures may begin with one or more B pictures or an I picture.

- If the group starts with one or more B pictures, those pictures must be followed by an I picture.

- The first I picture in the group may be followed by any number of I and P pictures, and any combination of an I and a P picture may have intervening B pictures.

- The group of pictures must end with an I or a P picture.

Note – A group of pictures can also contain all D pictures. (A D picture contains only low-frequency information and is used for fast visible searches.) However, the XIL library's MPEG-1 decompressor cannot decode bitstreams containing D pictures.

You specify the makeup of the group by assigning values to a structure of type `XilMpeg1Pattern`, as shown here.

```
typedef struct {
    char *pattern;
    Xil_unsigned32 repeat_count;
} XilMpeg1Pattern;
```

The string `pattern` determines the types of the pictures (in display order) that appear at the beginning of the group of pictures. This string must contain a combination of the characters shown in Table 19-5.

Table 19-5 Characters Representing Picture Types

Character	Picture Type
I	Intraframe picture
P	Forward-predicted picture
B	Bidirectionally predicted picture
D	DC-coefficient picture

For example, suppose you want each group of pictures to begin with the thirteen pictures shown in Figure 19-5.

Figure 19-5 Sample Group of Pictures

You would set `pattern` to "`IBBPBBPBBPBBP`".

The `repeat_count` parameter is a number greater than 0 that determines the total number of pictures in the group of pictures. For example, suppose that you set `repeat_count` to 3 as shown in this example.

```
XilCis cis;
XilMpeg1Pattern picture_pattern = {"IBBPBBPBBPBBP", 3};

xil_cis_set_attribute(cis, "COMPRESSOR_PATTERN",
    (void *)&picture_pattern);
```

This call requests that the pattern shown in Figure 19-5 be repeated 3 times, so there will be a total of 39 pictures in the group.

When you set this attribute, the encoder terminates the group of pictures it was working on immediately. Thus, the next call to `xil_compress()` causes the encoder to compress the first picture in a new group of the type you just specified.

If you do not set this attribute, a default implementation-specific pattern is used. To select the default pattern after having selected a custom pattern, you pass a null pointer to `xil_cis_set_attribute()`.

You can also read the value of `COMPRESSOR_PATTERN`. To do this, you might use code similar to the following.

```
XilCis cis;
XilMpeg1Pattern *picture_pattern;

xil_cis_get_attribute(cis, "COMPRESSOR_PATTERN",
    (void **)&picture_pattern);
```

If, after this call, `picture_pattern` is a null pointer, the encoder uses the default picture pattern. Otherwise, `picture_pattern` points to a structure containing a pattern string and a repeat count. In the latter case, when you have finished using the structure, your application should free the storage that the XIL library allocated to hold the pattern string and then free the storage the library allocated to hold the structure.

COMPRESSOR_PEL_ASPECT_RATIO *(R/W)*

The value of the `COMPRESSOR_PEL_ASPECT_RATIO` attribute, along with the width and height of a picture in pixels, indicates the shape a picture is meant to have when it is displayed. Basically, the ratio of the width to the height of the displayed picture should be:

width in pixels / (height in pixels * aspect ratio)

If you supply an aspect ratio, this value is written to the sequence header for your bitstream and can be read later by an XIL-compliant MPEG-1 decompressor. The default value is implementation specific.

The valid values for `COMPRESSOR_PEL_ASPECT_RATIO` are contained in the following enumeration.

```
typedef enum {
    NullDefault,
    Ratio_1_0,                 /* 1.0    */
    Ratio_0_6735,              /* 0.6735 */
    Ratio_0_7031,              /* 0.7031 */
    Ratio_0_7615,              /* 0.7615 */
    Ratio_0_8055,              /* 0.8055 */
    Ratio_0_8437,              /* 0.8437 */
    Ratio_0_8935,              /* 0.8935 */
    Ratio_0_9157,              /* 0.9157 */
    Ratio_0_9815,              /* 0.9815 */
    Ratio_1_0255,              /* 1.0255 */
    Ratio_1_0695,              /* 1.0695 */
    Ratio_1_0950,              /* 1.0950 */
    Ratio_1_1575,              /* 1.1575 */
    Ratio_1_2015               /* 1.2015 */
} XilMpeg1PelAspectRatio;
```

The most important values in the enumeration are `Ratio_1_0`, `Ratio_0_9157`, and `Ratio_1_0950`. The value `Ratio_1_0` indicates that the pictures in the sequence can be shown without distortion on a computer monitor, which has square pixels. `Ratio_0_9157` indicates that pictures can be displayed without distortion on CCIR Rec. 601 625-line televisions. And `Ratio_1_0950` indicates that pictures can be displayed without distortion on CCIR Rec. 601 525-line televisions.

You can use the `NullDefault` value to reset the aspect ratio to its default value.

The following code shows `COMPRESSOR_PEL_ASPECT_RATIO` being set to `Ratio_1_0`.

```
XilCis cis;
XilMpeg1PelAspectRatio aspect_ratio = Ratio_1_0;

xil_cis_set_attribute(cis, "COMPRESSOR_PEL_ASPECT_RATIO",
    (void *)aspect_ratio);
```

You can read the current value of this attribute using code similar to the following.

```
XilCis cis;
XilMpeg1PelAspectRatio aspect_ratio;

xil_cis_get_attribute(cis, "COMPRESSOR_PEL_ASPECT_RATIO",
    (void **)&aspect_ratio);
```

If `aspect_ratio` equals `NullDefault` after this call, the aspect ratio for the sequence is the default aspect ratio defined by the implementors of the encoder.

COMPRESSOR_PICTURE_RATE *(R/W)*

You use the `COMPRESSOR_PICTURE_RATE` attribute to specify the picture rate of the sequence being encoded in pictures per second. This value and the value of `COMPRESSOR_BITS_PER_SECOND` enable the encoder to determine how many bits it can use to encode each picture.

The valid values for COMPRESSOR_PICTURE_RATE are the following enumeration constants. These constants correspond to the rates of commonly available sources of analog and digital video.

```
typedef enum {
    NullDefault,
    Rate_23_976,                /* 23.976 */
    Rate_24,                    /* 24.0   */
    Rate_25,                    /* 25.0   */
    Rate_29_97,                 /* 29.97  */
    Rate_30,                    /* 30.0   */
    Rate_50,                    /* 50.0   */
    Rate_59_94,                 /* 59.94  */
    Rate_60                     /* 60.0   */
} XilMpeg1PictureRate;
```

The default value of this attribute is implementation specific. The following code shows the value being set to Rate_29_97.

```
XilCis cis;
XilMpeg1PictureRate picture_rate = Rate_29_97;

xil_cis_set_attribute(cis, "COMPRESSOR_PICTURE_RATE",
    (void *)picture rate);
```

You can read the current value of this attribute using code similar to the following.

```
XilCis cis;
XilMpeg1PictureRate picture_rate;

xil_cis_get_attribute(cis, "COMPRESSOR_PICTURE_RATE",
    (void **)&picture_rate);
```

If picture_rate equals NullDefault after this call, the rate for the sequence is the default rate defined by the implementors of the encoder.

19

COMPRESSOR_SLICES_PER_PICTURE *(R/W)*

The value of the `COMPRESSOR_SLICES_PER_PICTURE` attribute provides a
hint to the encoder concerning the number of slices into which it should divide
the next picture it compresses; however, an XIL-compliant encoder may ignore
this suggested value.

The valid values for this attribute range from 1 to the number of macroblocks
in the picture. (The default value of the attribute is implementation specific.)
Because slices are designed to aid in error recovery from bitstream errors, the
value you choose should be dictated by your assessment of the likelihood that
data will be corrupted. If bitstream errors are rare, one slice per picture may be
sufficient. On the other hand, if errors are common, one slice per row of
macroblocks may be needed. Don't request more slices than you need because
there is a certain amount of overhead—for instance, a 40-bit slice
header—associated with each slice.

The following code shows `COMPRESSOR_SLICES_PER_PICTURE` being set to
13.

```
XilCis cis;
int slices = 13;

xil_cis_set_attribute(cis, "COMPRESSOR_SLICES_PER_PICTURE",
    (void *)slices);
```

You can also read the value of this attribute. To do this, you might use the
following code.

```
XilCis cis;
int slices;

xil_cis_get_attribute(cis, "COMPRESSOR_SLICES_PER_PICTURE",
    (void **)&slices);
```

If slices contains 0 after this call, the value of the attribute is set to an
implementation-specific default.

COMPRESSOR_TIME_CODE *(R/W)*

The COMPRESSOR_TIME_CODE attribute enables you to associate a time stamp with the first picture in the next group of pictures that is encoded. This time stamp contains the same information as an SMPTE time stamp.

Before actually setting the attribute, you must declare and assign values to the members of a structure of type XilMpeg1TimeCode.

```
typedef struct {
    Xil_boolean drop_frame_flag;
    Xil_unsigned32 hours;
    Xil_unsigned32 minutes;
    Xil_unsigned32 seconds;
    Xil_unsigned32 pictures;
} XilMpeg1TimeCode;
```

An application obtains the values for most of the members of this structure—hours, minutes, seconds, and pictures—by reading a SMPTE time code in the video being encoded. The hours member can have a value in the range 0 to 23, and minutes and seconds can have values in the range 0 to 59. The pictures member, where a picture is a subdivision of a second, can also have a value in the range 0 to 59.

The other member in the structure, drop_frame_flag, should be set to 0 unless the picture rate for the bitstream is 29.97. In this case, the flag can be set to 0 or 1. A setting of 0 indicates that the bitstream is played back as if the picture rate were 30 pictures per second. A setting of 1 indicates that a play-back application should not count certain pictures so that movie time will not get ahead of clock time. Specifically, the application should not count pictures 0 and 1 at the beginning of each minute except minutes 0, 10, 20, 30, 40, and 50.

The following code sets `COMPRESSOR_TIME_CODE` so that the time stamp 19:07:34:13 is associated with the next group of pictures encoded. The drop-frame flag is set to 0.

```
XilCis cis;
XilMpeg1TimeCode time_code = {0, 19, 7, 34, 13};

xil_cis_set_attribute(cis, "COMPRESSOR_TIME_CODE",
    (void *)&time_code);
```

To read the value of `COMPRESSOR_TIME_CODE`, you would use code similar to this.

```
XilCis cis;
XilMpeg1TimeCode *time_code;

xil_cis_get_attribute(cis, "COMPRESSOR_TIME_CODE",
    (void **)&time_code);
```

After this call, `time_code` points to a structure of type `XilMpeg1TimeCode` that contains the current time-code information, or it is a null pointer, indicating that `COMPRESSOR_TIME_CODE` has not been set. If the library successfully creates a time-code structure, your application is responsible for freeing the structure's storage.

Decompression Attributes

Only one of the attributes discussed in this section—`DECOMPRESSOR_QUALITY`—affects how the decoder works. The remaining attributes simply return information from the bitstream being decoded.

`DECOMPRESSOR_QUALITY` *(R/W)*

The value of the `DECOMPRESSOR_QUALITY` attribute provides a hint to the decompressor concerning how it should handle the trade-off between the quality of reconstructed pictures and the speed of decoding those pictures. An XIL-compliant decoder need not act on this hint.

≡ 19

The valid values for this attribute are integers in the range 1 to 100. A value of 100 is a request that the decoder produce the highest quality pictures possible, and a value of 1 is a request that the decompressor decode pictures as fast as possible. By default, DECOMPRESSOR_QUALITY is set to 100.

The following code shows DECOMPRESSOR_QUALITY being set to 90. This setting usually provides satisfactory quality and may result in a significant increase in speed (compared to a setting of 100).

```
XilCis cis;
int quality = 90;

xil_cis_set_attribute(cis, "DECOMPRESSOR_QUALITY",
    (void *)quality);
```

To read the value of this attribute, use code similar to this.

```
XilCis cis;
int quality;

xil_cis_get_attribute(cis, "DECOMPRESSOR_QUALITY",
    (void **)&quality);
```

DECOMPRESSOR_BROKEN_LINK *(R)*

The DECOMPRESSOR_BROKEN_LINK attribute is a read-only attribute that indicates whether B pictures at the beginning of the current group of pictures (before any I pictures) can be decoded correctly. Generally, of course, these pictures can be decoded correctly. However, it's possible that during an editing session, the preceding group of pictures—or maybe even just the last I or P picture in that group—was removed. If that type of editing takes place and a B picture at the beginning of the next group of pictures contains macroblocks that were forward predicted or bidirectionally predicted, the decoder is not able to decode the complete picture. In this case, a broken-link flag should have been set in the group-of-pictures header at the time of the editing.

To read the value of this attribute, you might use code similar to this.

```
XilCis cis;
Xil_boolean broken_link;

xil_cis_get_attribute(cis, "DECOMPRESSOR_BROKEN_LINK",
    (void **)&broken_link);
```

If the broken-link flag for the group of pictures currently being decoded is set, the value of `broken_link` is `TRUE`. In this case, any B pictures at the beginning of the group cannot be decoded correctly. If the flag is not set, the value is `FALSE`.

DECOMPRESSOR_CLOSED_GOP *(R)*

The `DECOMPRESSOR_CLOSED_GOP` attribute is a read-only attribute that indicates whether the group of pictures currently being decoded is a closed group or an open group. A closed group is one that begins with an I picture or with one or more B pictures that can be decoded without reference to the last picture in the preceding group. This type of group can be decoded independently. An open group, on the other hand, begins with one or more B pictures, at least one of which contains macroblocks that are forward or bidirectionally predicted. Such a B picture cannot be decoded if the preceding group of pictures is not available (for example, if it has been edited out of the bitstream).

To read the value of this attribute, you might use code similar to this.

```
XilCis cis;
Xil_boolean closed_gop;

xil_cis_get_attribute(cis, "DECOMPRESSOR_CLOSED_GOP",
    (void **)&closed_gop);
```

If the closed-group-of-pictures flag for the group is set, the value of `closed_gop` is `TRUE`. In this case, any B pictures at the beginning of the group can be decoded independently of the preceding group. If the flag is not set, the value is `FALSE`.

DECOMPRESSOR_FRAME_TYPE *(R)*

The `DECOMPRESSOR_FRAME_TYPE` attribute is a read-only attribute whose value describes the type of the last picture decoded. This type can be any one of the types included in the following enumeration.

```
typedef enum {
    I,      /* intraframe picture */
    P,      /* forward predicted picture */
    B,      /* bidirectionally predicted picture */
    D       /* DC picture */
} XilMpeg1FrameType;
```

For a description of I, P, and B pictures, see "I Pictures and P Pictures" on page 403 and "B Pictures" on page 403.

This code illustrates how you might read the value of `DECOMPRESSOR_FRAME_TYPE`.

```
XilCis cis;
XilMpeg1FrameType frame_type;

xil_cis_get_attribute(cis, "DECOMPRESSOR_FRAME_TYPE",
    (void **)&frame_type);
```

DECOMPRESSOR_PEL_ASPECT_RATIO_VALUE *(R)*

The `DECOMPRESSOR_PEL_ASPECT_RATIO_VALUE` attribute is a read-only attribute whose value is the pixel aspect ratio stored in the sequence header for the MPEG-1 sequence you're decoding. This value, along with the width and height of the pictures in the sequence, defines the shape of a decoded picture.

The value stored in this attribute will be one of the following constants:

- 0.6735
- 0.7031
- 0.7615
- 0.8055
- 0.8437
- 0.8935

- 0.9157
- 0.9815
- 1.0
- 1.0255
- 1.0695
- 1.0950
- 1.1575
- 1.2015

The key values in this list are 1.0, 0.9157, and 1.0950. A value of 1.0 indicates that the pixels are square and that the pictures in the sequence are meant to be displayed on computer monitors. The value 0.9157 means that decoded pictures can be displayed without distortion on 625-line 50 Hz televisions, and the value 1.0950 indicates that decoded pictures can be displayed without distortion on 525-line 60 Hz televisions.

To read the value of this attribute, use code similar to this.

```
XilCis cis;
float aspect_ratio;

xil_cis_get_attribute(cis,
    "DECOMPRESSOR_PEL_ASPECT_RATIO_VALUE",
    (void **)&aspect_ratio);
```

DECOMPRESSOR_PICTURE_RATE_VALUE *(R)*

The value of the DECOMPRESSOR_PICTURE_RATE_VALUE read-only attribute is the picture-rate value from the sequence header of the MPEG-1 sequence you're decoding. This picture-rate value is the number of pictures that should be decoded and displayed each second.

The value will be one of the following constants:

- 23.976
- 24.0
- 25.0
- 29.97
- 30.0
- 50.0
- 59.94

- 60.0

You can use code similar to this to read the value of this attribute.

```
XilCis cis;
float picture_rate;

xil_cis_get_attribute(cis, "DECOMPRESSOR_PICTURE_RATE_VALUE",
    (void **)&picture_rate);
```

DECOMPRESSOR_TEMPORAL_REFERENCE *(R)*

Because the pictures in a group of pictures may need to be decoded in one order and displayed in another order, each picture contains a temporal-reference value. This value indicates a picture's place in the display order for the group. Picture 0 is meant to be displayed first, picture 1 to be displayed second, and so on. The maximum value the temporal-reference field can accommodate is 1023; therefore, if a group of pictures contains more than 1024 pictures, the counter is reset to 0 for picture 1025 (or more generally, for picture $(1024 * n) + 1$).

The DECOMPRESSOR_TEMPORAL_REFERENCE attribute contains the temporal-reference value for the most recently decompressed picture. To read the value of this attribute, you might use the code below.

```
XilCis cis;
int temporal_reference;

xil_cis_get_attribute(cis, "DECOMPRESSOR_TEMPORAL_REFERENCE",
    (void **)&temporal_reference);
```

Note – In a legal MPEG-1 bitstream, the temporal reference values for the pictures in a group must be consecutive values (except in the case where the counter resets). However, it is not uncommon for bitstreams that have been edited to violate this requirement. That is, if the pictures with temporal-reference values between 12 and 24 have been cut from a group, you may find that the picture with a temporal reference of 11 is followed by one with a temporal reference of 25.

`DECOMPRESSOR_TIME_CODE` *(R)*

One field in a group-of-pictures header is a SMPTE time code. This time code is associated with the first picture (in display order) in the group. When you read the `DECOMPRESSOR_TIME_CODE` attribute, the decoder places information about this time code for the current group of pictures in a structure of type `XilMpeg1TimeCode`.

```
typedef struct {
    Xil_boolean drop_frame_flag;
    Xil_unsigned32 hours;
    Xil_unsigned32 minutes;
    Xil_unsigned32 seconds;
    Xil_unsigned32 pictures;
} XilMpeg1TimeCode;
```

The `hours` member has a value in the range 0 to 23; `minutes`, `seconds`, and `pictures` have values in the range 0 to 59.

You can read the value of this attribute using code similar to this.

```
XilCis cis;
XilMpeg1TimeCode *time_code;

xil_cis_get_attribute(cis, "DECOMPRESSOR_TIME_CODE",
    (void **)&time_code);
```

Your application is responsible for freeing the storage the library allocates to hold the time-code structure.

MPEG-1 Molecules

The XIL library includes a series of *molecules* that accelerate the playback of MPEG-1 bitstreams. These molecules are optimized routines that perform the jobs of two or more functions from the XIL API. You do not call such an optimized routine directly; rather, the library calls a molecule when your program calls a predefined sequence of XIL functions, sometimes with specific arguments.

For example, if your program calls `xil_decompress()` to decode an image stored in an MPEG-1 CIS and then calls `xil_ordered_dither()` to dither the decoded image from a 24- to an 8-bit image, the library might not call these two functions. Instead, it might call a molecule that performs the decompression and the dithering in an optimized way.

For further information on accelerating the playback of MPEG-1 bitstreams, see Chapter 23, "Acceleration in XIL Programs." For information about the MPEG-1 molecules that are available and information about how to call those molecules, see "XIL Molecules" on page 477.

CCITT Group 3 and Group 4 Codecs 20

The CCITT Group 3 and Group 4 compression methods were developed by the International Telegraph and Telephone Consultative Committee for the compression of bi-tonal document images. Like continuous-tone still images, document images can be stored and transmitted much more inexpensively if they are compressed. Originally, the CCITT's compression techniques were adopted by the developers of facsimile equipment. Today, however, the techniques are also used heavily by the makers of general document storage and retrieval systems.

This chapter has the following main sections:

How CCITT Group 3 and Group 4 Codecs Work

The Group 3 and Group 4 codecs take advantage of a couple of important characteristics of document images. One of these characteristics is that document images tend to consist of small amounts of black (letters or lines) on a white background. Thus, on a given scanline, there are likely to be long stretches of white pixels interrupted by shorter runs of black pixels.

This characteristic has led to the use of run-length encoding in the CCITT compressors. This coding method involves translating information about runs of white and black pixels (within a scanline) into code words stored in a

Huffman table. Run lengths that occur commonly are represented by short code words, and run lengths that occur infrequently are represented by longer code words.

This type of encoding generally produces quite a bit of compression. For instance, a run of 1024 white pixels (128 bytes) might be represented by a code word of 9 bits. This translation results in a compression ratio of about 114:1. However, the compression ratio achievable with run-length encoding is highly dependent on the image being compressed. For example, a noisy image might contain many short runs whose code words contain more bits than the runs themselves.

On standard text, the XIL library's Group 3 compressor achieves a compression ratio of about 5:1, and the library's Group 4 compressor achieves a ratio of about 10:1.

A second characteristic of document images is that the position of a transition from a black pixel to a white pixel (or vice versa) on one scanline is usually not more than a few pixels away from a corresponding transition on the preceding scanline. This characteristic is sometimes called *vertical coherence.* Because document images have this characteristic, once one scanline has been encoded, subsequent lines can be encoded by specifying the position of a black-to-white transition relative to the same transition on the preceding line, or relative to the last transition on the same line.

The XIL library's CCITT Group 3 compressor uses the run-length encoding method described above, and the Group 4 compressor relies almost entirely on the two-dimensional technique.

CCITT Group 3 and Group 4 Decompressor Attributes

Although other compression standards include size information (the image width, height, and number of bands) within the data bitstream, the fax standards do not. Thus, before you can decompress a fax CIS, you must set the decompressor attributes for the width, height, and number of bands. These attributes are discussed separately in the following sections. The attributes are identified as read-only (R), write-only (W), or read/write (RW).

WIDTH *(W)*

If you have put compressed data into your CIS using `xil_cis_put_bits()` or `xil_cis_put_bits_ptr()`, you must set the value of the `WIDTH` attribute to the width in pixels of the images to be decompressed. If you do not set it, its value is 0, and an error occurs when you call `xil_decompress()`.

The legal values for this attribute are integers in the range 0 to 32,767. This code fragment shows the `WIDTH` attribute being set to 1728.

```
XilCis cis;
int width = 1728;

xil_cis_set_attribute(cis, "WIDTH", (void *)width);
```

HEIGHT *(W)*

If you have put compressed data into your CIS using `xil_cis_put_bits()` or `xil_cis_put_bits_ptr()`, you must set the value of this attribute to the height in pixels of the images to be decompressed. If you do not set it, its value is 0, and an error occurs when you call `xil_decompress()`.

The legal values for this attribute are integers in the range 0 to 32,767. This code fragment shows the `HEIGHT` attribute being set to 2156.

```
XilCis cis;
int height = 2156;

xil_cis_set_attribute(cis, "HEIGHT", (void *)height);
```

BANDS *(W)*

If you have put compressed data into your CIS using `xil_cis_put_bits()` or `xil_cis_put_bits_ptr()`, you must set the value of this attribute to the number of bands in the images to be decompressed. If you do not set it, its value is 0, and an error occurs when you call `xil_decompress()`.

The legal values for this attribute are integers in the range 0 to 32,767. This code fragment shows the BANDS attribute being set to 1.

```
XilCis cis;
int bands = 1;

xil_cis_set_attribute(cis, "BANDS", (void *)bands);
```

Cell Codec *21* ≡

The Cell image compression technology, which was developed by Sun, has been optimized for the rapid decompression and display of images on simple hardware. Therefore, Cell compression achieves reasonable display quality on indexed-color frame buffers. The initial focus of the Cell technology is on Sun-to-Sun communications, where the benefits of fast decoding outweigh the benefits of standards. Other possible areas of application include media distributions on CD-ROM and multimedia mail applications.

Note – To see the quality of an image that has been compressed and decompressed using the Cell compressor, see Color Plate 4.

This chapter has the following main sections:

≡ 21

How the Cell Codec Works

The Cell encoding process transforms individual video frames into a bytestream that is displayable with the Cell decompressor. Normally, the encoder works with RGB images; however, it can also handle XIL images whose color-space attribute is set to `ycc601` or `ycc709`. The decompressor always produces RGB images.

In the first step of the encoding process, video images are analyzed to produce an appropriate colormap to represent the frames to be encoded (unless the programmer has already specified one). This step allows the specification of the colormap size in order to leave colors unused. This strategy enhances cooperation with the window manager and other applications. Cell also provides for Adaptive Colormap Selection (ACS), in which a new colormap is generated when the current colormap becomes unsuitable. This colormap can be used in subsequent frames.

Given the images and the colormap, the second step is to encode the individual frames into a Cell bytestream. The basic coding scheme used in Cell encoding is based on an image coding method called Block Truncation Coding (BTC). A 4-by-4 region of pixels (a *cell*) from an image is represented by 2 colors and a 16-bit mask; see Figure 21-1.

Figure 21-1 Cell Compression

The mask indicates which color to place at each of the pixel positions. The mask and colors may be chosen to maintain certain statistics of the cell, or they may be chosen to reduce contouring in a manner similar to ordered dithering.

The primary advantage of this coding method lies in the similarity of the decoding process to the operation of character fonting. The character display process for a color frame buffer takes as input a foreground color, a

background color, and a mask that specifies which color to use at each pixel. Because this function is so important to the window system, it is often implemented as a display primitive in graphics accelerators. The Cell compression technique uses this existing capability to provide full-motion video decoding with no special hardware or modifications to the window system.

There are actually two different encoding methods used to generate the encoded bytestream. The BTC encoding method, chooses a 4-by-4 binary pixel mask and two colors (a foreground and a background color), while attempting to maintain the mean and variance of the luminance within the block. The Dither method determines the pair of colors from the colormap that produce the least error when dithered over the 4-by-4 region. The BTC method is fastest and usually produces good results; the Dither method is slower, but is less likely to produce noticeable contours in regions of slow color variation.

Finally, encoded frames are combined into a frame sequence in the interframe compression step, where the compression ratio can be increased by anywhere from a factor of two to a factor of five. The colors and mask in each cell are compared to those used in the previous frames of the movie. If the colors and mask match to within a certain tolerance, a special skip code is generated. Runs of skip codes are combined to further reduce the bytestream. If either a single color changes or the mask changes, special escape codes are sent to update the changed data. Changes in the colormap are also detected by the interframe encoder, which causes special codes to be inserted into the stream to update the colormap.

Note – Details about the makeup of a Cell bytestream can be found in Appendix D, "Cell and CellB Bytestream Definitions."

Choosing a Colormap

The compressor chooses which colormap to use to encode the current image in one of three ways. If ACS is enabled and a new colormap has not been associated with the compressor since the last call to `xil_compress()`, the compressor uses a colormap adapted to the current frame. When ACS is disabled, the compressor uses the colormap associated with the `COMPRESSOR_COLORMAP` attribute, if that attribute has been set. If ACS is disabled and the `COMPRESSOR_COLORMAP` attribute has not been set, the compressor calls `xil_choose_colormap()` to generate an optimal colormap

for the image. When this optimal colormap is created, it is associated with the `COMPRESSOR_COLORMAP` attribute and will be used to encode subsequent frames.

Cell Compression Ratios

The basic Cell compression technique achieves a compression ratio of 2 bits per pixel. Additional encodings further reduce the data required per pixel. First, the compressor can use code words to specify a run of constant intensity cells. This type of coding is very effective in synthetic imagery, often leading to rates of less than 1 bit per pixel. The decompressor can also optimize the decompression of these constant colored cells by using rectangle-fill hardware. A second coding technique involves the introduction of frame-to-frame coherence (interframe coding), by instructing the decoder to skip over cells that are identical to those in the previous frame. These skip codes are implemented by simply updating the current writing position to skip over the number of cells specified.

The combination of the basic Cell encoding with run codes and skip codes can lead to compression ratios of about 0.5 bits per pixel.

Image Types

The Cell compressor is designed to work with 3-band RGB and YC_bC_r images. The width and height of these images must be divisible by four. The Cell decompressor always produces RGB images.

Creating a Cell CIS

Before you can use the Cell codec to compress images or decompress a Cell bytestream, you must create a Cell CIS. You do this by passing the compressor name `Cell` to the `xil_cis_create()` function as shown in this code fragment.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "Cell");
```

≡ *21*

Cell Codec Attributes

As discussed in "General CIS Attributes" on page 303, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for CISs attached to a Cell codec. You set compressor-specific attributes using the `xil_cis_set_attribute()` function, and you read compressor-specific attributes using the `xil_cis_get_attribute()` function.

Attributes are identified as read-only (R), write-only (W), or read/write (R/W). The example, or examples, that conclude the discussion of each attribute indicate how the attribute can be used.

The Cell attributes can be broadly grouped into those that affect compression and those that affect decompression. The attributes are discussed under these headings.

Compression Attributes

Setting any of the following attributes affects how the Cell compressor compresses images.

`BITS_PER_SECOND` *(R/W)*

The `BITS_PER_SECOND` attribute controls the size of the bytestream the Cell compressor produces. You specify the maximum number of bits the compressor can use to encode one second's worth of video, and the compressor guarantees that it will meet this bit-rate requirement on a frame-group basis (where a frame group is a key frame and all the ensuing frames up to the next key frame). That is, if every sixth frame is a key frame and your video was captured at 30 frames per second, the compressor encodes each frame group in a maximum of `BITS_PER_SECOND` divided by 5.

Note – You use the `COMPRESSOR_FRAME_RATE` attribute to indicate the rate at which your images were captured.

Setting `BITS_PER_SECOND` to 0, the default value, disables bit-rate control. In addition, if you set `BITS_PER_SECOND` to a rate lower than the compressor can achieve, an error is generated, and bit-rate control is disabled.

This code shows the bit-rate being set to 1152000 bits per second. This is the rate necessary to encode 30 320-by-240 frames at half a bit per pixel.

```
XilCis cis;
int bit_rate;

bit_rate = 1152000;
xil_cis_set_attribute(cis, "BITS_PER_SECOND",
    (void *)bit_rate);
```

The code below reads the value of the BITS_PER_SECOND attribute.

```
XilCis cis;
int bit_rate;

xil_cis_get_attribute(cis, "BITS_PER_SECOND",
    (void **)&bit_rate);
```

COLORMAP_ADAPTION *(R/W)*

The value of the COLORMAP_ADAPTION attribute determines whether the Cell compressor's ACS feature is enabled. When enabled, ACS looks at the colors in the current image to generate a colormap for the next image to be compressed. Thus, each frame in a movie can have its own colormap.

The possible values for this attribute are TRUE and FALSE, which are values of type Xil_boolean. A setting of TRUE, the default value, enables ACS, and FALSE disables it.

This code shows the COLORMAP_ADAPTION attribute being set to FALSE.

```
XilCis cis;

xil_cis_set_attribute(cis, "COLORMAP_ADAPTION", (void *)FALSE);
```

The code below reads the value of the attribute.

```
Xil_boolean acs_enabled;

xil_cis_get_attribute(cis, "COLORMAP_ADAPTION",
    (void **)&acs_enabled);
```

COMPRESSOR_COLORMAP *(W)*

The COMPRESSOR_COLORMAP attribute specifies the colormap the Cell compressor should use as it encodes images. To set this attribute, you pass an XIL lookup table (a data structure of type XilLookup) that contains 8-bit indexes on the input side and 24-bit RGB values on the output side to xil_cis_set_attribute(). The default value of this attribute is NULL.

This code shows the attribute being set.

```
XilCis cis;
XilLookup colormap;

xil_cis_set_attribute(cis, "COMPRESSOR_COLORMAP",
    (void *)colormap);
```

COMPRESSOR_FRAME_RATE *(R/W)*

Use the COMPRESSOR_FRAME_RATE attribute to let the compressor know the rate at which the images to be compressed were captured. Express this rate in microseconds per frame (a microsecond is one millionth of a second). The default value is 33333, which indicates that the frames were captured at 30 frames per second.

The frame rate you supply, or the default frame rate, is encoded in the Cell bytestream. When you play your movie back, you can read the value of the DECOMPRESSOR_FRAME_RATE attribute to determine the rate at which frames should be decompressed.

This code fragment shows the frame rate being set to 66666 microseconds per frame (15 frames per second).

```
XilCis cis;
Xil_unsigned32 microseconds;

microseconds = 66666;
xil_cis_set_attribute(cis, "COMPRESSOR_FRAME_RATE",
    (void *)microseconds);
```

COMPRESSOR_MAX_CMAP_SIZE *(R/W)*

The COMPRESSOR_MAX_CMAP_SIZE attribute is an integer that defines the maximum number of entries in the colormap, or colormaps, encoded in the Cell bytestream. If ACS is enabled, this attribute limits the size of the colormaps produced by the compressor. If ACS is disabled, the attribute limits the size of the colormap you can pass to the compressor using the COMPRESSOR_COLORMAP attribute. If you pass a colormap with more than the maximum number of entries, the colormap is truncated.

When you first create a Cell CIS, the COMPRESSOR_MAX_CMAP_SIZE attribute is set to -1, which indicates that the attribute is settable at this point. To set the attribute to a value other than 256, you must set it prior to your application's first call to xil_compress(). If you have not set the attribute by that point, it defaults to 256. Also, note that the colormap size can only be set once during the life of a CIS.

Note – When decompressing a Cell bytestream, you can determine the maximum colormap size by reading the CIS attribute DECOMPRESSOR_MAX_CMAP_SIZE. Knowing this maximum enables you to allocate the appropriate number of X colorcells to hold the colormap.

This code shows the COMPRESSOR_MAX_CMAP_SIZE attribute being set to 240.

```
XilCis cis;
int cmap_size = 240;

xil_cis_set_attribute(cis, "COMPRESSOR_MAX_CMAP_SIZE",
    (void *)cmap_size);
```

The code below reads the value of the attribute.

```
XilCis cis;
int cmap_size;

xil_cis_get_attribute(cis, "COMPRESSOR_MAX_CMAP_SIZE",
    (void **)&cmap_size);
```

COMPRESSOR_USER_DATA *(W)*

This attribute writes user data to a Cell bytestream. This user data can be anything you choose as long as it does not exceed 8 Kbytes. It can be an XGL rendering or an executable. The attribute's main purpose, however, is to enable you to store audio data in the bytestream.

Before setting this attribute, you must assign values to the members of a structure of type `XilCellUserData`. The definition of this structure is shown here.

```
typedef struct {
    Xil_unsigned8 *data;
    Xil_unsigned32 length;
} XilCellUserData;
```

The `data` member is a pointer to the user data, and the `length` member specifies the length of the data in bytes.

Once you've set up this structure, you can set the attribute using code similar to this fragment.

```
XilCis cis;
XilCellUserData user_data;

xil_cis_set_attribute(cis, "COMPRESSOR_USER_DATA",
    (void *)&user_data);
```

After you set the attribute, the next time your application calls `xil_compress()`, the Cell compressor:

- Writes the user data to the bytestream just before the compressed image data.

- Clears the setting of the attribute.

Thus, each time you set `COMPRESSOR_USER_DATA`, the compressor writes data to the bytestream only once.

To read user data from a Cell bytestream, you must read the `DECOMPRESSOR_USER_DATA` attribute.

ENCODING_TYPE *(R/W)*

The `ENCODING_TYPE` attribute indicates whether a Cell compressor uses the BTC or the Dither method. BTC chooses a 4-by-4 binary pixel mask and two colors (a foreground and a background color), while attempting to maintain the mean and variance of the luminance within the block. The Dither technique determines the pair of colors from the colormap that produce the least error when dithered over the 4-by-4 region. The BTC method is fastest, and usually produces good results; the Dither method is slower, but is less likely to produce noticeable contours in regions of slow color variation.

The value of the attribute can be either of the enumeration constants shown here. The default value is `BTC`.

```
typedef enum {
    BTC, DITHER
} XilCellEncodingType;
```

The following call shows the `ENCODING_TYPE` attribute being set to `DITHER`.

```
XilCis cis;

xil_cis_set_attribute(cis, "ENCODING_TYPE", (void *)DITHER);
```

This code shows the attribute being read.

```
XilCis cis;
XilCellEncodingType encode_type;

xil_cis_get_attribute(cis, "ENCODING_TYPE",
    (void **)&encode_type);
```

KEYFRAME_INTERVAL *(R/W)*

A key frame in a Cell bytestream is one that contains a bytestream information header and a colormap and uses no interframe escape codes. The KEYFRAME_INTERVAL attribute is an integer (type int) that specifies how frequently the compressor should encode key frames in the bytestream. That is, if the value of the attribute is 6, every sixth frame will be a key frame. The default value of the attribute is 6.

If you set the KEYFRAME_INTERVAL attribute to 0, no key frames are encoded in the resulting Cell bytestream. In this case, bit-rate control is disabled. (For further information about bit-rate control, see "BITS_PER_SECOND (R/W)" on page 444.)

This code shows KEYFRAME_INTERVAL being set to 10.

```
XilCis cis;
int key_frame = 10;

xil_cis_set_attribute(cis, "KEYFRAME_INTERVAL",
    (void *)key_frame);
```

The code below reads the value of the KEYFRAME_INTERVAL attribute.

```
XilCis cis;
int key_frame;

xil_cis_get_attribute(cis, "KEYFRAME_INTERVAL",
    (void **)&key_frame);
```

`TEMPORAL_FILTERING` *(R/W)*

This attribute turns on or off a form of temporal filtering. When this filtering is turned on, the compressor does not encode a new value for a pixel in frame *n* if the value of that pixel is within a certain tolerance of the same pixel in frame *n* - 1. Having the filter on generally reduces noise in an image sequence and also significantly reduces the size of the bytestream the compressor produces. The primary reason to turn the filter off would be to eliminate ghosting artifacts.

By default, the attribute is set to `TRUE`.

This code shows `TEMPORAL_FILTERING` being set to `FALSE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "TEMPORAL_FILTERING", (void *)FALSE);
```

This code reads the value of the `TEMPORAL_FILTERING` attribute.

```
XilCis cis;
Xil_boolean status;

xil_cis_get_attribute(cis, "TEMPORAL_FILTERING",
    (void **)&status);
```

Decompression Attributes

Setting any of the following attributes affects the operation of the Cell decompressor.

`DECOMPRESSOR_COLORMAP` *(R/W)*

When playing a Cell movie by decompressing frames and then using `xil_nearest_color()` to convert the resulting RGB images to 8-bit images, you must set this attribute if you want your playback code to be accelerated. The attribute specifies the XIL lookup table that `xil_nearest_color()`

should use when doing its 24-bit to 8-bit conversion. Of course, your application also needs to write the values stored in this lookup table to the X colormap your application is using.

As the Cell decompressor decompresses frames, it may change the values in the lookup table. (By default, the lookup table is read-only, but you can make it writable by setting the RDWR_INDICES attribute, which is discussed in "RDWR_INDICES (W)" on page 454.) Therefore, your application may need to call the xil_lookup_get_version() function to check the version number of the lookup table after each call to xil_decompress(). A change in version number means that the values in the lookup table have changed. If the values have changed, your application must ensure that corresponding changes are made in its X colormap before displaying the most recently decompressed frame.

Note – "Playing Cell Movies" on page 338 discusses an example program that sets this attribute and checks the version number of the lookup table.

You can also read the value of the DECOMPRESSOR_COLORMAP attribute. If it has been set, you get back a handle to an XIL lookup table (the decompressor's colormap). If it has not been set, xil_cis_get_attribute() returns NULL.

This code fragment shows DECOMPRESSOR_COLORMAP being set.

```
XilCis cis;
XilLookup colormap;

xil_cis_set_attribute(cis, "DECOMPRESSOR_COLORMAP",
    (void *)colormap);
```

This code reads the value of the attribute.

```
XilCis cis;
XilLookup colormap;

xil_cis_get_attribute(cis, "DECOMPRESSOR_COLORMAP",
    (void **)&colormap);
```

DECOMPRESSOR_FRAME_RATE *(R)*

If the `COMPRESSOR_FRAME_RATE` attribute was set when the Cell compressor encoded your movie, you can use the `DECOMPRESSOR_FRAME_RATE` attribute to determine the rate, in microseconds per frame, at which the frames in the movie were captured. The attribute may contain different values at different points in the Cell bytestream.

If the `COMPRESSOR_FRAME_RATE` attribute was not set when the Cell compressor encoded your movie, the `DECOMPRESSOR_FRAME_RATE` attribute contains the default value 33333 (30 frames per second).

This code fragment shows the `DECOMPRESSOR_FRAME_RATE` attribute being read.

```
XilCis cis;
Xil_unsigned32 frame_rate;

xil_cis_get_attribute(cis, "DECOMPRESSOR_FRAME_RATE",
    (void **)&frame_rate);
```

DECOMPRESSOR_MAX_CMAP_SIZE *(R)*

This is a read-only attribute that indicates the size of the colormap you should use when playing back your Cell movie. It's important to use the smallest colormap possible because your application must create an X colormap that is the same size as the decompressor's XIL colormap. If the X colormap is too large, colormap flashing may occur when it is installed.

Information about a movie's maximum colormap size is encoded in the Cell bytestream that represents the movie. Therefore, it is an error to read this attribute of an empty CIS.

This code shows the `DECOMPRESSOR_MAX_CMAP_SIZE` attribute being read.

```
XilCis cis;
int cmapsize;

xil_cis_get_attribute(cis, "DECOMPRESSOR_MAX_CMAP_SIZE",
    (void **)&cmapsize);
```

DECOMPRESSOR_USER_DATA *(R)*

This attribute holds a pointer to any user data that was encoded with the most recently decompressed image. To get this pointer, you might use code similar to this.

```
XilCis cis;
XilCellUserData user_data;

xil_cis_get_attribute(cis, "DECOMPRESSOR_USER_DATA",
    (void **)&user_data);
```

If user data was encoded with the last image that was decompressed, `user_data.data` is a pointer to that data, and `user_data.length` is the length of the data in bytes. If no user data was encoded with that image, `user_data.data` is `NULL`, and `user_data.length` is 0.

Any data pointer you get by using this attribute is only valid until the next call to `xil_decompress()` so you need to copy it if it must be retained for a longer time.

RDWR_INDICES *(W)*

If your Cell movie-playback application uses the `xil_nearest_color()` function to convert 24-bit RGB images to 8-bit pseudocolor images, `xil_nearest_color()` uses the decompressor's colormap (XIL lookup table) to perform this conversion. For your application to display images using the best colors possible, the Cell decompressor must be able to alter the color values in this lookup table as the colors in the frames it is decompressing change. By default, however, the colormap is read only.

The `RDWR_INDICES` attribute enables you to make the colormap writable for a subset of its entries. Before you actually set the attribute, you must set up a structure of type `XilIndexList`. The definition of this structure is shown here.

```
typedef struct {
    Xil_unsigned32 *pixels;
    Xil_unsigned16 ncolors;
} XilIndexList;
```

You set the `ncolors` member of the structure to indicate how many of the
elements in the colormap are being made writable. The `pixels` array lists the
pixel values in the colormap whose RGB values the decompressor may change.

Note – If you allow the decompressor to alter the contents of its colormap, you
must check the version number of the colormap after each call to
`xil_decompress()`. You check this version number using the
`xil_lookup_get_version()` function. If the version number has
changed—that is, the contents of the colormap have changed—you must make
corresponding changes in your application's X colormap before displaying the
most recently decompressed frame.

Calls to `RDWR_INDICES` are not cumulative. Only the RGB values associated
with the pixel values specified in your most recent setting of the attribute are
writable.

The following code fragment shows the `RDWR_INDICES` attribute being set.

```
XilCis cis;
XilIndexList indexes;

xil_cis_set_attribute(cis, "RDWR_INDICES", (void *)&indexes);
```

To see how the structure `indexes` might be filled out, see the
`xilcis_color.c` function in the
`/usr/openwin/demo/xil/movie_player_example` directory.

Cell Molecules

The XIL library includes a series of *molecules* that accelerate the playback of
Cell movies. These molecules are optimized functions that perform the jobs of
two or more functions from the XIL API. You do not call such an optimized
function directly; rather, the library calls a molecule when your program calls a
predefined sequence of XIL functions, sometimes with specific arguments.
(These sequences of functions are discussed later in this section.)

For example, if your program calls `xil_decompress()` to decode an image stored in a Cell CIS and then calls `xil_nearest_color()` to convert the decoded image from a 24- to an 8-bit image, the library may not call these two functions. Instead, it may call a molecule that performs the decompression and the conversion in an optimized way.

Note – This replacement of two or more *atomic* functions by a molecule is made possible by the XIL library's deferred-execution scheme, discussed in Chapter 23, "Acceleration in XIL Programs." This chapter also contains a more detailed definition of molecules than the one presented here and explains how to determine whether a molecule you want to call is actually being executed.

The library's Cell-decompression molecules enable you to accelerate the playback of a Cell bytestream on:

- A one-bit destination image
- A display image associated with a GX frame buffer (available only on local GX frame buffer screens)
- An 8-bit destination image other than a GX display image

The procedures you follow to call these molecules are documented later in this chapter, see "Calling Cell Molecules" on page 457. Before moving on to that subject, however, this chapter lists some general rules you must follow to execute any decompression molecule.

Rules for Calling Decompression Molecules

This section goes over the global rules for calling decompression molecules.

- As mentioned earlier, you must call a predefined sequence of XIL functions, sometimes with specific arguments. This sequence is replaced by the molecule.

Note – The calls to the functions in the sequence do not have to be consecutive statements in your program. For instance, notice how the example shown in "Other 8-Bit Destination Images" on page 461," calls one function in the sequence, `xil_decompress()`, and then immediately calls an XIL function that is not part of the sequence and checks that function's return value. Depending on this return value, the example may then call a function that loads an X colormap before it gets around to calling the second function in the

---

sequence. The key here is that between calls to the functions in the sequence, the program cannot call any XIL functions whose execution the library defers. For more information about the XIL library's deferred-execution scheme, see "What Is Deferred Execution?" on page 473.

- The images being decompressed and the molecule's destination image must have the same width and height, except when the molecule performs a scaling operation. In that case, the destination image must have the same dimensions as the scaled version of the source image.

- The destination image must not have a region of interest. The destination will not have a region of interest unless you have explicitly set its region-of-interest attribute.

- The destination image must have an origin of 0.0, 0.0. This is the destination image's default origin.

- If a molecule uses an intermediate image, it too must have an origin of 0.0, 0.0 and no region of interest.

If any of these conditions is not met, the XIL functions in your playback code are executed individually, and are not replaced by a molecule. This has a significant impact on the speed at which your movie is played.

Calling Cell Molecules

Cell decompression molecules have been implemented for applications that play back movies using 1- and 8-bit destination images (usually display images). These molecules and the series of XIL functions you must call for these molecules to be executed are discussed below.

One-Bit Destination Image

One molecule has been defined for playing back Cell movies using a 1-bit destination image. This molecule is appropriate for playing movies on monochrome displays.

The molecule performs several tasks:

- Decompresses an image

- Converts the RGB image to a 1-band image by extracting the luminance of the RGB image

≡ 21

- Rescales the values in the 8-bit grayscale image so that they fall in the range 0 to 255

- *Optionally* zooms the 8-bit image by a factor of 2 in both the *x* and *y* dimensions

- Performs an ordered dither on the 8-bit image to produce a 1-bit image

For this molecule to be called, your application must include code similar to this.

```
XilColorspace rgblinear, ylinear;
float scale[1], offset[1];
XilLookup colorcube;
int mult[1] = {-1};
unsigned short dims[1] = 2;
XilDitherMask dmask;

rgblinear = xil_colorspace_get_by_name(state, "rgblinear");
ylinear = xil_colorspace_get_by_name(state, "ylinear");
xil_set_colorspace(imageRGB_24, rgblinear);
xil_set_colorspace(imageY_8, ylinear);
scale[0] = 255.0 / (235.0 - 16.0);
offset[0] = -16.0 * scale[0];

/* Basically, the colorcube to be used for the ordered dither
   must be defined as shown here. The molecule will still
   execute if mult[0] is 1 instead of -1; however,
   this change will cause images to be
   displayed in reverse video.  */

colorcube = xil_colorcube_create(state, XIL_BIT, XIL_BYTE, 1, 0,
    mult, dims);

/* The dither mask to be used in the ordered dither must be an
   8-by-8 mask. You can change the values in the mask if you
   want to. */
dmask = xil_dithermask_get_by_name(state, "dm881");

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageRGB_24);
xil_color_convert(imageRGB_24, imageY_8);
xil_rescale(imageY_8, imageY_8, scale, offset);
if (ZOOM) {
    xil_scale(imageY_8, zoom_imageY_8, "nearest", 2.0, 2.0);
    xil_ordered_dither(zoom_imageY_8, zoom_image_1, colorcube,
    dmask);
}
else
    xil_ordered_dither(imageY_8, image_1, colorcube, dmask);
/* MOLECULE ENDS HERE */
```

Note – In the code above, some arguments are shown in boldface. These arguments must be typed as shown (using the exact characters, but without the boldface attribute) for the molecule to execute correctly.

SPARC: Eight-Bit Display Image (GX)

Another molecule has been defined specifically for playing back Cell movies on *local* GX frame-buffer screens. This molecule performs these tasks:

- Decompresses an image

- Converts the RGB image to an 8-bit image by finding the closest match for the RGB values in the source image in a lookup table

- *Optionally* zooms the 8-bit image by a factor of 2 in both the *x* and *y* dimensions

Note – Several of the molecule rules presented in "Rules for Calling Decompression Molecules" on page 456 do not apply to the version of this molecule that does not perform the zooming operation. If no zooming is requested, the destination image for the molecule does not need to be the same size as the images stored in the Cell CIS. In addition, the destination image can have a region of interest and an origin other than 0.0, 0.0.

For this molecule to be called, your application must use this code.

```
XilLookup cmap;
XilIndexList indexlist;

/* You must set this attribute, or the molecule will not
   execute. cmap is the lookup table that will be passed
   to xil_nearest_color().*/
xil_cis_set_attribute(cis, "DECOMPRESSOR_COLORMAP",
    (void *)cmap);

/* To get the best results, you should set this attribute
   so that the decompressor can change any of the entries in
   cmap. */
xil_cis_set_attribute(cis, "RDWR_INDICES", (void *)&indexlist);

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageRGB_24);

/* This checking of the lookup table's version number is
   necessary, but is not part of the molecule. */
if (lu_version != xil_lookup_get_version(cmap)) {
    /* Include code to write the appropriate values from cmap to
       the application's X colormap */
    ...
    lu_version = xil_lookup_get_version(cmap);
}

if (ZOOM) {
    xil_nearest_color(imageRGB_24, image_8, cmap);
    xil_scale(image_8, zoom_image_GX, "nearest", 2.0, 2.0);
}
else
    xil_nearest_color(imageRGB_24, image_GX, cmap);
/* MOLECULE ENDS HERE */
```

Other 8-Bit Destination Images

If you want to play back your Cell movie using an 8-bit destination image
other than a local GX display image, you can use one of three molecules. The
first two molecules use a nearest-color strategy to convert images from 24 bits

to 8 bits, and the third molecule uses an ordered dither. All three molecules require that you set the DECOMPRESSOR_COLORMAP CIS attribute before you execute them.

Nearest-Color Approach

Functionally the first molecule in this class is identical to the basic molecule (no zoom) designed for playing movies on a local GX frame buffer. (See "Eight-Bit Display Image (GX)" on page 460.) However, it is not as fast as the GX version, so displaying a movie on an indexed-color frame buffer other than the GX is not quite as fast as on a GX.

For this molecule to execute, your application must use the code shown in "Eight-Bit Display Image (GX)" on page 460. In addition:

- The destination image must be stored on a 32-bit aligned boundary. It is stored on such a boundary unless it is a child image created with an *x* offset from its parent that is not a multiple of four.

- The pixel stride in the destination image must be one. This is always true unless the destination image is a single-band child of a multiband parent.

The second molecule in this class rapidly decompresses Cell images and displays them in a small window. It performs the following tasks:

- Decompresses an image

- Converts the RGB image to an 8-bit image by finding the closest match for the RGB values in the source image in a lookup table

- Scales the image down using *x* and *y* scale factors of $1 / (4 * n)$, where *n* is a positive integer

The destination image must be at least 4-by-4 pixels in size.

For this molecule to be called, your application must use code similar to this.

```
/* MOLECULE BEGINS HERE */
xil_decompress(cis, imageRGB_24);
if (lu_version != xil_lookup_get_version(cmap)) {
    /* Include code to write the appropriate values from cmap to
       the application's X colormap. */
    ...
    lu_version = xil_lookup_get_version(cmap);
}
xil_nearest_color(imageRGB_24, image_8, cmap);
xil_scale(image_8, small_image_8, "nearest", .25, .25);
/* MOLECULE ENDS HERE */
```

Ordered-Dither Approach

This molecule performs the following tasks:

- Decompresses an image

- *Optionally* zooms the RGB image by a factor of 2 in the *x* and *y* dimensions

- Dithers the 3-band image to a 1-band image using a colorcube

For the molecule to execute, the destination image must be 32-bit aligned and must have a pixel stride of one.

☰ *21*

To call this molecule, your application must use code similar to this.

```
/* Create a colorcube and dither mask. All legal XIL
   colorcubes are supported, including those that have
   decreasing ramps in one or more bands. The dither mask
   must be a 4-by-4 mask, but there are no restrictions
   regarding its contents. */
...

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageRGB_24);
if (ZOOM) {
    xil_scale(imageRGB_24, zoom_imageRGB_24, "nearest", 2.0,
        2.0);
    xil_ordered_dither(zoom_imageRGB_24, zoom_image_8, colorcube,
        dithermask);
}
else
    xil_ordered_dither(imageRGB_24, image_8, colorcube,
        dithermask);
/* MOLECULE ENDS HERE */
```

CellB Codec 22 ≡

The Cell codec discussed in Chapter 21, "Cell Codec," is useful primarily in authoring applications. Its advantages are its fast decoding and the high quality of the images it produces, particularly on indexed-color frame buffers. The CellB codec, which derives from its Cell counterpart, is intended for use primarily in videoconferencing applications. It features a greater balance between the time spent compressing and decompressing images than the Cell codec and employs a fixed colormap. The CellB codec's strengths include:

- Software compression at interactive rates
- Very fast decoding and display, especially on indexed-color frame buffers
- Low rates of CPU use
- Good quality output

This chapter has the following main sections:

How the Codec Works	*page 466*
Creating a CellB CIS	*page 469*
CellB Decompression Attributes	*page 469*
CellB Molecules	*page 471*

≡ *22*

How the Codec Works

The CellB compressor works only on YC_bC_r images that conform to the guidelines set forth in CCIR Recommendation 601. The compressor performs *intraframe* compression by representing 4-by-4 blocks of pixels using cell codes. It performs *interframe* encoding using skip codes.

Cell Codes

The images you compress using the CellB compressor must have a width and height that are multiples of four because the compressor works with 4-by-4 cells of pixels. For each frame, the compressor begins with the cell in the upper-left corner and then proceeds from left to right. The compressor processes rows of cells in this way, moving from the top of the frame to the bottom.

When the compressor encodes a cell without reference to a cell in a preceding frame, it uses a four-byte cell code to represent the content of that cell. This cell code specifies two colors and includes a 16-bit bit mask that indicates which of the two colors should be used to represent each pixel in the cell. See Figure 22-1.

Index into table of chrominance values

16-bit mask

| | | C_B/C_R | Y/Y |

Index into table of luminance values

Figure 22-1   Cell Code

The two colors are encoded as follows. The compressor calculates the average C_b and C_r values for the 4-by-4 block. Then, it calculates an index into a table of 256 vectors in which each vector looks like the one shown in Figure 22-2.

Byte 0   Byte 1

C_B value | C_R value

Figure 22-2 Vectors in Chrominance Table

The values in the vector pointed to by the index are the pair of values in the table nearest to the mean C_b and C_r values for the cell. The compressor writes the index to this vector to the third byte of the cell code.

The compressor also analyzes the luminance values in the cell. First, it calculates the mean luminance for the cell. Second, it partitions the 16 luminance values in the cell into those values that fall below the mean and those that fall above the mean. Then, it calculates the average luminance in the two partitions.

After arriving at the average luminance values for the two partitions, the compressor calculates an index into a table of vectors of the form shown in Figure 22-3.

Byte 0   Byte 1

Y_0 value | Y_1 value

Figure 22-3 Vectors in Luminance Table

The values in the vector pointed to by the index are the pair of values in the table closest to the average luminance values for the two partitions. The compressor then writes the index to this vector to the fourth byte of the cell code.

The first color for a cell consists of the first byte of the cell's luminance vector and the chrominance values in the cell's chrominance vector. The second color consists of the second byte of the cell's luminance vector and the same chrominance values.

The bit mask shown in Figure 22-1 is filled out in this way. Each bit in the mask is associated with a pixel in the cell. If the luminance value for a pixel is below the mean luminance value for the cell, its bit is set to 0. This pixel will be represented by the first color when the cell is decompressed. If a pixel's luminance value is above the mean, its bit is set to 1.

Because each cell code represents the values of 16 pixels using 32 bits, compression using cell codes alone leads to a compression rate of 2 bits per pixel. To better this compression rate, the CellB compressor uses skip codes to achieve interframe compression.

Skip Codes

The CellB compressor encodes the first image in a sequence using cell codes exclusively. But after the first image, the compressor begins looking for cells in the current image that match—within a certain tolerance—the corresponding cells in the preceding image. Anywhere from 1 to 32 consecutive cells that match their counterparts in the previous image can be represented by a single 1-byte skip code.

The only restriction on the use of skip codes is that the cell at a particular set of coordinates cannot be skipped indefinitely. There are a couple of reasons for this restriction. First, a user might join a videoconference that is already in progress. Cells represented by skip codes in the first image the uer receives will not be displayed correctly, and this problem must be corrected within a certain period of time. Similarly, if the CellB bytestream is being sent over an unreliable transport and a packet of data is lost, the period of the resulting error should be limited.

The policy concerning skip codes is that a particular cell must be updated at least every *n* frames, where *n* is an implementation-specific maximum. The exact value used is selected randomly for each cell each time that cell is encoded using a cell code. A random number is used to prevent the periodic bit-rate increase that might result were each cell to be updated at a fixed interval.

In a typical videoconference, about 80 percent of the cells in the average frame are represented by skip codes. This ratio leads to an average compression rate of about .8 bits per pixel.

Note – For details about the makeup of a CellB bytestream, see Appendix D, "Cell and CellB Bytestream Definitions."

Creating a CellB CIS

Before you can use the CellB codec to compress images or decompress a CellB bytestream, you must create a CellB CIS. To do this you pass the compressor name `CellB` to the `xil_cis_create()` function, as shown in this code fragment.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "CellB");
```

CellB Decompression Attributes

As discussed in "General CIS Attributes" on page 219, there is a class of attributes that can be set for any CIS. There is also a set of attributes that are valid only for a CIS attached to a CellB codec. You set these codec-specific attributes using the `xil_cis_set_attribute()` function and read them using `xil_cis_get_attribute()`. Attributes are identified as read-only (R), write-only (W), or read/write (R/W).

`WIDTH` *and* `HEIGHT` *(W)*

If you have put compressed data into your CIS using `xil_cis_put_bits()` or `xil_cis_put_bits_ptr()`, you *must* set the values of these attributes to the width and height in pixels of the images to be decompressed. If you do not set these attributes, their values will be 0, and an error will occur if you:

- Create an XIL image into which to decompress the compressed image sequences in the CIS by calling the `xil_cis_get_output_type()` and `xil_create_from_type()` functions.

- Call `xil_decompress()` to decompress an image stored in the CIS.

The legal values for both attributes are integers in the range 4 to 32764 (`short int`). This code fragment shows the `WIDTH` attribute being set to 320 and the `HEIGHT` attribute being set to 240.

```
XilCis cis;
short width = 320;
short height = 240;

xil_cis_set_attribute(cis, "WIDTH", (void *)width);
xil_cis_set_attribute(cis, "HEIGHT", (void *)height);
```

`IGNORE_HISTORY` *(R/W)*

The `IGNORE_HISTORY` attribute affects your ability to seek forward and backward in a CellB bytestream. Such seeking is somewhat problematic in CellB because the codec relies so heavily on interframe encoding and does not require periodic key frames.

By default, `IGNORE_HISTORY` is set to `FALSE`. In this case, seeking backward is illegal because no past frame contains all the information necessary to reproduce an entire image. Seeking forward is possible, but to ensure that it can decode properly the frame you seek, the decoder must actually decode all the frames you "skip."

If you set `IGNORE_HISTORY` to `TRUE`, you're telling the decoder that it's acceptable if, after seeking, it does not produce a correct image. You're willing to let it decode only the cells described with cell codes in the frame you're seeking to, and to fill in bad cells as it decodes subsequent frames. In this case, seeking backward is legal, and seeking forward is faster than it would be otherwise. The drawback to setting `IGNORE_HISTORY` to `TRUE` is that you may have to decode a number of frames after seeking before you get a properly reconstructed picture. (The exact number of frames is implementation dependent.)

This code shows `IGNORE_HISTORY` being set to `TRUE`.

```
XilCis cis;

xil_cis_set_attribute(cis, "IGNORE_HISTORY", (void *)TRUE);
```

You can also read the value of this attribute using code similar to that shown here.

```
XilCis cis;
Xil_boolean history_status;

xil_cis_get_attribute(cis, "IGNORE_HISTORY",
    (void **)&history_status);
```

CellB Molecules

The XIL library includes a series of molecules that accelerate the playback of CellB bytestreams. These molecules are optimized routines that perform the jobs of two or more functions from the XIL API. You do not call such an optimized routine directly; rather, the library calls a molecule when your program calls a predefined sequence of XIL functions, sometimes with specific arguments.

For example, if your program calls `xil_decompress()` to decode an image stored in a CellB CIS and then calls `xil_ordered_dither()` to convert the decoded image from a 24- to an 8-bit image, the library may not call these two functions. Instead, it may call a molecule that performs the decompression and the dithering in an optimized way.

For more information about accelerating the playback of CellB bytestreams, see Chapter 23, "Acceleration in XIL Programs." For information about the CellB molecules that are available and information about how to call those molecules, see "XIL Molecules" on page 477.

≡ 22

Acceleration in XIL Programs 23 ≣

Obviously, much of the speed of an XIL program is determined by the speed of the individual functions that make up the API. However, XIL applications can also realize big performance improvements when the library is able to replace the execution of a sequence of API-level functions (or *atoms*) with the execution of an optimized function that performs the work of all the atoms in the sequence. This type of optimized function is called a *molecule.*

You do *not* call a molecule directly. Rather, the library itself recognizes sequences of atoms that can be replaced by a molecule and performs the replacement automatically. This type of replacement is made possible by the library's deferred-execution scheme.

This chapter has the following main sections:

What Is Deferred Execution?	*page 473*
XIL Molecules	*page 477*
Troubleshooting Molecules	*page 486*
Side Effects of Executing Molecules	*page 490*
XIL Functions That Relate to Deferred Execution	*page 491*

What Is Deferred Execution?

In general, when an atomic function that affects the state of an image or a compressed image sequence is called, the function is not executed immediately. Instead, information about the operation—such as the function called and the

arguments to the function—is stored by the library. This information continues to be stored until the library must produce a particular destination image—for example, because that image is to be displayed.

Suppose the library has stored the five atomic operations shown in Figure 23-1 and that by performing these operations, it could decompress an image from a Cell CIS and prepare that image for display on a monochrome display.

Figure 23-1 Stored Atomic Operations

When it must produce the destination of the ordered dither, instead of simply performing the five atomic functions, the library searches a list of molecules to see if all or part of this sequence can be replaced by a molecule. If the library finds a molecule that can perform this entire sequence of operations, the execution of the program proceeds as shown in Figure 23-2.

Flow of execution

xil_decompress()

xil_color_convert()

xil_rescale()

xil_scale()

xil_ordered_dither()

dither

decompress

color
convert

Molecule

scale

rescale

Figure 23-2 Replacing Atomic Functions With a Molecule

The molecule shown is executed and performs the jobs of all five atomic functions. None of the atomic functions is ever executed.

The ability to replace a series of atomic functions with a molecule like this can lead to dramatic increases in performance. Two reasons for this increase are that a molecule may:

- Need to page an image into memory only once, whereas the equivalent atomic functions would have paged the image into memory several times

- Not need to create temporary images that would be required by the equivalent atomic functions

XIL Molecules

The molecules available in the base XIL library can be grouped into two categories:

- Molecules that decompress images stored in a compressed image sequence. This is the largest category of molecules, and these molecules are available on all platforms that can run XIL applications.

- SPARC: Molecules that perform a common XIL operation and then display the results of the operation on a GX frame buffer. These molecules are available only on local GX frame-buffer screens.

- Molecules provided by third party hardware vebdors or other platform-specific acceleration. For example, the SunVideo™ product has a number of compression molecules and Ultra™ systems have a wide array of molecules to accelerate display on Ultra Creator systems.

In this section some general rules you must follow to execute XIL molecules are discussed first. (Particular molecules may require that you follow other rules as well.) Then the molecules that have been implemented in the categories mentioned in the preceding paragraph are discussed.

Rules for Executing Molecules

There are a few general rules you must follow to execute XIL molecules:

- The images that a molecule works with—source, intermediate, and destination—must have the same width and height. The principal exception to this rule occurs when a molecule performs a scaling operation. In that case, the destination image for the scale operation must have the same dimensions as the scaled source image.

- The images the molecule works with must have the same region of interest. For decompression molecules, this rule dictates that the molecule's destination image and any intermediate images have no region of interest because the images stored in the CIS cannot have a region of interest. In addition, if a molecule includes a scale operation, the images involved cannot have a region of interest. These images do not have a region of interest unless you have explicitly set their region-of-interest attributes.

- The images the molecule works with must have the same origin. For decompression molecules, this rule dictates that the molecule's destination image and any intermediate images have origins of 0.0, 0.0. This is true because the images stored in a CIS cannot have anything other than the default origin.

There are one or two exceptions to these rules among the decompression molecules. These exceptions are covered in "Video Decompression Molecules."

Video Decompression Molecules

The library's video decompression molecules decompress an image from a compressed image sequence, process that image in some way, and write it to a destination, usually a display image. This destination may have a depth of 1 bit (a monochrome display), 8 bits (an indexed-color display), or 24 bits (a true-color display). To display video at the best possible speed, it is important that you use these molecules. They provide much better performance than the equivalent set of atomic functions.

Eight-Bit Destination

This molecule is optimized where the molecule's destination image is an 8-bit display image associated with a local GX frame buffer. However, the molecule also accelerates the decompression and subsequent processing of images being written to other 8-bit destinations as well.

The molecule performs the following tasks:

- Decompresses an image from a CIS

- *Optionally* rescales the values in the three bands of the YC_bC_r image

- Dithers the 24-bit YC_bC_r image to an 8-bit pseudocolor image

- *Optionally* zooms the dithered image by a factor of 2 in both the *x* and *y* dimensions

There are certain restrictions on the use of this molecule. One is that the molecule can decompress images from a JPEG CIS only if those images are YC_bC_r images and the images are 4:1:1 interleaved in the bitstream. (CellB, H.261, and MPEG-1 CISs always contain YC_bC_r images.) In addition, in most cases, JPEG, H.261, and MPEG-1 CISs must contain images whose width and height are multiples of 16. Exceptions to this rule are discussed in "Exceptions to the Rules" on page 481.

Note – Arguments shown in boldface must be typed as shown for the molecule to execute correctly.

For this molecule to be executed, your application must include the following code.

```
float scale_values[3], offset_values[3];
XilLookup colorcube, colormap;
XilDitherMask dmask;

scale_values[0] = 255.0 / (235.0 - 16.0);
scale_values[1] = 255.0 / (240.0 - 16.0);
scale_values[2] = 255.0 / (240.0 - 16.0);
offset_values[0] = -16.0 * scale_values[0];
offset_values[1] = -16.0 * scale_values[1];
offset_values[2] = -16.0 * scale_values[2];
colorcube = xil_lookup_get_by_name(state, "cc855");
dmask = xil_dithermask_get_by_name(state, "dm443");

/* The lookup table yuv_to_rgb is matched with the colorcube
   cc855. If you use cc855 when dithering your images, you must
   install the 200 sets of RGB values in yuv_to_rgb in your
   application's X colormap before executing the molecule.
   Install the first set of RGB values at colorcell 54. */
colormap = xil_lookup_get_by_name(state, "yuv_to_rgb");
```

```
/* MOLECULE STARTS HERE */
xil_decompress(cis, imageYCC_24);

/* Perform this rescale if your image's color space is CCIR Rec.
   601 YCC. The rescale adjusts the range of the values in the
   three bands of the image to 0 to 255. The ordered dither
   operation that takes place later will produce the best results
   if the values are in this range. Note that YCC images produced
   by compressors that use the JFIF format have already been
   rescaled to 0 to 255 and, therefore, should not be rescaled
   here. */
if (CCIR601)
    xil_rescale(imageYCC_24, imageYCC_24, scale_values,
        offset_values);
if (ZOOM) {
    xil_ordered_dither(imageYCC_24, imageYCC_8, colorcube,
        dmask);
    xil_scale(imageYCC_8, zoom_displayimage, "nearest", 2.0,
        2.0);
}
else
    xil_ordered_dither(imageYCC_24, displayimage, colorcube,
        dmask);
/* MOLECULE ENDS HERE */
```

Exceptions to the Rules

Table 23-1 shows the exceptions to the general rules for executing this molecule. The exceptions have been classified on the basis of the decompressor to which they relate.

Table 23-1 Exceptions to the General Decompression-Molecule Rules

Decompressor	Exceptions
CellB	If you want the molecule to zoom the image it decompresses, the call to `xil_scale()` that zooms the image must precede the call to `xil_ordered_dither()`.
JPEG	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16.
	If the destination image is *not* a display image associated with a GX window, the molecule cannot perform the optional zoom.
	If the destination image is *not* a display image associated with a GX window, the destination cannot be a single-band child of a multiband parent image.
H.261	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16.
	There is one exception to the general rules that a molecule's destination image must have the same width and height as the images in the CIS and that the destination image must have an origin of 0.0, 0.0. The molecule can execute even though the two rules mentioned above are not met, if the following conditions are satisfied: (1) the destination image is a display image associated with a window on a local GX frame buffer, (2) the coordinates of the destination image's origin are less than or equal to 0.0, and (3) all of the pixels in the display image are written to. This exception allows for the clipping of letterboxed images.
MPEG-1	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16.

Twenty-Four-Bit Destination

The molecule decompresses a YC_bC_r image from a CIS and converts the color space of the YC_bC_r image to RGB.

Obviously, the images in a JPEG CIS must be YC_bC_r images before you can use this molecule. (CellB, H.261, and MPEG-1 CISs always contain YC_bC_r images.) In addition, the images in a JPEG CIS must be 4:1:1 interleaved in the bitstream. One other restriction is that JPEG, H.261, and MPEG-1 CISs must contain images whose width and height are multiples of 16.

For this molecule to be executed, your application must include the following code.

```
XilColorspace ycc, rgb;

ycc = xil_colorspace_get_by_name(state, "ycc601");
rgb = xil_colorspace_get_by_name(state, "rgb709");
xil_set_colorspace(imageYCC_24, ycc);
xil_set_colorspace(imageRGB_24, rgb);

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageYCC_24);
xil_color_convert(imageYCC_24, imageRGB_24);
/* MOLECULE ENDS HERE */
```

The molecule can only convert images from `ycc601` to `rgb709` (as shown in the example).

CCITT Group 4 Decompression Molecule

Because it is common to transpose an image immediately after decompressing it from a CCITT compressed image sequence, there is a molecule for accelerating the transposition of images that were compressed with the Group 4 compressor. The molecule takes advantage of the fact that, when decompressing these images, an intermediate representation of the original image is generated, and it is faster to transpose this representation than it is to transpose the bits of the actual image. For images that were compressed with the Group 3 compressor, this intermediate representation is not generated and so the molecule does not execute.

The `xil_transpose()` call must immediately follow the image decompression; otherwise the intermediate representation is lost and the molecule cannot execute. In this case, the transposition is performed on the bit image.

The type of flip designated on the `xil_transpose()` call determines whether the destination size is checked as

```
src_x_size == dest_x_size && src_y_size == dest_y_size
```

or

```
src_x_size == dest_y_size && src_y_size == dest_x_size
```

One of the above expressions must be true for the molecule to be called. Table 23-2 shows the destination image size expected for the source image size (100, 140).

Table 23-2 Type of Flip Designated on the Call to `xil_transpose()`

Type of Flip	Expected Destination Size
`XIL_FLIP_X_AXIS`	(100, 140)
`XIL_FLIP_Y_AXIS`	(100, 140)
`XIL_FLIP_X_180`	(100, 140)
`XIL_FLIP_MAIN_DIAGONAL`	(140, 100)
`XIL_FLIP_ANTIDIAGONAL`	(140, 100)
`XIL_FLIP_90`	(140, 100)
`XIL_FLIP_270`	(140, 100)

≡ *23*

SPARC: Molecules That Result in a Display

Molecules that combine a common operation and a display are available only on local GX frame-buffer screens. Before you look over the list of molecules in this category, you should understand that operations that read from or write to a device image (including a display image) are treated specially by the XIL library.

Generally, calling an XIL function results in the library's storing one operation. However, operations that read from or write to a device image cause *two* operations to be stored. For example, you might call `xil_scale()` using a scanner as your source image in order to read an image from the scanner and zoom it. The two operations stored are a *capture* operation, which writes its output to a temporary image, and a scale operation, which writes its output to the destination image named in the call to `xil_scale()`. Likewise, you might call `xil_scale()` using a display as your destination image in order to zoom the image and display it. In this case, the two operations stored are a scale and a *display* operation.

Several molecules are available that replace two-operation sequences in which the second operation is a display to a GX frame buffer. These molecules are:

- Add (`xil_add_const()`), pixel-by-pixel, the constant values of a source image to a GX display image

- Copy (`xil_copy()`) an 8-bit image to a GX display image

- Copy (`xil_copy()`) a GX display source image to a GX display destination image (in-place copy)

- Cast (`xil_cast()`) a 1-bit image to a GX display image

- Multiply (`xil_multiply_const()`), pixel-by-pixel, a source image by a GX display destination image

- Pass a 1-bit image through a lookup table (`xil_lookup()`) to a GX display image

- Pass an 8-bit image through a lookup table to produce a different result to a GX display image

- Rescale (`xil_rescale()`) an image to a GX display image

- Scale (`xil_scale()`) a GX display image about its origin with nearest-neighbor interpolation and a scale factor of 2.0 in both x and y

- Set all the values of a GX display image (`xil_set_value()`)

- Threshold (`xil_threshold()`) a 16-bit image (set value of bands within a specified range); threshold the 16-bit image; then cast (`xil_cast()`) the 16-bit image to a GX display image

- Translate (`xil_translate()`) an 8-bit image using nearest-neighbor interpolation to a GX display image

≡ *23*

Troubleshooting Molecules

Once you have coded your application with a replaceable sequence of atomic functions, you may want to make sure a molecule executes when your program runs. And if the molecule is not called, you should determine why it's not being called. This section discusses both how to determine whether a molecule is executing and what to check for if the molecule is not being called.

Determining Whether Molecules Are Executing

To determine whether a molecule you want to execute is executing, set the `XIL_DEBUG` environment variable before executing your program:

```
% setenv XIL_DEBUG show_action
```

Once you've done this, the XIL library prints a message to `stderr` each time an operation that affects the state of an XIL image or a compressed image sequence (CIS) is executed. By looking at these messages, you can determine whether the molecules your application is attempting to call are being executed.

For example, try setting `XIL_DEBUG` to `show_action` and then playing a JPEG movie using the `xilcis_example` example program. Instructions for running the example are presented in "Running the Movie Player" on page 328. As the program runs, you see one or more pairs of messages—depending on the length of the movie—displayed on `stderr`.

```
XIL_ACTION[XilDeviceCompJpegMemory]:ordereddither8to8(rescale8(
decompress_Jpeg()))
XIL_ACTION[XilDeviceIOioxlib]:display_ioxlib()
```

The first message indicates that a molecule has been executed. This molecule performs a decompression, a rescale, and an ordered dither. The second message indicates that this molecule was followed by an atomic display operation; this message assumes the display operation uses function calls from `ioxlib`.

If you were now to edit this program so that it did something that prevented the molecule from being executed—like setting the origin of the display image to 1.0, 1.0—you would see the following messages:

```
XIL_ACTION[XilDeviceCompJpegMemory]:ordereddither8to8(rescale8(
decompress_Jpeg()))
XIL_ACTION:FAILED
XIL_ACTION[XilDeviceCompJpegMemory]:decompress_Jpeg()
XIL_ACTION[XilDeviceComputeMemory]:rescale8()
XIL_ACTION[XilDeviceComputeMemory]:ordereddither8to8()
XIL_ACTION[XilDeviceIOioxlib]:display_ioxlib()
```

As you can see, the molecule would be unable to execute, and the decompression, rescale, and ordered dither would have to be executed individually.

Besides the `XIL_DEBUG` environment variable, there are two XIL functions that enable you to control whether the messages discussed above are printed. One of these, `xil_state_set_show_action()`, sets a system-state attribute called `SHOW_ACTION`. The setting of this attribute determines whether the library prints the messages, does not print the messages, or prints the messages only if `XIL_DEBUG` is set to `show_action`. The other function, `xil_state_get_show_action()`, reads the value of `SHOW_ACTION`. See the man pages for more information about these functions.

Determining Why a Molecule Is Not Executing

Basically, there are two reasons why a molecule might not execute. Either you didn't call the proper sequence of atomic functions with the proper arguments, or you did something that caused the library to flush some operations from storage before you finished building the molecule. These two situations are discussed in more detail in the next section.

Not Calling a Correct Sequence of Functions

The most likely reason a molecule is not executing is that you haven't followed all the rules for calling the molecule. Remember, you must call the correct functions, in the proper order, sometimes with specific arguments. Any mistake can cause the molecule not to be called or to fail.

For example, let's say you are decompressing a JPEG movie and displaying frames on an 8-bit frame buffer, and that you call the following code.

```
xil_decompress(cis, imageYCC);
xil_rescale(imageYCC, imageYCC, scale, offset);
xil_ordered_dither(imageYCC, image8, colorcube, dmask);
xil_scale(image8, zoom_displayimage, "bilinear", 2.0, 2.0);
```

If for no other reason, these four functions won't be replaced by a molecule because the call to `xil_scale()` uses an interpolation type of `bilinear`. For this molecule to execute, the interpolation type for the scale must be `nearest` (nearest neighbor). So, if you expect a molecule to execute and it doesn't, first check to make sure that you've met all the guidelines presented in "XIL Molecules" on page 477. Also see Appendix A, "Optimizations and Molecules For XIL Version 1.3.

A more subtle problem can arise when the calls that make up a molecule are not consecutive calls in your program. For instance, consider the following code fragment taken from the Cell movie playback code discussed in "Playing Cell Movies" on page 338.

```
xil_decompress(cis, image24);

/* Look at the Lookup version number to see if it is time
   to reinstall the colormap. */
if (lu_version != xil_lookup_get_version( xil_cmap )) {
    cell_install_cmap(x_cmap, displayimage, xil_cmap, ilist);
    lu_version = xil_lookup_get_version( xil_cmap );
}
xil_nearest_color(image24, displayimage, xil_cmap);
```

This code is attempting to call a molecule that replaces the `xil_decompress()-xil_nearest_color()` sequence.

Your first thought may be that this code won't cause the molecule to be executed because the calls to `xil_decompress()` and `xil_nearest_color()` are not consecutive. Between these two calls, the program calls `xil_lookup_get_version()` at least once and may call a user-defined function, `cell_install_cmap()`, which doesn't contain any XIL functions. Actually, this code causes the molecule to be called because only the decompress and nearest-color operations are deferred and become part of the stored chain of operations that the library tries to replace with a molecule.

Of course, Calls to non-XIL functions, like `cell_install_cmap()` are never deferred by the library. In addition, there is a class of XIL functions that are never deferred, and calls that read object attributes fall in this class. Therefore, the call to `xil_lookup_get_version()` does not affect the deferred-operation chain.

On the other hand, if you used the following code, the library would not execute a molecule.

```
xil_decompress(cis, image24);
xil_add_const(image24, constants, image24);
xil_nearest_color(image24, displayimage, xil_cmap);
```

The add-constant operation would be deferred (because it changes values in the image) and would prevent the library from finding an `xil_decompress()-xil_nearest_color()` sequence.

Flushing Operations Before a Molecule Is Complete

One other subtle problem to guard against is calling a function that causes the library to execute atomically functions that you intended to be the beginning of a molecule. For example, let's say that you want to execute a molecule that replaces four API-level functions. You call the first three atomic functions, and information about these deferred operations is stored. Then before you call the fourth function, you do something that causes the first three operations to be flushed from storage and executed.

What could you do to cause this to happen? There are a number of things, some obvious and some not so obvious. At the obvious end of the spectrum, you could:

- Turn deferred execution off using the `xil_state_set_synchronize()` function. Once deferred execution is turned off, of course, no operations are deferred.

- Call `xil_sync()` on an intermediate image. This call causes the library to produce that image immediately, which involves flushing all the stored operations needed to produce it.

- Use an intermediate image whose synchronization flag has been set using the `xil_set_synchronize()` function. The operations required to produce this image cannot be deferred.

You can also flush operations that you meant to be part of a molecule by:

- Calling `xil_export()` to export an intermediate image. The operations needed to produce the intermediate image must be executed before the image can be exported.

- Calling a function that returns information about the values in an intermediate image—like `xil_extrema()`, `xil_histogram()`, or `xil_get_pixel()`. Such a call causes the operations that will produce the source image to be executed.

Side Effects of Executing Molecules

When a molecule executes, it may not create one or more temporary images that would have been created had the set of atomic functions replaced by the molecule been executed. This section briefly explains the possible side effects of this behavior and how you can deal with them.

As an example, consider the following JPEG decompression code.

```
XilImage imageYCC, displayimage;

xil_decompress(cis, imageYCC);
xil_rescale(imageYCC, imageYCC, scale, offset);
xil_ordered_dither(imageYCC, displayimage, colorcube, dmask);
```

If these functions are executed atomically, decompressed data is written to the 24-bit image `imageYCC`, the data in `imageYCC` is rescaled, and then the data in `imageYCC` is dithered to produce an 8-bit image. However, if a JPEG decompression molecule is executed, `imageYCC` is never actually used. This difference can lead to unexpected results later on.

Suppose the decompression molecule executed, and `imageYCC` was never written to. If you later ask the library to destroy the CIS (without having first destroyed `imageYCC`), before it destroys the CIS, the library calls `xil_decompress()` to decompress data from the CIS and write it to `imageYCC`. The library does this because it knows that it never wrote data to `imageYCC`, that the contents of `imageYCC` depend on the CIS, and that the image may be used again later in the program. If you don't plan to use the image later in your program, the call to `xil_decompress()` is unnecessary and hurts the performance of your program. Another option is to use temporary images for intermediate images in the chain of operation. (For details on temporary images, see "Temporary Images" on page 61.)

You can check for unexpected side effects like the one described above by setting `XIL_DEBUG` to `show_action` and monitoring `stderr` to see which functions execute when you run your program.

XIL Functions That Relate to Deferred Execution

Several XIL functions have a direct effect on the way deferred execution works. These are listed in Table 23-3.

Table 23-3 Functions That Affect Deferred Execution

Function	Description
`xil_sync()`	Forces the library to compute the value of an image when the operations that produce the image have been deferred.
`xil_cis_sync()`	Forces any outstanding call to `xil_compress()` to complete when it would otherwise have been deferred.
`xil_set_synchronize()`	Sets an image's synchronization attribute. When an image has this attribute set, operations on it cannot be deferred.

Table 23-3 Functions That Affect Deferred Execution

Function	Description
`xil_get_synchronize()`	Determines whether an image's synchronization attribute is set.
`xil_state_set_synchronize()`	Turns deferred execution on or off.
`xil_state_get_synchronize()`	Determines whether deferred execution is being used or not.
`xil_toss()`	Tells the library to "toss," or throw away, the contents of an image, but not to destroy the image.

Optimizations and Molecules For XIL Version 1.3 A≡

This appendix describes the optimizations that are available in the 1.3 release of the XIL imaging library. Optimizations are presented in the following categories:

- Functions
- Xlib display
- X Shared Memory Extension display
- Molecules
- GX display molecules

Functions

Affine

`xil_affine()` is optimized to:

- Call `xil_rotate()` if only rotation is specified

- Call `xil_scale()` if only scaling is specified

- Call `xil_translate()` if only translation is specified

- Call `xil_transpose()` if origins for source and destination are set to the center of the image and transposition is specified

Arithmetic, Relational, and Logical

These functions are specially optimized for packed 1-band and 3-band images.

Blend

`xil_blend()` is optimized for the following data:

- 1-band `BYTE` with `BYTE` Alpha
- 3-band `BYTE` with `BYTE` Alpha

Cast

`xil_cast()` is optimized when casting images from:

- 1-band `BYTE` -> `BIT`
- n-band `BIT` -> `BYTE`
- n-band `BIT` -> `SHORT`
- n-band `BIT` -> `FLOAT`

Convolve

`xil_convolve()` is optimized under the following circumstances:

- If separable kernel, the separable kernel function is called, which does less work.

- If a 3x3 kernel with a key at 1,1 for `BYTE` or `SHORT` data

- If square kernels for 1-band `BYTE` or `SHORT` data

Error Diffusion

`xil_error_diffusion()` is optimized for:

- Color cube
- Floyd-steinberg 8 bit -> 1 bit
- Floyd-steinberg 24 bit -> 8 bit (3-band `BYTE` -> 1-band `BYTE`)

Lookup

`xil_lookup()` is optimized for the following source and destination image data types:

- 1-band BYTE -> 1-band BYTE
- 1-band BYTE -> 1-band BYTE
- 1-band BYTE -> 3-band BYTE
- 3-band BYTE -> 3-band BYTE

Nearest Color

`xil_nearest_color()` is optimized for 3-band BYTE -> 1-band BYTE.

Ordered Dither

`xil_ordered_dither()` is optimized for 24-bit -> 8-bit
(3-band BYTE -> 1-band BYTE).

Paint

`xil_paint()` is optimized for 3-band BYTE image data.

Rotate

`xil_rotate()` is optimized for 90-, 180-, and 270-degree rotations and the origin is the center of the image.

Scale

`xil_scale()` is optimized for the following interpolation types:

- Nearest neighbor (2.0 in both x and y)
- Bilinear (0.5 or 2.0 in both x and y)

≡ *A*

Subsample Binary-to-Gray

`xil_subsample_binary_to_gray()` is optimized for the following x and y scale factors:

- 0.25, 0.25 (1/4 in x and y)
- 0.33333, 0.33333 (1/3 in x and y)
- 0.5, 0.5 (1/2 in x and y)

Threshold

`xil_threshold()` is optimized as follows:

- Two thresholds with the same intermediate image are done as a combined threshold
- 1-band `BYTE` is specially optimized

Xlib Display

Xlib displays are optimized as follows:

- `xil_set_value()` is done as `XFillRectangle()`

- `xil_cast()` of a 1-bit image to 8-bit display uses the X server to convert from 1-bit to 8-bit (only 1-bit data are sent)

X Shared Memory Extension Display

The X Shared Memory Extension is optimized as follows:

- When no device-specific acceleration is available, XIL uses the X Shared Memory Extension until there is no more system shared memory to display images

- The `xil_set_value()` and `xil_cast()` functions are optimized in the same way as Xlib displays (see "Xlib Display")

Molecules

This section summarizes the molecules that are available with XIL 1.3 and lists the available optimizations.

Cell Decompression

```
xil_decompress() -> xil_nearest_color()
```

```
xil_decompress() -> xil_ordered_dither()
```

Optimizations:

- 4x4 dither mask
- 1-band with pixel stride of 1 for destination
- 4-byte alignment

```
xil_decompress() ->[xil_scale()] -> xil_ordered_dither()
```

Optimizations:

- 4x4 dither mask
- 1-band with pixel stride of 1 for destination
- 4-byte alignment

Note – `xil_scale()` only supports 2.0 scale factor in x and y.

CellB Decompression

```
xil_decompress() -> xil_color_convert()
```

- Color conversion must be ycc601 -> rgb709

```
xil_decompress() -> xil_ordered_dither()
```

```
xil_decompress() -> xil_rescale() -> xil_ordered_dither()
```

FaxG4 Decompression

```
xil_decompress() -> xil_transpose()
```

- Supports FLIP 90, 180, 270

JPEG Baseline Sequential, MPEG-1, H.261 Decompression

```
xil_decompress() -> xil_color_convert()
```

- Color conversion must be ycc601 -> rgb709

≡ *A*

xil_decompress() -> [xil_rescale()] -> xil_ordered_dither

- 4x4 dither mask
- 855 color cube only
- Image dimensions are multiple of 16

GX Display Molecules

These optimizations are performed as a single operation into the frame buffer when the display image on a GX device is used as the destination. Often, the frame buffer is used to accelerate the operations, providing greater speed.

- `xil_copy()`

- `xil_set_value()`

- `xil_add_const()`

- `xil_cast()` -> BIT -> BYTE

- `xil_copy()` with a GX display image used as source and destination (in-place copy)

- `xil_lookup()` 1-band BIT -> 1-band BYTE

- `xil_lookup()` 1-band BYTE -> 1-band BYTE

- `xil_multiply_const()`

- `xil_rescale()`

- `xil_scale()` with nearest interpolation and a scale factor equal to 2.0 in both x and y

- `xil_translate()`

- `xil_threshold()` for SHORT ; `xil_threshold()` for SHORT; `xil_cast()` from SHORT -> BYTE (medical imaging "window/level"operation)

These operations are performed into a GX frame buffer with a GX display image as the destination.

XIL Error Messages *B* ≡

Error Messages

This appendix contains a listing of the error messages that can be generated by the XIL library. Breaks in the sequential numbering of the errors do not indicate missing errors. These are merely gaps that occurred as the numbers were assigned.

Note – Your application may receive errors from functions not listed in this appendix. The function list is not complete at this writing.

$\equiv B$

Table B-1 XIL Library Error Messages

Error Number	Error Message	Function That Generates Error
di -1	Out Of Memory	All functions.
di-2	Image source1 bands or type does not match destination	`xil_extrema()`, `xil_fill()`, `xil_add()`, `xil_add_const()`, `xil_and()`, `xil_and_const()`, `xil_threshold()`, `xil_black_generation()`, `xil_convolve()`, `xil_copy()`, `xil_dilate()`, `xil_divide()`, `xil_divide_by_const()`, `xil_divide_into_const()`, `xil_erode()`, `xil_histogram()`, `xil_multiply()`, `xil_multiply_const()`, `xil_not()`, `xil_or()`, `xil_or_const()`, `xil_ordered_dither()`, `xil_paint()`, `xil_rescale()`, `xil_set_value()`, `xil_soft_fill()`, `xil_squeeze_range()`, `xil_subsample_adaptive()`, `xil_subtract()`, `xil_subtract_const()`, `xil_subtract_from_const()`, `xil_transpose()`, `xil_xor()`, `xil_cis_get_output_type()`, `xil_decompress()`, `xil_absolute()`, `xil_edge_detection()`, `xil_max()`, `xil_min()`, `xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-3	Image source 2 does not match destination	`xil_extrema(), xil_fill(), xil_add(), xil_add_const(), xil_and(), xil_and_const(), xil_threshold(), xil_black_generation(), xil_convolve(), xil_copy(), xil_dilate(), xil_divide(), xil_divide_by_const(), xil_divide_into_const(), xil_erode(), xil_histogram(), xil_multiply(), xil_multiply_const(), xil_not(), xil_or(), xil_or_const(), xil_ordered_dither(), xil_paint(), xil_rescale(), xil_set_value(), xil_soft_fill(), xil_squeeze_range(), xil_subsample_adaptive(), xil_subtract(), xil_subtract_const(), xil_subtract_from_const(), xil_transpose(), xil_xor(), xil_cis_get_output_type(), xil_decompress(), xil_absolute(), xil_edge_detection(), xil_max(), xil_min(), xil_tablewarp(), xil_tablewarp_horizontal(), xil_tablewarp_vertical()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-4	Image source 3 does not match destination	`xil_extrema()`, `xil_fill()`, `xil_add()`, `xil_add_const()`, `xil_and()`, `xil_and_const()`, `xil_threshold()`, `xil_black_generation()`, `xil_convolve()`, `xil_copy()`, `xil_dilate()`, `xil_divide()`, `xil_divide_by_const()`, `xil_divide_into_const()`, `xil_erode()`, `xil_histogram()`, `xil_multiply()`, `xil_multiply_const()`, `xil_not()`, `xil_or()`, `xil_or_const()`, `xil_ordered_dither()`, `xil_paint()`, `xil_rescale()`, `xil_set_value()`, `xil_soft_fill()`, `xil_squeeze_range()`, `xil_subsample_adaptive()`, `xil_subtract()`, `xil_subtract_const()`, `xil_subtract_from_const()`, `xil_transpose()`, `xil_xor()`, `xil_cis_get_output_type()`, `xil_decompress()`
di-5	Operation not implemented	All functions.
di-6	Could not get ROI of an image	A secondary error caused by an internal error. Could occur with any function.
di-7	Could not intersect ROIs	A secondary error caused by an internal error. Could occur with any function.
di-8	Could not get region list from ROI	A secondary error caused by an internal error. Could occur with any function.

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-9	Could not get src1 as memory	`xil_add()`, `xil_add_const()`, `xil_affine()`, `xil_and()`, `xil_and_const()`, `xil_black_generation()`, `xil_blend()`, `xil_cast()`, `xil_convolve()`, `xil_copy()`, `xil_copy_pattern()`, `xil_dilate()`, `xil_divide()`, `xil_divide_by_const()`, `xil_divide_into_const()`, `xil_erode()`, `xil_error_diffusion()`, `xil_extrema()`, `xil_fill()`, `xil_rotate()`, `xil_scale()`, `xil_translate()`, `xil_transpose()`, `xil_histogram()`, `xil_lookup()`, `xil_multiply()`, `xil_multiply_const()` `xil_nearest_color()`, , `xil_not()`, `xil_or()`, `xil_or_const()`, `xil_ordered_dither()`, `xil_paint()`, `xil_rescale()`, `xil_soft_fill()`, `xil_subsample_adaptive()`, `xil_subsample_binary_to_gray()`, `xil_subtract()`, `xil_subtract_const()`, `xil_subtract_from_const()`, `xil_threshold()`, `xil_xor()`, `xil_edge_detection()`, `xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`

Table B-1   XIL Library Error Messages   *(Continued)*

Error Number	Error Message	Function That Generates Error
di-10	Couldn't get dst1 as memory	`xil_add()`, `xil_add_const()`, `xil_affine()`, `xil_and()`, `xil_and_const()`, `xil_black_generation()`, `xil_blend()`, `xil_cast()`, `xil_choose_colormap()`, `xil_color_convert()`, `xil_convolve()`, `xil_copy()`, `xil_copy_pattern()`, `xil_dilate()`, `xil_divide()`, `xil_divide_by_const()`, `xil_divide_into_const()`, `xil_erode()`, `xil_error_diffusion()`, `xil_extrema()`, `xil_fill()`, `xil_rotate()`, `xil_scale()`, `xil_translate()`, `xil_transpose()`, `xil_lookup()`, `xil_multiply()`, `xil_multiply_const()`, `xil_nearest_color()`, `xil_not()`, `xil_or()`, `xil_or_const()`, `xil_ordered_dither()`, `xil_paint()`, `xil_rescale()`, `xil_soft_fill()`, `xil_subsample_adaptive()`, `xil_subsample_binary_to_gray()`, `xil_subtract()`, `xil_subtract_const()`, `xil_subtract_from_const()`, `xil_threshold()`, `xil_xor()`, `xil_absolute()`, `xil_copy_with_planemask()`, `xil_max()`, `xil_min()`, `xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`
di-11	Could not get alpha as memory	`xil_blend()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-12	Could not get combined ROI	`xil_add(),` `xil_add_const(),xil_and(),` `xil_and_const(),` `xil_black_generation(),` `xil_blend(), xil_cast(),` `xil_color_convert(), xil_copy(),` `xil_dilate(), xil_divide(),` `xil_divide_by_const(),` `xil_divide_into_const(),` `xil_erode(),` `xil_error_diffusion(),` `xil_lookup(), xil_multiply(),` `xil_multiply_const(),` `xil_nearest_color(), xil_not(),` `xil_or(), xil_or_const(),` `xil_ordered_dither(),` `xil_paint(), xil_rescale(),` `xil_subtract(),` `xil_subtract_const(),` `xil_subtract_from_const(),` `xil_threshold(), xil_xor(),` `xil_absolute(),` `xil_copy_with_planemask(),` `xil_max(), xil_min()`
di-13	Dependent count overflow	`xil_compress(), xil_decompress()`
di-14	Could not copy ROI	A secondary error caused by memory resource limitations. Could occur with any function.
di-15	Could not get src2 as memory	A secondary error caused by an internal error. Could occur with any function.
di-16	ROI translation failed	`xil_copy_pattern()`. Secondary error probably due to resource problems.
di-17	Attempted to get an unknown attribute	`xil_cis_get_attribute()`
di-18	Attempted to get a set-only attribute	`xil_cis_get_attribute()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-19	Attempted to set an unknown attribute	`xil_cis_set_attribute()`
di-20	Attempted to set a get-only attribute	`xil_cis_set_attribute()`
di-21	Cannot create op	A secondary error probably due to resource problems. Could occur with any function.
di-22	Image type mismatch	Any function
di-23	Unable to perform the specified function	Should not happen unless an error occurred during the operation
di-24	Could not get the ROI list	`xil_transpose()`
di-55	XilCis: JPEG attribute error: cannot have interleaved bands with different selector(s) and/or pt_transform values	Reported by the JPEG lossless compressor
di-56	XilCis: JPEG attribute error: `DECOMPRESSION_QUALITY` value must be >= 1 and <= 100	`xil_cis_set_attribute()`
di-57	XilCis: JPEG attribute error: `ENCODE_INTERLEAVED` value must be either TRUE or FALSE	`xil_cis_set_attribute()`
di-58	XilCis: JPEG attribute error: `ENCODE_VIDEO` value must be either TRUE or FALSE	`xil_cis_set_attribute()`
di-59	XilCis: JPEG attribute error: `COMPRESSION_QUALITY` value must be >=1 and <=100	`xil_cis_set_attribute()`
di-60	XilCis: JPEG attribute error: `TEMPORAL_FILTER` value must be either TRUE or FALSE	`xil_cis_set_attribute()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-61	XilCis: JPEG attribute error: `OPTIMIZE_HUFFMAN_TABLES` value must be either TRUE or FALSE	`xil_cis_set_attribute()`
di-62	XilCis: JPEG attribute error: `COMPRESSED_DATA_FORMAT` value must be either INTERCHANGE or ABBREVIATED_FORMAT	`xil_cis_set_attribute()`
di-63	XilCis: Cell attribute error: Must set `DECOMPRESSOR_COLORMAP` before `RDWR_INDICES`	`xil_cis_set_attribute()` (for Cell)
di-64	XilCis: Cell attribute error: `RDWR_INDICES`: Invalid NULL index list	`xil_cis_set_attribute()` (for Cell)
di-65	XilCis: Cell attribute error: `RDWR_INDICES`: Invalid index list count	`xil_cis_set_attribute()` (for Cell)
di-66	XilCis: ROIs not allowed for compress image	Reported by JPEG lossless compressor
di-67	XilCis: JPEG bytestream error: Invalid code value index	`xil_decompress()`. This error can also be generated by `xil_cis_has_frame()`, `xil_cis_number_of_frames()`, and `xil_cis_seek()` if user inserts data into CIS with `xil_cis_put_bits_[ptr]()` with an unknown or partial number of frames.
di-68	XilCis: JPEG bytestream error: Invalid symbol table ID	`xil_decompress()`. This error can also be generated by `xil_cis_has_frame()`, `xil_cis_number_of_frames()`, and `xil_cis_seek()` if user inserts data into CIS with `xil_cis_put_bits_[ptr]()` with an unknown or partial number of frames.

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-69	XilCis: JPEG bytestream error: Invalid Huffman code length	`xil_decompress()`. This error can also be generated by `xil_cis_has_frame()`, `xil_cis_number_of_frames()`, and `xil_cis_seek()` if user inserts data into CIS with `xil_cis_put_bits_[ptr]()` with an unknown or partial number of frames.
di-70	XilCis: JPEG bitstream error: Invalid value for prediction selector	`xil_cis_set_attribute()`
di-71	XilCis: JPEG bitstream error: Invalid value for point transform	`xil_cis_set_attribute()`
di-72	XilCis: JPEG bitstream error: Invalid SOF marker for this decompressor	`xil_decompress()`
di-73	XilCis: JPEG bitstream error: Invalid selector in SOF segment	`xil_decompress()`
di-74	XilCis: JPEG bitstream error: Unsupported image precision in SOF segment	`xil_decompress()`
di-75	XilCis: JPEG bitstream error: Invalid image height in SOF segment	`xil_decompress()`
di-76	XilCis: JPEG bitstream error: Invalid image width in SOF segment	`xil_decompress()`
di-77	XilCis: JPEG bitstream error: Invalid qtable marker 0	`xil_compress()`
di-78	XilCis: JPEG bitstream error: Table not used by any bands	`xil_compress()`
di-79	XilCis: JPEG bitstream error: Invalid quantizer precision identifier	`xil_cis_create()`, `xil_compress()`, `xil_decompress()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-80	XilCis: JPEG bitstream error: Number of quantizer tables too large	`xil_cis_create()`
di-81	XilCis: JPEG bitstream error: Table identifier not in use	`xil_decompress()`
di-82	XilCis: JPEG bitstream error: Invalid table identifier	`xil_decompress()`
di-83	XilCis: JPEG bitstream error: Invalid ac table identifier	`xil_decompress()`
di-84	XilCis: JPEG bitstream error: dc table identifier not in use	`xil_compress()`
di-85	XilCis: JPEG bitstream error: Invalid dc table identifier	`xil_compress()`
di-86	XilCis: JPEG bitstream error: Invalid component identifier	`xil_decompress()`
di-87	XilCis: JPEG bitstream error: Attempted use of non-loaded ac table	`xil_decompress()`
di-88	XilCis: JPEG bitstream error: Attempted use of non-loaded dc table	`xil_compress()`
di-89	XilCis: JPEG bitstream error: Attempted use of non-loaded qtable	`xil_compress()`
di-90	XilCis: JPEG bitstream error: Invalid htable identifier	`xil_decompress()`
di-91	XilCis: JPEG bitstream error: Invalid qtable identifier	`xil_cis_create(), xil_compress()`
di-92	XilCis: JPEG bitstream error: Unknown htable type	`xil_cis_set_attribute()`
di-93	XilCis: JPEG bitstream error: Invalid band number	`xil_cis_get_attribute()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-94	XilCis: JPEG bitstream error: Table index too large	`xil_decompress()`
di-95	XilCis: Internal error	`xil_cis_create()`, `xil_cis_get_output_type()`, `xil_cis_get_attribute()`, `xil_compress()`, `xil_decompress()`, `xil_cis_has_data()`, `xil_nearest_color()`, `xil_cis_destroy()`
di-96	XilCis: Wrong number of ops in molecule	`xil_nearest_color()`
di-97	XilCis: Cell bitstream error	`xil_nearest_color()`
di-98	XilCis: JPEG bitstream error	`xil_decompress()`
di-100	XilCis: No data to decompress	`xil_cis_get_output_type()`
di-101	XilCis: Could not create CIS destination image	`xil_decompress()`. A secondary error that occurs when the compressor is unable to allocate a temporary image to compress into.
di-102	XilCis: JPEG cannot encode image as 4:1:1 yuv, using `ENCODE_INTERLEAVED` attribute instead	`xil_compress()`
di-103	XilCis: JPEG cannot interleave image, it has 5 or more bands	`xil_compress()`
di-104	XilCis: Wrote more than maximum frame size into buffer space	`xil_compress()`. Internal error. This should never happen.
di-105	XilCis: Seek to frame number not in CIS	`xil_cis_seek()`
di-106	XilCis: Desired frame number no longer in CIS	`xil_cis_seek()`

Table B-1   XIL Library Error Messages   *(Continued)*

Error Number	Error Message	Function That Generates Error
di-107	XilCis: CIS is empty, cannot seek to frame number	`xil_cis_seek()`
di-108	XilCis: No previous desired frame type to seek backward to	`xil_cis_seek()`. Internal error.
di-109	XilCis: Unable to complete seek by skipping (burning) frames	`xil_nearest_color()`
di-110	XilCis: Complete frame does not exist. No data to decompress	`xil_decompress()`
di-111	XilCis: Image width and height must be a multiple of 4	`xil_cis_get_attribute()`, `xil_decompress()`, `xil_cis_get_output_type()`, `xil_cis_has_data()`
di-112	XilCis: Could not create XilLookup for colormap	`xil_decompress()` (for Cell)
di-113	XilCis: Fax: Source not a bit image	`xil_compress()`
di-114	XilCis: Incomplete frame in buffer, cannot copy until complete	`xil_compress()`, `xil_cis_put_bits()`
di-115	Xilcis: `start_frame` adjusted past `read_frame`. CIS data lost.	`xil_cis_seek()`, `xil_compress()`
di-116	XilCis: Unsupported or illegal precision for compressed image	`xil_decompress()`. Internal error.
di-117	XilCis: Invalid NULL CIS specified	`xil_cis*() (ny xil_cis* routine)`
di-118	XilCis: Partial frame end adjustment error on frame position	`xil_compress()`, `xil_cis_put_bits()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-119	XilCis: Partial frame end adjustment error on frame list	`xil_compress()`, `xil_cis_put_bits()`
di-120	XilCis: Cannot create copy op	`xil_compress()`, `xil_decompress()`
di-121	XilCis: Invalid compression type or specification	`xil_compress()`, `xil_decompress()`
di-122	XilCis: Can't create compression op	`xil_decompress()`
di-123	XilCis: Decompress destination image's type or nbands does not match CIS	`xil_decompress()`
di-124	XilCis: Illegal `relative_to` value in `xil_cis_seek()`	`xil_cis_seek()`
di-125	XilCis: Compression device is unavailable	`xil_cis_create()`
di-126	XilCis: Couldn't create compression device	`xil_cis_create()`
di-127	Lookup would be too large for data type	`xil_lookup_create()`, `xil_colorcube_create()`
di-128	Colorcube multipliers do not match sizes	`xil_colorcube_create()`
di-129	Object would be too large for 32-bit addressability	`xil_create()`, `xil_create_from_type()`, `xil_create_temporaty()`, `xil_kernel_create()`, `xil_dithermask_create()`, `xil_sel_create()`, `xil_lookup_create()`, `xil_colorcube_create()`, `xil_histogram_create()`
di-130	Attempted to read or write outside the range of a lookup	`xil_lookup_get_values()`, `xil_lookup_set_values()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-131	Invalid NULL lookup	`xil_ordered_dither()`, `xil_error_diffusion()`, `xil_nearest_color()`, `xil_lookup_convert()`, `xil_choose_colormap()`, `xil_lookup()`, `xil_lookup_destroy()`, `xil_lookup_get_input_datatype()`, `xil_lookup_get_output_datatype()`, `xil_lookup_get_output_nbands()`, `xil_lookup_get_num_entries()`, `xil_lookup_get_version()`, `xil_lookup_get_offset()`, `xil_lookup_set_offset()`, `xil_lookup_get_colorcube()`, `xil_lookup_get_colorcube_info()`, `xil_lookup_get_values()`, `xil_lookup_set_values()`, `xil_lookup_create_copy()`
di-132	Look-up type mismatch	`xil_lookup_convert()`
di-133	Lookup may not have input type of `XIL_FLOAT`	`xil_lookup_create()`, `xil_colorcube_create()`
di-134	Incorrect data type for image	`xil_create()` `xil_create_temporary()`
di-135	Attempted to set attribute on a non-device image	`xil_set_device_attribute()`
di-136	Attempted to get attribute of a non-device image	`xil_get_device_attribute()`
di-137	Requested pixel outside range of image	`xil_set_pixel()`, `xil_get_pixel()`
di-138	Could not get image as memory	`xil_get_memory_storage()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-139	Could not propagate image storage	`xil_get_memory_storage()`, `xil_set_memory_storage()`. This is probably the result of another error, such as running out of memory.
di-140	Could not get image storage	Internal system error
di-141	Could not create storage device	`xil_get_pixel()`, `xil_set_pixel()`, `xil_get_memory_storage()`
di-142	Cannot get memory storage on non-exported image	`xil_get_memory_storage()`
di-143	Could not access storage device	Internal system error
di-144	Could not copy colormap	`xil_ordered_dither()`
di-145	Could not create ROI	A secondary error caused by an internal error. Could occur with any function.
di-146	Invalid parameters passed to function	`xil_add()`, `xil_create()`, `xil_create_child()`, `xil_create_copy()`, `xil_set_origin()`, `xil_histogram_create()`, `xil_kernel_create()`, `xil_lookup_create()`, `xil_sel_create()`
di-147	Could not create image	Secondary error in `xil_create()` `xil_create_child()`, `xil_create_copy()` `xil_create_temporary()`
di-148	Child count exceeded on parent—could not create child image	`xil_create_child()`
di-149	Could not create input/output device	Internal system error

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-150	Could not access input/output device	Internal system error
di-151	Image already exported	`xil_export()`
di-152	Image is of a type that cannot be exported	`xil_export()`
di-153	Cannot import image that was not previously exported	`xil_import()`
di-154	Image must be exported to set memory storage	`xil_set_memory_storage()`
di-155	Cannot set memory storage on a child image	`xil_set_memory_storage()`
di-156	Image band mismatch	`xil_cast(), xil_copy(), xil_affine(), xil_blend(), xil_copy_pattern(), xil_rotate(), xil_scale(), xil_subsample_binary_to_gray(), xil_translate(), xil_absolute(), xil_edge_detection(), xil_max(), xil_min()`
di-157	Undefined image data type encountered	Internal error
di-158	Could not load `.so` file	`xil_open(), xil_create_from_device()`
di-159	Could not extract symbols from `.so` file	`xil_open()`
di-160	Could not expand path	`xil_open()`
di-161	Could not open `/lib/ xil.compute` file	`xil_open()`
di-162	XIL Xlib I/O Driver: Could not get graphics context	`xil_create_from_window()`
di-163	Could not copy kernel	`xil_convolve(), xil_kernel_create(), xil_error_diffusion()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-164	Input and output must be in CMYK color space	`xil_black_generation()`
di-165	Alpha channel image must have only 1 band	`xil_blend()`
di-166	Invalid size for lookup	`xil_choose_colormap()`, `xil_lookup_create()`, `xil_error_diffusion()`, `xil_nearest_color()`
di-167	Image must have 3 bands	`xil_choose_colormap()`
di-169	SEL must be specified	`xil_dilate()`, `xil_erode()`
di-170	Could not copy SEL	`xil_dilate()`, `xil_erode()`, `xil_sel_create_copy()`
di-171	Coefficient(s) would cause a divide by zero	`xil_divide()`
di-172	Supplied coefficient(s) contained NaN	`xil_divide()`, `xil_multiply()`, `xil_subtract()`
di-173	Number of bands in src image does not match look-up table	`xil_error_diffusion()`, `xil_nearest_color()`, `xil_ordered_dither()`
di-174	Data type of dest image does not match look-up table input type	`xil_error_diffusion()`, `xil_nearest_color()`
di-175	Data type of src image does not match look-up table output type	`xil_error_diffusion()`, `xil_nearest_color()`, `xil_ordered_dither()`
di-176	Output image must be single banded	`xil_error_diffusion()`, `xil_nearest_color()`, `xil_ordered_dither()`
di-177	Could not copy lookup	`xil_error_diffusion()`, `xil_nearest_color()`, `xil_lookup_create_copy()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-179	Data type of src image does not match look-up table input type	`xil_lookup()`
di-180	Data type of dest image does not match look-up table output type	`xil_lookup()`
di-181	Input image must be single banded	`xil_lookup()`
di-182	Brush must be specified	`xil_paint()`
di-183	Could not copy brush	`xil_paint()`
di-184	XilCis: Error AdjustStart	Internal error
di-185	Attempt to insert at illegal position in linked list	Internal error
di-186	Attempt to delete at illegal position in linked list	Internal error
di-187	NULL ROI passed to intersect routine	`xil_roi_intersect()`
di-188	Could not create image type	`xil_cis_get_output_type()`, `xil_cis_get_input_type()`, `xil_create()`, `xil_create_child()`, `xil_create_copy()`, `xil_create_from_device()`, `xil_create_from_type()`, `xil_create_from_window()`
di-189	Could not create kernel	`xil_kernel_create()`
di-190	Could not create dithermask	`xil_dithermask_create()`
di-191	Could not create SEL	`xil_sel_create()`
di-192	Could not create lookup	`xil_lookup_create()`
di-193	Could not branch to op	Internal error
di-194	No capture operation defined for this device	`xil_copy()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-195	No display operation defined for this device	`xil_copy()`
di-197	Internal error in getting op parameters	Internal error
di-198	Internal error in setting op parameters	Internal error
di-199	Internal error searching for optimized execution path	Internal error
di-202	Source color space not specified	`xil_color_convert()`
di-203	Destination color space not specified	`xil_color_convert()`
di-204	Unsupported color conversion	`xil_color_convert()`
di-205	Could not get colorcube information	Internal error
di-206	Lookup must be a colorcube	`xil_ordered_dither()`
di-207	Invalid NULL image specified	All functions
di-208	Tried to remove an error handler that was not installed	`xil_remove_error_handler()`
di-209	Cannot get system state from image	Internal error
di-210	Cannot create lookup for returning data	`xil_squeeze_range()`
di-211	Cannot squeeze range of multiband image	`xil_squeeze_range()`
di-212	XIL I/O Driver: Could not open specified frame buffer	`xil_create_from_device()`
di-213	XIL I/O Driver: Could not perform ioctl on frame buffer	`xil_create_from_device()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-214	XIL I/O Driver: Frame buffer not of expected type	`xil_create_from_device()`
di-215	XIL I/O Driver: Could not mmap device	`xil_create_from_device()`
di-216	Could not get FLOAT image as memory	Internal error
di-217	XIL Memory Driver: Could not get named storage type	A secondary error that can be seen after any call that creates or copies images or CISs
di-218	XIL Memory Driver: Could not create named storage device	A secondary error that can be seen after any call that creates or copies images or CISs
di-219	XIL I/O Driver: Could not connect to dga	`xil_create_from_device()`
di-220	XIL I/O Driver: Could not get window geometry	`xil_create_from_device()`
di-221	Invalid NULL kernel specified	`xil_convolve()`, `xil_error_diffusion()`, `xil_kernel_get_width()`, `xil_kernel_get_height()`, `xil_kernel_get_key_x()`, `xil_kernel_get_key_y()`, `xil_kernel_create_copy()`
di-222	Invalid NULL dither mask specified	`xil_dithermask_get_width()`, `xil_dithermask_get_height()`, `xil_dithermask_get_nbands()`, `xil_dithermask_create_copy()`, `xil_error_diffusion()`, `xil_ordered_dither()`
di-223	Tried to write to an invalid CIS	`xil_compress()`
di-224	Tried to read from an invalid CIS	`xil_decompress()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-230	XilCis: Cell attribute failed because too few bytes per frame group	`xil_cis_set_attribute()` `(BITS_PER_SECOND)`
di-231	XilCis: Cell key frame interval is too large for bit rate control	`xil_cis_set_attribute()` `(KEYFRAME_INTERVAL)`
di-232	XilCis: Cell key frame interval exceeds maximum	`xil_cis_set_attribute()` `(KEYFRAME_INTERVAL)`
di-233	Must specify histogram	`xil_histogram()`
di-234	Number of bands in image and histogram do not match	`xil_histogram()`
di-235	Zero bins in histogram not allowed	`xil_create_histogram()`
di-236	Cannot create histogram	`xil_create_histogram()`
di-237	Could not initialize compression type	`xil_cis_create()`
di-238	Could not initialize input/output type	`xil_create_from_device()`
di-239	Could not set image attribute	`xil_set_attribute()`. A secondary error.
di-240	Error opening or parsing `/lib/xil.modules`	`xil_open()`
di-241	Error loading Xlib display driver	`xil_create_from_window()`
di-242	Couldn't create display image	`xil_create_from_window()`. A secondary error.
di-243	Number of bands cannot be 0	`xil_lookup_create()`
di-244	Cannot create system state	`xil_open()`. A secondary error.
di-245	Invalid dither mask dimensions	`xil_dithermask_create()`
di-246	Cannot create global state	`xil_open()`. A secondary error.

Table B-1   XIL Library Error Messages   *(Continued)*

Error Number	Error Message	Function That Generates Error
di-247	Cannot initialize localization information	`xil_open()`
di-248	Unable to load any compute device handlers	`xil_open()`. Probably a secondary error due to improper configuration.
di-249	XIL Xlib I/O Driver: Could not get window attributes	`xil_create_from_window()`
di-250	XIL Xlib I/O Driver: Could not create ximage	`xil_create_from_window()`
di-251	Number of bands in src image does not match dither mask	`xil_ordered_dither()`
di-252	Invalid colorcube dimensions	`xil_colorcube_create()`
di-253	XilCis: Cell compress image width is not an even multiple of 4	`xil_compress()`
di-254	XilCis: Cell compress image height is not an even multiple of 4	`xil_compress()`
di-255	Could not copy image	`xil_create_copy()`
di-256	Could not install error handler	`xil_install_error_handler()`
di_257	Invalid NULL ROI specified	Any of the `xil_roi*()` functions
di-258	Seed pixel not within fill region defined by fill color	`xil_soft_fill()`
di-259	Invalid NULL data pointer	`xil_dithermask_create()`, `xil_histogram_create()`, `xil_kernel_create()`, `xil_lookup_get_values()`, `xil_lookup_set_values()`, `xil_sel_create()`, `xil_device_create()`, `xil_interpolation_table_create()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error

di-260	Invalid NULL system state	`xil_create(),`
		`xil_create_from_type(),`
		`xil_create_from_device(),`
		`xil_create_from_window()`
		`xil_create_temporary(),`
		`xil_create_temporaty_from_type(),`
		`xil_cis_create(),`
		`xil_roi_create(),`
		`xil_kernel_create(),`
		`xil_dithermask_create(),`
		`xil_sel_create(),`
		`xil_lookup_create(),`
		`xil_colorcube_create(),`
		`xil_histogram_create(),`
		`xil_close(),`
		`xil_state_get_synchronize(),`
		`xil_state_set_synchronize(),`
		`xil_state_get_show_action(),`
		`xil_state_set_show_action(),`
		`xil_install_error_handler(),`
		`xil_remove_error_handler(),`
		`xil_image_get_by_name(),`
		`xil_lookup_get_by_name(),`
		`xil_imageytpe_get_by_name(),`
		`xil_cis_get_by_name(),`
		`xil_dithermask_get_by_name(),`
		`xil_kernel_get_by_name(),`
		`xil_sel_get_by_name(),`
		`xil_roi_get_by_name(),`
		`xil_histogram_get_by_name(),`
		`xil_colorspace_get_by_name()`

di-261	XilCis: Attempted to set Cell encoding type to unknown value	`xil_cis_set_attribute()` `("CELL_ENCODING_TYPE")`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-262	Invalid NULL SEL specified	`xil_sel_get_width()`, `xil_sel_get_height()`, `xil_sel_get_key_x()`, `xil_sel_get_key_y`, `xil_sel_cretae_copy()`
di-263	Invalid kernel value (NaN)	`xil_kernel_create()`
di-264	Invalid kernel value (infinity or -infinity)	`xil_kernel_create()`
di-265	Invalid NULL histogram specified	`xil_histogram_get_values()`, `xil_histogram_get_limits()`, `xil_histogram_get_info()`, `xil_histogram_get_nbins()`, `xil_histogram_get_nbands()`, `xil_histogram_destroy()`

di-266	Invalid image dimensions specified	`xil_create()`,
		`xil_create_from_type()`
		`xil_create_temporary()`

di-267	Could not copy dither mask	`xil_ordered_dither()`, `xil_dithermask_create_copy()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-268	Invalid NULL object pointer	`xil_lookup_get_version()`, `xil_object_get_type()`, `xil_cis_get_name()`, `xil_cis_set_name()`, `xil_dithermask_get_name()`, `xil_dithermask_set_name()`, `xil_get_name()`, `xil_set_name()`, `xil_histogram_get_name()`, `xil_histogram_set_name()`, `xil_imagetype_get_name()`, `xil_imagetype_set_name()`, `xil_kernel_get_name()`, `xil_kernel_set_name()`, `xil_lookup_get_name()`, `xil_lookup_set_name()`, `xil_roi_get_name()`, `xil_roi_set_name()`, `xil_sel_get_name()`, `xil_sel_set_name()`
di-269	Could not set image ROI	Internal error
di-271	Invalid dither mask value (NaN)	`xil_dithermask_create()`
di-272	Invalid dither mask value (greater than 1.0 or less than 0.0)	`xil_dithermask_create()`
di-273	Invalid SEL value (must be 0 or 1)	`xil_sel_create()`
di-274	Could not create internal symbol table	`xil_cis_create()`
di-275	Could not create internal Cell compressor object	`xil_cis_create()`
di-276	Internal error op number not found	Internal error
di-277	Could not create internal CisBufferManager object	`xil_cis_create()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-278	Could not create internal base `XilDeviceCompression` object	`xil_cis_create()`
di-279	Could not set object name	`xil_cis_set_name()`, `xil_dithermask_set_name()`, `xil_set_name()`, `xil_histogram_set_name()`, `xil_imagetype_set_name()`, `xil_kernel_set_name()`, `xil_lookup_set_name()`, `xil_roi_set_name()`, `xil_sel_set_name()`. A secondary error.
di-280	Could not create `JpegLL` compressor object	`xil_cis_create()`
di-281	Could not create `JpegLL` decompressor object	`xil_cis_create()`
di-282	Could not create `XilCis`	`xil_cis_create()`
di-283	XilCis: Cell user data size is too large	`xil_cis_set_attribute()` (`"CELL_USER_DATA"`)
di-284	XilCis:Bbitstream error: Invalid SOF marker	`xil_decompress()`, `xil_cis_number_of_frames()`, `xil_cis_get_attribute()`, `xil_cis_set_attribute()`, `xil_cis_get_input_type()`, `xil_cis_get_output_type()`, `xil_cis_attempt_recovery()`, `xil_cis_get_bits_ptr()`
di-285	XilCis: Invalid bitstream parameters	`xil_decompress()`, `xil_cis_number_of_frames()`, `xil_cis_get_attribute()`, `xil_cis_set_attribute()`, `xil_cis_get_input_type()`, `xil_cis_get_output_type()`, `xil_cis_attempt_recovery()`, `xil_cis_get_bits_ptr()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-286	Could not create full set of standard objects	`xil_open()`. A secondary error.
di-287	Insert of operation failed	`xil_error_diffusion()`, `xil_nearest_color()`, `xil_ordered_dither()`, `xil_squeeze_range()`. A secondary error.
di-288	XilCis: Image for compress has ROIs or non-zero origin	`xil_compress()`
di-289	Invalid data in kernel for use with error diffusion	`xil_error_diffusion()`
di-290	Partial frame start adjustment error on frame position	`xil_decompress()` (with partial frame in the CIS), `xil_cis_number_of_frames()`, `xil_cis_get_attribute()`, `xil_cis_set_attribute()`, `xil_cis_get_input_type()`, `xil_cis_get_output_type()`, `xil_cis_attempt_recovery()`, `xil_cis_get_bits_ptr()`
di-291	Partial frame flag not set on buffer with partial frame	`xil_decompress()`, `xil_cis_number_of_frames()`, `xil_cis_get_attribute()`, `xil_cis_set_attribute()`, `xil_cis_get_input_type()`, `xil_cis_get_output_type()`, `xil_cis_attempt_recovery()`, `xil_cis_get_bits_ptr()`
di-292	Partial frame start adjustment error—mismatch with buffer start	`xil_decompress()`, `xil_cis_number_of_frames()`, `xil_cis_get_attribute()`, `xil_cis_set_attribute()`, `xil_cis_get_input_type()`, `xil_cis_get_output_type()`, `xil_cis_attempt_recovery()`, `xil_cis_get_bits_ptr()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-293	Current buffer manager read frame unknown	Internal error
di-294	XilCis: Could not derive output type from data in CIS	`xil_decompress()`
di-295	Colorspace and image's number of bands mismatch	`xil_set_colorspace()`
di-296	Invalid NULL colorspace specified	`xil_color_convert()`. An internal error.
di-297	Could not copy colorspace	`xil_color_convert()`. An internal error.
di-298	Could not create colorspace	`xil_open()`. An internal error, probably due to running out of memory.
di-299	Number of bands in destination image does not match lookup table	`xil_lookup()`
di-300	Could not create internal fax compressor	`xil_compress()`
di-301	Invalid NULL X region specified	`xil_roi_add_region()`
di-302	Invalid colorcube multipliers	`xil_colorcube_create()`
di-303	XilCis: Invalid Cell key frame interval – negative	`xil_cis_set_attribute()` `(KEYFRAME_INTERVAL)`
di-304	XilCis: Illegal seek backward in a non-random access CIS	`xil_cis_seek()`
di-305	Bad value for `Xil_boolean` type	`xil_cis_set_attribute()` `(TEMPORAL_FILTERING,` `COLORMAP_ADAPTATION)`
di-306	XilCis: Cell colormaps must contain at least 2 colors	`xil_cis_set_attribute()` `(COMPRESSOR_MAX_CMAP_SIZE,` `COMPRESSOR_COLORMAP)`
di-307	Cell colormaps cannot contain more than 256 colors	`xil_cis_set_attribute()` `(COMPRESSOR_MAX_CMAP_SIZE)`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-308	XilCis: Cell `COMPRESSOR_MAX_CMAP_SIZE` attribute cannot be changed after first `xil_compress()` call	`xil_cis_set_attribute()` `(COMPRESSOR_MAX_CMAP_SIZE)`
di-310	XilCis: Quantizer value out of range	`xil_cis_set_attribute()` `(QUANTIZATION_TABLE)`
di-311	XilCis: Failure decompressing Cell header	`xil_decompress()`
do-312	XilCis: Illegal H.261 bitstream	`xil_decompress()`
di-313	Compressor for px64 device compression not yet implemented	`xil_compress()`
di-314	XilCis: Unable to sync up with current frame PSC in H.261 bitstream	`xil_decompress()`
di-315	XIL I/O Driver: `uname` system call failed	`xil_create_from_window()`
di-316	XIL Xlib I/O Driver: Could not get ximage	`xil_get_pixel()`
di-317	XilCis: Error in seeking. No frames to skip/burn	`xil_cis_seek()`
di-318	XilCis: Attribute error: Value must be either TRUE or FALSE	`xil_cis_set_attribute()`
di-319	XilCis: No user data established for already scanned Mpeg frame	`xil_cis_seek()`
di-320	XIL I/O Driver: Could not get frame buffer info	`xil_create_from_window()`
di-321	Cannot resize a child image	Internal error

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-322	Cannot resize non-device images	Internal error
di-323	Compressor for MPEG-1 device compression not yet implemented	`xil_compress()`
di-324	XilCis: Attribute error: Value out of range	`xil_cis_set_attribute()`
di-325	XilCis: D (DC intra-coded) frame type not supported	`xil_decompress()`
di-326	Matrix size does not match with src or dst number of bands	`xil_band_combine()`
di-327	Tried to use an invalid image in an operation	Any operation with an image that becomes invalid. An image becomes invalid when an operation that is supposed to write to it fails.
di-328	Internal error in ROI scan conversion	Internal error
di-329	XilCis: Mpeg1 bitstream error, incomplete frame	Any routine that involves a seek operation within a CIS
di-330	XIL Xlib I/O Driver: Could not capture image—window not mapped	`xil_copy()`
di-331	Null lookup list or zero number of lookups to combine	`xil_lookup_create_combined()`
di-332	Input type of lookups are not of same type	`xil_lookup_create_combined()`
di-333	Output type of lookups are not of same type	`xil_lookup_create_combined()`
di-334	Not all lookups have a single band	`xil_lookup_create_combined()`

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-335	Requested lookup table doesn't exist—`num_bands` too large	`xil_lookup_create_combined()`
di-336	Wrong lookup type—not a combined lookup object	`xil_lookup_create_combined()`
di-337	XilCis: Wrote more than `max_frame_size` bytes into frame	`xil_compress()`
di-338	XilCis: `moveEndStart` error, read frame must be at end of buffer	`xil_cis_get_bits_ptr()`
di-339	XilCis: `moveEndStart` error, read frame must have following frame	`xil_cis_get_bits_ptr()`
di-340	XilCis: `removeStartFrame` error, read frame must be start of buffer	`xil_cis_get_bits_ptr()`
di-341	XilCis: mpeg1 seek, `prev_nonbframe_id` is frame which already has valid `display_id`	`xil_cis_seek()`, `xil_decompress()` for MPEG-1 CIS
di-342	XilCis: `doneBufferSpace()` called with negative number of bytes, illegal	`xil_compress()`
di-343	Unsupported edge detection method	`xil_edge_detection()`
di-344	Invalid NULL attribute specified	`xil_attribute_set_value()`, `xil_device_create()`
di-347	ROI rectangle width/height cannot be <= 0	Any function that passes an image as an argument
di-348	NULL warp table specified	`xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-349	Warp table datatype mismatch	`xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`
di-350	Warp table nbands mismatch	`xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`
di-351	Could not create Interpolation Table	`xil_affine()`, `xil_interpolation_table_create()`, `xil_rotate()`, `xil_scale()`, `xil_tablewarp()`, `xil_tablewarp_horizontal()`, `xil_tablewarp_vertical()`, `xil_translate()`
di-352	Invalid NULL Interpolation Table specified	`xil_interpolation_table_create()`
di-354	X & Y skip values must have value greater than 0	`xil_histogram()`, `xil_histogram_create()`
di-356	Cannot open specified data file(s)	Could occur with any function whose source image is a Photo CD device image.
di-358	Width and height must be defined before parsing `XilCis`	`xil_decompress()` of a CellB CIS
di-359	Cannot read from a write-only device	Could occur with any function that tries to read from a write-only device
di-360	Cannot write to a read-only device	Could occur with any function that tries to write to a read-only device
di-361	Invalid default tile size specified for current tiling mode	
di-362	I/O device failed to provide a required attribute for its type and was not loaded	

B

Table B-1   XIL Library Error Messages   *(Continued)*

Error Number	Error Message	Function That Generates Error
di-364	Unable to process configuration information in OWconfig file(s)	
di-365	Mismatch in expected number of boxes for XilBoxList::getNext() -- full operation was not done	
di-366	Cannot re-size an exported image	
di-367	X scale factor not in the range [0.0, 1.0]	
di-368	Y scale factor not in the range [0.0, 1.0]	
di-369	Could not dlopen() libdga.so.1	
di-370	Could not dlsym() required DGA symbol from libdga.so.1	
di-371	xil_get/set_memory_storage not supported for given datatyp	
di-372	Invalid NULL XilColorspaceList specified	
di-373	Invalid NULL XilColorspace given for the XilColorspaceList	
di-374	Could not create XilStorage	
di-375	Could not create XilColorspaceList	
di-376	I/O device operation failed	
di-377	Storage device operation failed	

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-378	Attempted an operation which is not supported for this type of I/O device	
di-379	Could not dlopen() libkcs.so.1	
di-380	Could not dlsym() required KCMS symbol from libkcs.so.1	
di-381	KCMS function did not return KCS_SUCCESS	
di-382	Cannot set a non XIL_COLORSPACE_NAME XilColorspace on an image	
di-383	Invalid NULL XilStorage object specified	
di-384	Unable to Sync	
di-385	Unsupported datatype for this operation	
di-386	Device handler specified invalid an operation branch number of '%d'	
di-387	Device handler's describeMembers() returned failure. '%s' not loaded	
di-391	Could not copy XilHistogram	
di-392	Could not initialize the image tile size	
di-395	Could not get image tile size	
di-396	Cannot set or get tile size on a non-exported image	

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-398	xil_convert() is not supported for combined lookups	
di-399	Copy failed when in updating version for non-deferrable object with deferrable dependents	
di-400	Could not successfully execute operation	
di-401	Could not get source ROI in global space	
di-402	Could not get destination ROI in global space	
di-403	Adding box entry to boxlist failed	
di-404	Could not lock DAG	
di-405	Datatype mismatch when setting storage information	
di-406	Band offset is too large for setting storage information	
di-407	Invalid storage type encountered	
di-408	When tiling mode is XIL_STRIPPING, tile X size must be 0	
di-409	Encountered an invalid tileing mode	
di-411	Thread creation failed	
di-412	Error in loading device handler %s	
di-414	Requested name of an invalid object type	

Table B-1 XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-415	Unable to set preprocess routine information for the operation	
di-416	Invalid NULL interpolation specified	
di-417	Unable to set input type on device compressor	
di-418	Could not decompress header information	
di-421	Unsupported interpolation type specified	
di-422	Initializing XilTileList class failed	
di-423	Set memory storage failed	
di-424	Reducing op queue length has failed	
di-425	Updating operation status encountered unknown status -- operation may not have executed	
di-426	Setting invalid band or parameter on storage object	
di-427	Could not determine source position number in evaluating dependent	
di-428	Invalid operation for a temporary image	`xil_export()`, `xil_get_pixel()`, `xil_set_colorspace()`, `xil_set_origin()`, `xil_set_pixel()`, `xil_set_roi()`
di-429	Using destroyed image in operation	

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-430	Missing mutex reference while operation evaluating -- unable to wait	
di-431	Convex region intersection generated more than maximum of 16 vertices	
di-432	ROI transpose failed	
di-433	ROI unite failed	
di-434	Operation source #1 has incorrect datatype	
di-435	Operation source #2 has incorrect datatype	
di-436	Operation source #3 has incorrect datatype	
di-437	Operation destination has incorrect datatype	
di-438	Illegal operation for an imported image	
di-439	Operation destination has incorrect number of bands	
di-440	Could not copy XilImageType	
di-441	Could not copy XilStorage	
di-442	Could not copy XilInterpolationTable	
di-443	Tile number is higher than there are tiles in `setOpStatus()`	
di-444	Tile number is higher than there are tiles in `getOpStatus()`	

Table B-1   XIL Library Error Messages *(Continued)*

Error Number	Error Message	Function That Generates Error
di-445	Could not translate from one internal ROI implementation to another	
di-446	Invalid `NULL` affine transfomation matrix specified	
di-447	Source smaller than kernel/sel not yet implemented	
di-448	Invalid XIL XilImageType specified	`xil_create_from_type()`, `xil_create_temporary_from_type()`
di-449	Attempting to change a non-NULL DAG reference	
di-450	Non-integral sampling ratios not yet implemented	
di-451	Setting the tile size on a child image is not supported	
di-452	Mismatch of tiling mode and tile size	

B

XIL-XGL Interoperability C≡

Using XIL and XGL Calls in the Same Program

The XIL imaging library includes two functions to process an image using both XIL and XGL 3.*x* calls in the same program.

Note – This interface is provided in the form of example code that you must build into a static library.

The code for these two functions can be found in the `convert.c` source file, which is located in the `/usr/openwin/demo/xil/libxil_to_xgl` directory. The `Makefile` in this directory is provided to build the code into `libxil_to_xgl.a`. This means that programs that use the conversion functions must link with:

- `-lxil`
- `-lxgl`
- `-lxil_to_xgl`

With `xil_to_xgl()`, you can convert an XIL memory image to an XGL memory raster *or* you can convert an XIL display image to an XGL window raster. This is the prototype for this function.

```
Xgl_ras
xil_to_xgl(XilImage src, Xgl_sys_state xgl_state);
```

The `src` parameter is a handle to an XIL memory image or display image, and `xgl_state` is a handle to an XGL System State object. If the function is successful, the return value is a handle to an XGL memory raster or window raster. If the functions fails, the return value is `NULL`.

With the other function, `xgl_to_xil()`, you can convert an XGL memory raster to an XIL memory image or you can convert an XGL window raster to an XIL display image.

```
XilImage
xgl_to_xil(Xgl_ras src, XilSystemState xil_state);
```

The `src` parameter is a handle to an XGL memory raster or window raster, and `xil_state` is an XIL system state. The function's return value is a handle to an XIL memory image or display image. If the function fails, the return value is `NULL`.

Conversions

Table C-1 lists the specific conversions that these functions make possible. Within each row, the XIL image listed in the left column can be converted to the XGL object in the right column and vice versa.

Table C-1   XIL-XGL Interoperability

XIL Image	XGL Object
A one-band `XIL_BYTE` memory image	A memory raster with the attribute `XGL_COLOR_INDEX`
A four-band `XIL_BYTE` memory image	A memory raster with the attribute `XGL_COLOR_RGB`
A one-band `XIL_BYTE` display image	A window raster with the attribute `XGL_COLOR_INDEX`
A three-band `XIL_BYTE` display image	A window raster with the attribute `XGL_COLOR_RGB`

Note – The conversions shown in the first row of Table C-1 are possible only if the width of the XIL image is an even number.

C ≡

Restrictions

Once you have both an XIL and an XGL handle to an object, you can process it using either XIL functions or XGL functions. The only restriction on combining these calls is that there cannot be any outstanding deferred operations on the object at the time you switch from making XIL calls to XGL calls, or vice versa. This means that if you have been processing an image using XIL functions, you must call `xil_sync()` to flush all outstanding operations on the image before operating on the image using XGL functions. Similarly, if you have been processing a raster using XGL functions, you must call `xgl_context_flush()` before performing any XIL operations on the raster.

≡ *C*

Cell and Cell B Bytestream Definitions **D☰**

This appendix provides an overview of the Cell technology, then focuses on the descriptions of the codes that are used in a Cell bytestream. Although some rationale for the existence of the various codes is discussed, specific implementation information about Cell encoders or decoders is not provided.

This appendix has the following main sections:

Introduction to Cell

The Cell image compression technology, which was developed by Sun Microsystems™, provides high-quality, low bit-rate image compression at low computational cost. Applications where Cell compression can be used include videoconferencing, media distributions on CD-ROM, and multimedia mail applications. The bytestream that is produced by a Cell encoder is variable length and is made up of instructional codes and information about the compressed video image.

There are two versions of the Cell image compression technology: Cell and CellB. Cell compression, which is designed for use with movies, is computationally asymmetric; it takes longer to compress video data than it

does to decompress the data. To provide high-quality images, Cell supports Adaptive Colormap Selection, which enables an encoder to change colormaps dynamically. The bytestream of images that are compressed with Cell can be decompressed in software on any SPARCstation™ workstation. Depending on compression ratios, 640x480-resolution movies can be played back at 30 frames/second (fps). Multiple smaller movies can be displayed simultaneously at this rate.

CellB, which is derived from Cell, is designed for use in videoconferencing applications. To reduce computational overhead and meet the timing demands of videoconferencing, CellB compression is more computationally symmetric than Cell. CellB uses a fixed colormap and is designed to use vector quantization techniques. The bytestream of a CellB image is simpler (and more compact) than that of a Cell image, which reduces requirements on network bandwidth.

Encoding Images for Cell

Cell works with 3-band RGB images, with no subsampling, and requires that the width and height of the images be divisible by four. CellB takes industry-standard 3-band YC_bC_r video as input.

A cell encoder breaks the video into cells. A cell is 16 pixels, arranged in a 4-by-4 group, as shown in Figure D-1. Cells are encoded into the bytestream in scanline order, from left to right and from top to bottom.

Figure D-1   Cell

The basic encoding scheme used in both versions of Cell is based on an image coding method called Block Truncation Coding (BTC). The 16 pixels in a cell are represented by a 16-bit mask and two colors. The values in the mask

specify which color to place at each of the pixel positions. The mask and colors can be chosen to maintain certain statistics of the cell or to reduce contouring in a manner similar to ordered dither.

The primary advantage of BTC is that its decoding process is similar to the operation of character fonting in a color frame buffer. The character display process for a frame buffer takes as input a foreground color, a background color, and a mask that specifies whether to use the foreground or background color at each pixel. Because this function is so important to the window system, it is often implemented as a display primitive in graphics accelerators. The Cell compression technique leverages these existing primitives to provide full-motion video decoding without special hardware or modifications to the window system.

The basic component of the Cell and CellB bytestreams is the four-byte cell code. The first two bytes of the cell code are a bitmask, as shown in Figure D-2. Each bit in the mask represents a pixel in a 16-bit cell. The bitmask is normalized so that the most significant bit is 0. The figure shows the relationship of the bits in the mask to the location of the pixels in a cell. A value of 0 in a mask bit means that the pixel is rendered in the background color (color 0). A value of 1 means that the pixel is rendered in the foreground color (color 1).

The last two bytes of a cell code establish a pixel's color. Cell and CellB differ in the way that pixel colors are derived. In Cell, the values in the color 0 and color 1 bytes are indexes into a colormap. In CellB, the values are indexes into Y/Y and C_b/C_r vector quantization tables.

Cell from a 24-bit image

Cell Encoder

Cell Code (Four Bytes)

Pixel Bitmask

15   0   Color 1   Color 0

| 0 | M | M | M | M | M | M | M | M | M | M | M | M | M | M | BBBBBBBB | FFFFFFFF |

Cell A

| C_b/C_r Vector | Y/Y Vector |

Cell B

Always
Normalized
To 0

0 `	`M14	M13	M12
M11	M10	M09	M08
M07	M06	M05	M04
M03	M02	M01	M00

Figure D-2   Encoding a Cell

Cell Bytestream Description

The Cell bytestream can contain the following codes:

- Key-frame headers
- Key parameters
- Cell codes
- Run length codes
- Escape codes

At a minimum, the Cell bytestream must contain a key-frame header, a key frame, and cell codes. More compression is provided by using run length codes for cells. The escape codes enable you to skip frames, load colormaps, change masks and colors, and include non-video (user) data in the bytestream.

The codes are described in the following sections.

Key-Frame Header and Key Parameters

A Cell bytestream contains periodic *key* frames, which are free of interframe escape codes. Every key frame is preceded by a key-frame header, which is described below. Immediately after the key-frame header, the colormap is encoded. If the colormap has not changed since the previous frame, it is repeated anyway. The first frame of a Cell movie is always a key frame.

Key-Frame Header

The key-frame header begins with a series of 8 bytes of all 1's (0xff). This code is followed by a number of parameters, which are introduced by 1-byte codes. The end of a key-frame header is marked by an all-zero byte.

```
Start of Key-Frame Header
┌─────────────────────────────────────────────┐
│ 11111111   11111111   11111111   11111111    │
│ 11111111   11111111   11111111   11111111    │
└─────────────────────────────────────────────┘
              .
              .
              .
       Parameters
              .
              .
End of Key-Frame Header
┌──────────┐
│ 00000000 │
└──────────┘
```

≡ *D*

Image Dimensions Key-Frame Code

The following code precedes the 16-bit unsigned width and height of an image.

Code	Width	Height
00000001	WWWWWWWW WWWWWWWW	HHHHHHHH HHHHHHHH

Frame Rate Key-Frame Code

The following code precedes a 32-bit unsigned number that sets the frame rate in microseconds per frame.

Code	Frame Rate
00000010	RRRRRRRR RRRRRRRR RRRRRRRR RRRRRRRR

Size of Colormap Key-Frame Code

The following code precedes a byte that specifies the maximum size (Count+1) of a colormap in the bytestream.

Code	Count
00000011	CCCCCCCC

Reserved Key-Frame Codes

The key-frame header codes in the range 00000000 to 11111111 (inclusive) are reserved.

Example Key-Frame Header

As an example, a key-frame header for a 320-by-240 movie with a frame rate of 30 fps and a maximum colormap size of 240 would look like this (in hex):

```
ff ff ff ff ff ff ff ff        Start of key-frame header
01 01 40 00 f0                  Dimensions 320 (0x140) by 240 (0xf0)
02 00 00 82 35                  33,333 (0x8235) usecs per frame
03 ef                           Max colormap size is 239 (0xef) + 1
00                              End of key-frame header
c0 ...                          Escape code for colormap
```

Cell Code

The four-byte code that describes a 4-by-4 Cell cell is shown below. The values in the Color1 and Color0 bytes are indexes into the current colormap. If a pixel's bit is set, the color that is indexed in the Color1 byte is used. If not set (0), the pixel is rendered with the color that is indexed by the value of the Color0 byte.

```
    4x4 Bitmask          Color1     Color0
0MMMMMMM   MMMMMMMM   FFFFFFFF   BBBBBBBB
```

The relationship of the bitmask bits to the pixels in a cell is shown below.

```
                  Pixel Bitmask
    15  ◄──────────────────────────── 0
    0 M M M M M M M   M M M M M M M M

              ┌────┬────┬────┬────┐
              │ 0  `│`M14│ M13│ M12│
   ────►      ├────┼────┼────┼────┤
              │ M11│ M10│ M09│ M08│
 Always 0     ├────┼────┼────┼────┤
              │ M07│ M06│ M05│ M04│
              ├────┼────┼────┼────┤
              │ M03│ M02│ M01│ M00│
              └────┴────┴────┴────┘
```

☰ *D*

Run Length Code

Runs of cells that use the same color can be described by a run length code, which is shown below. The code causes the next Count+1 cells to be rendered in the color that is specified in the Color0 byte.

```
   Runlength Code      Color0      Count
  00000000  00000000   BBBBBBBB   CCCCCCCC
```

Escape Codes

A Cell encoder can use the following escape codes to perform operations such as loading colormaps, skipping frames, and including user data in the compressed bytestream. All escape codes start with a 1 in the code's most significant bit.

Skip Cells

The following code causes the next N+1 cells to be skipped (to a maximum of 64 cells).

```
  Code
  10NNNNNN
```

Load a New Colormap

The following code loads a new colormap. Count+1 byte-triples (red byte, green byte, and blue byte) follow the Count byte.

```
  Code        Count
  11000000   CCCCCCCC     rrrrrrrr  gggggggg   bbbbbbbb
```

Use Same Mask With New Colors

The following code uses the same mask as the previous cell, but with new colors.

```
Code        Color0      Color1
11000010   BBBBBBBB   FFFFFFFF
```

Use Same Mask With Color 0

The following code uses the same mask as the previous cell, but with a new color as indexed by the value of the Color0 byte.

```
Code        Color0
11000011   BBBBBBBB
```

Use Same Mask With Color 1

The following code uses the same mask as the previous cell, but with a new color as indexed by the value of the Color1 byte.

```
Code        Color1
11000100   FFFFFFFF
```

Use Same Colors With New Mask

The following code uses the same colors as the previous cell, but with a new bitmask.

```
Code          4x4 Bitmask
110000101 MMMMMMMM   MMMMMMMM
```

≡ *D*

Include User Data

The following code enables you to include your own data in the bytestream. The next N+1 bytes in the bytestream are user data.

```
Code              Byte Count
11000110 │ NNNNNNNN  NNNNNNNN  NNNNNNNN │ uuuuuuuu   uuuuuuuu
```

Skip an Entire Frame

The following code causes the next frame to be skipped.

```
Code
11000111
```

First Byte of Key-Frame Header

The following code is the first byte of a key-frame header, which was described earlier.

```
Code
11111111
```

Reserved Codes

The codes in the range 11001000 to 11111110 (inclusive) are reserved.

Summary of Cell Codes

Table D-1 lists the Cell codes.

Table D-1 Cell Bytestream Codes

Code	Description
0MMMMMMM MMMMMMMM FFFFFFFF BBBBBBBB	Cell code: M=bit mask, F=color 1 index, B=color 0 index
00000000 00000000 BBBBBBBB CCCCCCCC	Run length code: B=color 0 index, cell count = C+1
10NNNNNN	Interframe skip: cell count N+1
11000000 CCCCCCCC	Load new colormap: count of entries (RGB triples) that follow = C+1
11000010 BBBBBBBB FFFFFFFF	Same mask, new colors: B=color 0 index, F=color 1 index
11000011 BBBBBBBB	Same mask, new color 0: B=color 0 index
11000100 FFFFFFFF	Same mask, new color 1: F=color 1 index
11000101 0MMMMMMM MMMMMMMM	Same colors, new mask: M=bit mask
11000110 NNNNNNNN NNNNNNNN NNNNNNNN	User data: count of user bytes that follow = C+1
11001000–11111111	Reserved
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111	Start of key-frame header
00000001 WWWWWWWW WWWWWWWW HHHHHHHH HHHHHHHH	Key frame, image dimensions: W=width, H=height
00000010 RRRRRRRR RRRRRRRR RRRRRRRR RRRRRRRR	Key frame, frame rate in usec: R=frame rate
00000011 CCCCCCCC	Key frame, maximum colormap size = C+1
00000000	End of key-frame header

CellB Bytestream Description

CellB compression is designed for use in videoconferencing applications. Rather than indexing into a colormap to determine pixel colors (as in Cell), CellB is designed to be used with vector quantization and dequantization techniques in the YC_bC_r color space.

≡ *D*

The CellB bytestream contains no information about image size and frame rates. It's the responsibility of the videoconferencing application to provide this information.

The CellB bytestream consists of:

- Cell codes
- Skip codes
- Quantization-table specification codes

```
              Cell Code              Skip Code                Quantization Table
                                                                    Code
 0MMMMMMM MMMMMMMM FFFFFFFF BBBBBBBB 100SSSSS 0MMMMMMM 11111111T  ⌐  ⌐
```

Cell Code

The four-byte code that describes a 4-by-4 CellB cell is shown below. There are two possible luminance (Y) levels for each cell but only one pair of chrominance (C_b and C_r) values.

The value in the C_b/C_r byte represents the chrominance component of the cell. The value in the Y/Y byte represents two luminance values (Y_0 and Y_1) that can represent the cell's luminance.

```
    4x4 Bitmask        U/V Code   Y/Y Code
 0MMMMMMM   MMMMMMMM  UVUVUVUV   YYYYYYYY
```

If a pixel's bit is set in the 4-by-4 bitmask, the color that is rendered for the pixel is $<Y_1, C_b, C_r>$. If a pixel's bit is not set, the pixel is rendered by the color represented by $<Y_0, C_b, C_r>$.

The relationship of the bitmask bits to a cell's pixels is shown below.

C_b/C_r Quantization Table

The C_b/C_r field of the CellB bytecode represents the chrominance component of the cell. This C_b/C_r code is an index into a table of vectors that represent two independent components of chrominance. Figure D-3 on page 556 shows the default chrominance table. "Cb/Cr Table Values" on page 564 contains a list of the values used in the table.

The distribution of values in this default table is based on the observation that, in videoconferencing applications, a cell's C_b/C_r vectors are clustered around the origin (0,0). Therefore, the C_b/C_r codeword is circularly symmetric, with higher densities near the origin.

Figure D-3   Default CellB Chrominance Quantization Table

Y/Y Quantization Table

The Y/Y field of the CellB bytecode represents two luminance values of a cell (Y_0 and Y_1). This Y/Y code is an index into a table of two-component luminance vectors. Figure D-4 on page 558 shows the default luminance table. "Y/Y Table Values" on page 560 provides the list of values for the table.

The distribution of values in the default luminance table is statistically optimized. The quantizer takes advantage of the high correlation of luminance values within local regions of a cell. This results in a set of representative vectors that are most densely populated around the diagonal, where y1 equals y2.

An observer's sensitivity to contrast is also taken into account, resulting in a distribution of points that is farther apart in regions where the contrast between two values is low.

In Figure D-4, the value for the Y/Y code is selected by approximating the mean luminance for the cell. Then, the pixels within the cell are separated into groups that are above and below the mean luminance. The mean luminance of these two groups is used to index a two-dimensional vector quantizer, which returns a byte value for the Y/Y code.

Figure D-4   Default CellB Luminance Quantization Table

Skip Code

The single-byte CellB skip code is shown below.

```
100SSSSS
```

The skip code tells the decoder to skip the next S+1 cells in the frame that is being decoded, which, for CellB, supports a simple form of interframe encoding. There are five skip bits, so that a maximum of 32 cells can be skipped with a single-byte skip code.

New Y/Y Table

The single-byte "new Y/Y table" code is shown below.

```
11111110
```

This code tells the decoder that the next 512 bytes are a new Y/Y quantization table. The bytes are arranged as:

```
Y1_000   Y2_000
Y1_001   Y2_001
.        .
.        .
.        .
Y1_255   Y2_255
```

Note – This code is not implemented in the current CellB compressor and decompressor.

New U/V Table

The single-byte "new U/V table" code is shown below.

```
11111111
```

This code tells the decoder that the next 512 bytes are a new U/V quantization table. The bytes are arranged as:

```
U_000    V_000
U_001    V_001
.        .
```

```
.          .
.          .
U_255     V_255
```

> **Note** – This code is not implemented in the current CellB compressor and decompressor.

Default CellB Quantization Tables

Y/Y Table Values

Table D-2 lists the default values for the CellB Y/Y quantization table.

Table D-2 Default CellB Y/Y Table *(1 of 4)*

Index	Y1	Y2	Index	Y1	Y2	Index	Y1	Y2
0	16	20	87	120	136	174	112	64
1	16	24	88	128	136	175	128	64
2	16	32	89	128	144	176	72	68
3	16	48	90	128	160	177	76	72
4	16	64	91	128	176	178	80	72
5	16	80	92	128	192	179	88	72
6	16	112	93	136	144	180	80	76
7	16	144	94	136	152	181	84	80
8	16	176	95	144	152	182	88	80
9	16	208	96	144	160	183	96	80
10	16	240	97	144	176	184	112	80
11	20	24	98	144	192	185	128	80
12	24	28	99	144	208	186	144	80
13	24	32	100	144	240	187	176	80
14	24	40	101	152	160	188	208	80
15	28	32	102	152	168	189	240	80
16	32	36	103	160	168	190	88	84

Table D-2 Default CellB Y/Y Table *(2 of 4)*

Index	Y1	Y2	Index	Y1	Y2	Index	Y1	Y2
17	32	40	104	160	176	191	92	88
18	32	48	105	160	192	192	96	88
19	32	64	106	160	208	193	104	88
20	32	80	107	160	224	194	96	92
21	32	96	108	168	176	195	100	96
22	36	40	109	168	184	196	104	96
23	40	44	110	176	184	197	112	96
24	40	48	111	176	192	198	128	96
25	40	56	112	176	208	199	144	96
26	44	48	113	176	224	200	160	96
27	48	52	114	176	240	201	104	100
28	48	56	115	184	192	202	108	104
29	48	64	116	184	200	203	112	104
30	48	80	117	192	200	204	120	104
31	48	96	118	192	208	205	112	108
32	48	112	119	192	224	206	116	112
33	48	144	120	192	240	207	120	112
34	48	176	121	200	208	208	128	112
35	48	208	122	200	216	209	144	112
36	48	240	123	208	216	210	160	112
37	52	56	124	208	224	211	176	112
38	56	60	125	208	240	212	208	112
39	56	64	126	216	232	213	240	112
40	56	72	127	224	240	214	128	120
41	60	64	128	20	16	215	136	120
42	64	68	129	24	16	216	136	128
43	64	72	130	32	16	217	144	128

D

Table D-2 Default CellB Y/Y Table *(3 of 4)*

Index	Y1	Y2	Index	Y1	Y2	Index	Y1	Y2
44	64	80	131	48	16	218	160	128
45	64	96	132	64	16	219	176	128
46	64	112	133	80	16	220	192	128
47	64	128	134	112	16	221	144	136
48	68	72	135	144	16	222	152	136
49	72	76	136	176	16	223	152	144
50	72	80	137	208	16	224	160	144
51	72	88	138	240	16	225	176	144
52	76	80	139	24	20	226	192	144
53	80	84	140	28	24	227	208	144
54	80	88	141	32	24	228	240	144
55	80	96	142	40	24	229	160	152
56	80	112	143	32	28	230	168	152
57	80	128	144	36	32	231	168	160
58	80	144	145	40	32	232	176	160
59	80	176	146	48	32	233	192	160
60	80	208	147	64	32	234	208	160
61	80	240	148	80	32	235	224	160
62	84	88	149	96	32	236	176	168
63	88	92	150	40	36	237	184	168
64	88	96	151	44	40	238	184	176
65	88	104	152	48	40	239	192	176
66	92	96	153	56	40	240	208	176
67	96	100	154	48	44	241	224	176
68	96	104	155	52	48	242	240	176
69	96	112	156	56	48	243	192	184
70	96	128	157	64	48	244	200	184

Table D-2 Default CellB Y/Y Table *(4 of 4)*

Index	Y1	Y2	Index	Y1	Y2	Index	Y1	Y2
71	96	144	158	80	48	245	200	192
72	96	160	159	96	48	246	208	192
73	100	104	160	112	48	247	224	192
74	104	108	161	144	48	248	240	192
75	104	112	162	176	48	249	208	200
76	104	120	163	208	48	250	216	200
77	108	112	164	240	48	251	216	208
78	112	116	165	56	52	252	224	208
79	112	120	166	60	56	253	240	208
80	112	128	167	64	56	254	232	216
81	112	144	168	72	56	255	240	224
82	112	160	169	64	60			
83	112	176	170	68	64			
84	112	208	171	72	64			
85	112	240	172	80	64			
86	120	128	173	96	44			

≡ *D*

C_b/C_r Table Values

C_b/C_r Table Values

Table D-3 lists the default values for the CellB C_b/C_r quantization table.

Table D-3 Default CellB C_b/C_r Table   *(1 of 4)*

Index	U	V	Index	U	V	Index	U	V
0	16	16	87	128	184	174	160	120
1	16	48	88	128	192	175	160	128
2	16	80	89	128	208	176	160	136
3	16	112	90	128	224	177	160	144
4	16	144	91	132	136	178	160	152
5	16	176	92	132	140	179	160	160
6	16	208	93	132	144	180	160	168
7	16	240	94	132	148	181	160	176
8	48	16	95	132	152	182	160	184
9	48	48	96	136	104	183	160	192
10	48	80	97	136	112	184	160	208
11	48	112	98	136	120	185	160	224
12	48	144	99	136	128	186	168	112
13	48	176	100	136	132	187	168	120
14	48	208	101	136	136	188	168	128
15	48	240	102	136	140	189	168	136
16	64	112	103	136	144	190	168	144
17	64	128	104	136	148	191	168	152
18	64	144	105	136	152	192	168	160
19	64	160	106	136	156	193	168	168
20	64	176	107	136	160	194	168	176
21	80	16	108	136	168	195	176	16
22	80	48	109	136	176	196	176	48
23	80	80	110	136	184	197	176	64
24	80	96	111	140	132	198	176	80

Table D-3   Default CellB C$_b$/C$_r$ Table   *(2 of 4)*

Index	U	V	Index	U	V	Index	U	V
25	80	112	112	140	136	199	176	96
26	80	128	113	140	140	200	176	112
27	80	144	114	140	144	201	176	120
28	80	160	115	140	148	202	176	128
29	80	176	116	140	152	203	176	136
30	80	192	117	140	156	204	176	144
31	80	208	118	144	16	205	176	152
32	80	240	119	144	48	206	176	160
33	96	80	120	144	64	207	176	168
34	96	96	121	144	80	208	176	176
35	96	112	122	144	96	209	176	192
36	96	128	123	144	104	210	176	208
37	96	144	124	144	112	211	176	224
38	96	160	125	144	120	212	176	240
39	96	176	126	144	128	213	184	128
40	96	192	127	144	132	214	184	136
41	96	208	128	144	136	215	184	144
42	104	128	129	144	140	216	184	152
43	104	136	130	144	144	217	184	160
44	104	144	131	144	148	218	192	80
45	104	152	132	144	152	219	192	96
46	104	160	133	144	156	220	192	112
47	112	16	134	144	160	221	192	128
48	112	48	135	144	168	222	192	144
49	112	64	136	144	176	223	192	160
50	112	80	137	144	184	224	192	176
51	112	96	138	144	192	225	192	192

Table D-3 Default CellB C_b/C_r Table *(3 of 4)*

Index	U	V	Index	U	V	Index	U	V
52	112	112	139	144	208	226	192	208
53	112	120	140	144	224	227	208	16
54	112	128	141	144	240	228	208	48
55	112	136	142	148	132	229	208	80
56	112	144	143	148	136	230	208	96
57	112	152	144	148	140	231	208	112
58	112	160	145	148	144	232	208	128
59	112	168	146	148	148	233	208	144
60	112	176	147	148	152	234	208	160
61	112	192	148	148	156	235	208	176
62	112	208	149	152	104	236	208	192
63	112	224	150	152	112	237	208	208
64	112	240	151	152	120	238	208	240
65	120	112	152	152	128	239	224	112
66	120	120	153	152	132	240	224	128
67	120	128	154	152	136	241	224	144
68	120	136	155	152	140	242	224	160
69	120	144	156	152	144	243	224	176
70	120	152	157	152	148	244	240	16
71	120	160	158	152	152	245	240	48
72	120	168	159	152	156	246	240	80
73	120	176	160	152	160	247	240	112
74	128	64	161	152	168	248	240	144
75	128	80	162	152	176	249	240	176
76	128	96	163	152	184	250	240	208
77	128	104	164	156	136	251	240	240
78	128	112	165	156	140	252	0	0

Table D-3 Default CellB C$_b$/C$_r$ Table   *(4 of 4)*

Index	U	V	Index	U	V	Index	U	V
79	128	120	166	156	144	253	0	0
80	128	128	167	156	148	254	0	0
81	128	136	168	156	152	255	0	0
82	128	144	169	160	64			
83	128	152	170	160	80			
84	128	160	171	160	96			
85	128	168	172	160	104			
86	128	176	173	160	112			

≡ D

Bibliography E≣

Andrews, H. C. *Computer Techniques in Image Processing.* New York: Academic Press, 1970.

Baxes, G. A. *Digital Image Processing: A Practical Primer.* Englewood Cliffs, N.J.: Prentice-Hall, 1984.

A nonmathematical introduction to image processing.

Castleman, K. R. *Digital Image Processing.* Englewood Cliffs, N.J.: Prentice-Hall, 1979.

Foley, J. D., et al. *Computer Graphics: Principles and Practice.* 2nd ed. Reading, Mass.: Addison-Wesley, 1990.

Discusses dithering, fills, geometric transforms, and color models.

Gonzalez, R. C., and P. Wintz. *Digital Image Processing.* 2nd ed. Reading, Mass.: Addison-Wesley, 1987.

Green, W.B. *Digital Image Processing: A Systems Approach.* New York: Van Nostrand Reinhold, 1983.

Groff, V. *The Power of Color in Design for Desktop Publishing.* Portland: MIS Press, 1990.

≡ *E*

Hall, E. L. *Computer Image Processing and Recognition.* New York: Academic Press, 1979.

International Organization for Standardization and International Electrotechnical Commission. *Information Technology—Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1.5 Mbits/s.* 1992.

The MPEG-1 standard.

International Organization for Standardization and International Electrotechnical Commission. *Information Technology—Digital Compression and Coding of Continuous-Tone Still Images.* 1991.

The JPEG standard.

International Telegraph and Telephone Consultative Committee (CCITT). "Recommendation H.261 - Video Codec for Audiovisual Services at p x 64 Kbits∕s." *Study Group XV - Report R 37.* 1990.

The H.261 standard.

Jain, A. *Fundamentals of Digital Image Processing.* Englewood Cliffs, N.J.: Prentice-Hall, 1989.

Magnenat-Thalmann, N., and D. Thalmann. *Image Synthesis: Theory and Practice.* Tokyo: Springer-Verlag, 1987.

Nye, A. *Xlib Programming Manual.* O'Reilly & Associates, Inc., 1988.

OpenWindows Version 3.0.1 Programmer's Guide. Mountain View, Ca.: SunSoft, 1992.

Discusses colormaps in Sun's implementation of X11.

Pennebaker, W. B., and J. L. Mitchell. *JPEG Still Image Data Compression.* Van Nostrand Reinhold, 1991.

A detailed description of the JPEG still-image compression standard.

PostScript Language Reference Manual. 2nd ed. Reading, Mass.: Addison-Wesley, 1990.

Discusses color models and filtering operations.

Pratt, W. K. *Digital Image Processing.* 2nd ed. New York: Wiley, 1991.

Raster Graphics Handbook. 2nd ed. New York: Van Nostrand Reinhold, 1985.

Contains a good appendix on color models.

Rogers, D. F. *Procedural Elements for Computer Graphics.* New York: McGraw-Hill, 1985.

Provides information about dithering and fills.

Rosenfield, A., and A. C. Kak. *Digital Picture Processing.* 2nd ed. 2 vols. New York: Academic Press, 1982.

Scheifler, R. W., and J. Gettys. *X Window System.* Bedford, Mass.: Digital Press, 1988.

Covers color management in the X Window System.

Thorell, L. G., and W. J. Smith. *Using Computer Color Effectively: An Illustrated Reference.* Englewood Cliffs, N.J.: Prentice Hall, 1990.

Ulichney, R. *Digital Halftoning.* Cambridge, Mass.: MIT Press, 1987.

Contains good discussions of halftoning and dithering.

Wallace, G. K. "The JPEG Still Picture Compression Standard." *Communications of the ACM.* April 1991, p. 31.

A good introduction to JPEG compression.

Wolberg, G. *Digital Image Warping.* Los Alamitos, Ca.: IEEE Computer Society Press, 1990.

≡ E

Glossary

AC coefficient

A **DCT** coefficient that corresponds to nonconstant data in the original image.

active buffer

Represents the buffer that will be affected when an operation uses a double-buffered image.

additive color system

A color model in which colors are built by adding together primary colors. **RGB** is an example.

affine transform

Affine transforms include such operations as scaling, rotation, translation, and shearing. What these operations have in common is that they can change the lengths and angles of lines, but not their parallelism. The XIL library contains a function `xil_affine()` that in one operation can scale, rotate, and translate an image.

atom

A single XIL function. Because of its deferred-execution scheme, the library is sometimes able to find groups of atoms that it can replace with an optimized routine that does the jobs of all the atoms in the group. These optimized routines are called **molecules**.

attribute

A term from object-oriented programming that refers to a characteristic of an object.

band stride

The number of data elements from a pixel in one band of an image to the same pixel in the next band.

Baseline sequential codec

A sequential coder-decoder defined by the **JPEG** standard. It is designed to handle images with 8-bit samples and uses Huffman coding for its entropy coding.

bicubic interpolation

Bicubic **interpolation** is the most time consuming of the XIL library's interpolation methods, but produces the best results. When this type of interpolation is requested, the library calculates the value of a pixel in the destination image by determining the point in the source image to which that pixel maps and then examining the values of the sixteen pixels closest to that point.

bilinear interpolation

When the XIL library uses bilinear (or first-order) **interpolation** to determine the value of an image at noninteger coordinates, it calculates the value by looking at the values of the four pixels surrounding the point of interest and then using a bilinear equation. This type of interpolation yields better results than **nearest neighbor interpolation**, but can itself have an undesirable smoothing effect on an image. To alleviate this problem, you can use **bicubic interpolation**.

Block Truncation Coding

The image compression method on which the **Cell encoding** scheme is based. In Cell compression, a 4-by-4 region of pixels from an image is represented by two colors and a 16-bit mask.

CCIR

The Committée International de la Radio. This treaty organization, part of the International Telecommunications Union (ITU), is responsible for obtaining international agreement on standards for radio and television transmission and the international exchange of programs.

CCIR Recommendation 601

An international standard for digitizing PAL, NTSC, and SECAM analog video.

CCITT

International Telegraph and Telephone Consultative Committee, an international association and standards body composed primarily of representatives from national telephone agencies. The CCITT promulgates telephony standards, such as X.25, the **Group 3** facsimile standard, and the **H.261** videoconferencing standard (p*x*64).

CCITT Group 3

A standard that specifies how facsimile machines must compress and decompress image data. The compression method relies heavily on **run-length encoding**. Runs of white pixels and runs of black pixels are represented with codes from a **Huffman** table.

CCITT Group 4

A standard for compressing document images. This compression technique takes advantage of a characteristic of document images called vertical coherence. This means that transitions from black to white or vice versa generally occur in almost the same place on adjacent scanlines. The Group 4 encoder compresses an image by recording information about the relative locations of these transitions.

Cell

A 4-by-4 block of pixels. (*See also* **Cell encoding**.)

Cell encoding

A video compression algorithm developed by Sun. In Cell encoding, a 4-by-4 region of pixels is represented by two colors and a 16-bit mask that indicates which of the two colors to place at each of the 16 pixel positions. The colors and mask are chosen to preserve the mean and variance of the luminance and the average chrominance for the 4-by-4 block.

Cell decoding takes advantage of the fonting hardware commonly found in bitmapped displays.

CellB encoding

A video-compression algorithm derived from the **Cell** algorithm. As with a Cell compressor, the most fundamental task of a CellB compressor is to encode 4-by-4 cells of pixels in four bytes. The first two bytes of each cell code are a 16-bit mask that indicates which of two colors will represent each pixel in the cell. The third and fourth bytes contain indexes into tables of luminance and chrominance values and define the two colors to be used in encoding the cell.

CellB coders and decoders are intended primarily for use in videoconferencing applications.

chrominance

The portion of a composite signal that carries color information. For example, the C_b or C_r component of a YC_bC_r signal represents part of a pixel's chrominance. (*See also* **luminance**.)

CIF

Common Interchange Format. CIF format images contain YC_bC_r data and are 352 pixels wide and 288 pixels high. Images in this format are one of the two types of images that may be supported by **H.261** codecs.

CIS

Compressed image sequence. The XIL library's compressors store (generally related) compressed images in structures called CIS buffers. The images may represent frames in a movie, pages in a document, and so on.

CMY color model

In the CMY color model, the subtractive primaries cyan, magenta, and yellow are used to filter their complements (red, green, and blue) from white light. You use this color model when working with devices like color printers that put colored ink on paper. For example, to create the color red on a color printer, you put down a mixture of magenta and yellow ink. The magenta filters out its complement green; the yellow filters out its complement blue; and you see only red.

CMYK color model

Similar to the CMY color model, but uses a fourth color: black. Black is used to replace equal amounts of cyan, magenta, and yellow. This color model is used in offset color printing.

codec

A coder-decoder for the compression and decompression of image data.

colormap

A color lookup table that stores a set of **RGB** values. Applications index into the colormap to get the values to drive the red, blue, and green guns of an RGB monitor.

compressed image sequence

See **CIS**.

compression

The process of converting data from its original format to a format that requires fewer bits. Compressed data uses less storage than uncompressed data and can be transmitted over a network more quickly. Some compressors, such as the **CCITT Group 3** compressor, are called **lossless** compressors because they compress data in a way so that a decompressor can regenerate the original data exactly. Other compressors are called **lossy** compressors because they compress data in a way that prevents the original data set from being regenerated exactly.

convolution

An image-processing operation frequently used to sharpen an image, blur an image, or highlight the edges in an image. The operation calculates the values of pixels in the destination image using the values of a neighborhood of pixels in the source image and the values in a special filter called a **convolution kernel**.

convolution kernel

A two-dimensional array of weighted values used in a **convolution** operation. In the XIL library, a kernel is a data structure of data type `XilKernel`. (*See also* **separable convolution kernel**.)

DC coefficient

A **DCT** coefficient that corresponds to the average level of the input image.

DCT

Discrete Cosine Transform. Many encoders, including those that conform to the **JPEG**, **MPEG-1**, and **H.261** standards, perform a DCT on an 8-by-8 block of image data as part of the image-compression process. The DCT converts the video data from the spatial domain to the frequency domain. The DCT takes an 8-by-8 matrix, whose values represent brightness levels at particular *x,y* coordinates, and produces an 8-by-8 matrix whose values represent relative amounts of the 64 spatial frequencies that make up the input data's spectrum.

The DCT provides a basis for compression because most of the frequency levels for a block will be zero or close to zero and do not need to be encoded.

decoder

A program that takes data that has been encoded, or compressed by an **encoder** and decompresses it. A decoder can be implemented in hardware, software, or a combination of both. The decompressed data may or may not match the original data set exactly, depending on how the data was encoded. (*See also* **lossless compression** and **lossy compression**.)

decompression

The restoring of data that has undergone compression to its original state, or to something close to its original state. How closely the decompressed data matches the original data depends on the compression algorithm used. (*See also* **lossless compression** and **lossy compression**.)

depth

Number of bits representing one band of one pixel of an image.

display image

The XIL library treats displays as special images, and operations allow display images to serve as destinations. This strategy enables an operation to draw on a screen directly, without requiring an intermediate copy. In some cases, display images can also serve as source images.

dithering

The XIL library includes a table-lookup function that enables you to convert a single-band image of any data type to a single-band or multiband image of any data type. Dithering can be thought of as an *inverse* lookup operation; that is, dithering enables you to convert a single-band or multiband image of any data type to a single-band image of any data type. The most common dithering operations convert 3-band, 8-bit images to 1-band, 8-bit images and 1-band, 8-bit images to 1-band, 1-bit images.

encoder

An encoder is a program that encodes data for the purpose of achieving data compression. The encoders included with the XIL library are designed to compress images.

entropy coding

Entropy coding is the final step in the compression process in **DCT**-based encoders (such as the **JPEG** baseline sequential encoder). In this step, the encoder compresses quantized **DCT** coefficients using **Huffman** coding. Therefore, values that occur frequently are encoded with fewer bits than are values that occur infrequently.

error diffusion

A technique for removing some of the artifacts produced during the dithering process. In its most common form, Floyd-Steinberg error diffusion, this technique involves (1) determining the amount of error produced in dithering a particular pixel and (2) distributing fractions of that error to the pixels to the right of and below the pixel just dithered.

frame

A single image taken from a movie.

full-motion video

The showing of a series of related digital images at a rate sufficient to give the illusion that objects in the images are moving naturally.

gamma correction

In a linear color space, color levels are equally spaced throughout a gamut. The problem with this type of color space is that level-to-level changes at the low end of the gamut seem greater to the eye than equal changes at the high end. In a gamma-corrected color space, color levels are spaced logarithmically so that level-to-level changes seem consistent throughout the range.

geometric operation

An image-processing operation that changes the size, shape, or orientation of objects in an image. XIL geometric operations include scaling, rotation, and general affine transforms.

H.261

A video compression standard developed by the **CCITT** for use in encoding video to be transferred over an **ISDN** (Integrated Service Digital Network). In this compression method, data is compressed so that the output bit rate is p x 64 Kbits per second, where *p* can range from 1 to 30 depending on the number of ISDN channels used. This standard was developed primarily to support video phone and videoconferencing applications.

handle

An identifier that refers to a unique object. In the XIL library, for example, an identifier of type `XilImage` is a handle to an actual image.

histogram

A collection of information about how frequently certain gray levels or colors appear in an image. The XIL library contains a data structure of type `XilHistogram` to hold this type of information.

Huffman encoding

A method of compressing a given set of data based on the relative frequency of the individual elements: the more often an element occurs, the shorter (in bits) its corresponding code. Huffman encoding is often used to compress text files, with the coding based on letter frequency. Huffman encoding is also used in many **DCT**-based video-compression algorithms.

image

A two-dimensional array of **pixels** that represents an object.

indexed color

See **pseudocolor**.

in-place operations

Operations that require both a source and a destination image but that allow them to be the same image. If a function doesn't support in-place operations, the source and destination images must be different images.

interframe compression

In image sequences, consecutive frames generally have more similarities than differences. These similarities, or shared elements, are sometimes referred to as being temporally redundant. This redundancy is important because it allows groups of individually compressed frames to be compressed further. That is, if five frames in a group look the same in the upper-left corner, that area needs to be encoded only once; then the encoder can simply note that the same data appears in the next four frames. This type of encoding is called interframe compression.

interpolation

A way of calculating a value that falls between other, known values. In image processing, interpolation frequently plays a part in geometric operations such as rotation. After that type of spatial transformation, pixel locations in the output image correspond to noninteger coordinates in the input image. Therefore, the pixel values in the output must be calculated by looking at the values of the pixels surrounding the point of interest in the input. The XIL library supports several types of interpolation, including **nearest neighbor**, **bilinear**, and **bicubic interpolation**.

intraframe compression

The compression that can take place within a single image. Contrast with **interframe compression**.

ISDN

Integrated Service Digital Network. A worldwide public telecommunications network designed to handle many types of data, including voice, text, graphics, and video. The **CCITT** designed the **H.261** video encoder to produce compact bitstreams that can be sent over ISDN lines.

JPEG

Joint Photographic Experts Group. A joint venture of the **CCITT** and ISO that has developed a standard for compressing grayscale or color *still* images. Actually, the standard defines a number of methods for compressing images. Several of these are **lossy** methods based on the Discrete Cosine Transform (**DCT**), but one method is **lossless** and is based on a **predictive coding** technique.

key frame

If the video portion of a movie has been compressed using both **intraframe compression** and **interframe compression**, a decoder cannot decompress the majority of the video frames without referring to preceding—and sometimes succeeding—frames. However, the first frame in the movie, and usually other frames as well, do not have interframe dependencies and can be decoded in isolation. These frames are called key frames. Key frames, besides the one that starts the movie, enable the decoder to decompress other frames without playing the movie from the beginning.

linear remapping

See **rescaling**.

local operation

An image-processing operation in which more than one pixel in a source image is used in calculating the value of a single pixel in the destination image. Generally, the source-image pixels used include the pixel corresponding to the destination pixel being calculated and some set of pixels surrounding that source pixel.

lookup table

In XIL applications, lookup tables are used for general image modification. Each entry in an XIL lookup table contains an index—a value that may appear in the source image—and a value or set of values to be written to the destination image. For each pixel in the source, a table-lookup function finds the pixel's value on the index side of the table and then writes the output value or values for that entry to the corresponding pixel in the destination.

lossless compression

The compression of data in such a way that the original data can be restored exactly. **Huffman encoding** is an example of a lossless compression technique. Some compressors, such as the **JPEG** baseline sequential compressor, combine lossy and lossless compression algorithms. (*See also* **lossy compression**.)

lossy compression

A type of compression that results in the loss of some of the original data. Lossy compression trades the potential loss of some image quality for the opportunity for greater compression. The **JPEG** baseline sequential and **Cell** methods are examples of lossy compression techniques. (*See also* **lossless compression**.)

luminance

The portion of a composite signal that carries brightness information. For example, luminance information is contained in the Y component of a YC_bC_r signal. Video compression techniques take advantage of the fact that the human eye is more sensitive to variations in luminance than it is to variations in color (**chrominance**). Therefore, chrominance values can be compressed (with **lossy** techniques) more than luminance values, resulting in greater overall compression.

macroblock

Both the **H.261** and the **MPEG-1** specifications define a unit within an image called a macroblock. This unit is a 16-by-16 block of pixels. Both H.261 and MPEG-1 encoders can switch from intraframe encoding to interframe encoding on a macroblock basis.

molecule

The XIL term for an optimized routine that performs the work of two or more XIL functions (**atoms).** You don't call a molecule directly. Instead, the XIL runtime system executes a molecule whenever your program calls a sequence of XIL functions that the molecule can replace. This type of substitution is possible because of the library's deferred-execution scheme.

motion compensation

Both **H.261** and **MPEG-1** encoders can perform interframe compression by encoding the differences between the values in a **macroblock** in the picture being encoded and the values in the corresponding macroblock in the preceding image. Sometimes, however, it's desirable to encode the differences between the values in a macroblock in the current image and the values in a macroblock in the preceding image that is slightly offset from the one that corresponds spatially with the macroblock being encoded. This approach is desirable because it enables the encoder to track the movement of objects from image to image and, thus, to encode smaller differences. If an encoder uses this approach, it must of course record the extent of the offset mentioned above. This offset is recorded in a motion vector.

movie

A contiguous series of video frames (and optionally synchronized audio) that are displayed fast enough to provide the illusion of motion. A frame rate of 30 frames/second is a typical target for a smooth-running movie.

MPEG

Moving Picture Experts Group. This group has developed standards for compressing moving pictures and audio data and for synchronizing video and audio datastreams. The XIL library includes a decompression module that can decode MPEG-1 video bitstreams.

The MPEG-1 video-compression standard is similar to the **H.261** standard developed by the **CCITT**, but places less emphasis on low bit rates. By accepting a higher bit rate—up to 1.5 Mbits per second—an MPEG-1 codec is able to recreate very high-quality pictures and to produce a bitstream that is easily editable.

The rate of 1.5 Mbits/s makes the MPEG-1 codec especially viable in applications that read compressed data from CD-ROMs. For example, putting an MPEG-1 bitstream on a CD-ROM is an effective way to distribute movies, business presentations, and training videos.

nearest neighbor interpolation

One of the methods that the XIL library uses to determine the value of an input image at noninteger coordinates. When using this method, the XIL library takes the value it is looking for to be the value of the pixel closest to the point of interest. This type of **interpolation** is the fastest type, but can introduce artifacts in the output image; for example, smooth lines in the input image may show up as jagged lines in the output.
Nearest neighbor interpolation is sometimes called zero-order interpolation.

origin

In the XIL library, each image object has an **attribute** called its origin. This origin is a pair of floating-point numbers that represent *x,y* coordinates in the image. (The point 0.0,0.0 is in the upper-left corner.) When an image-processing operation is performed, the origins of the input and output images are aligned, and the rectangle formed by the intersection of these images serves as an implicit region of interest (**ROI**).

pixel

Picture element. In a raster grid, a pixel is the smallest unit that can be addressed and given a color or intensity.

pixel stride

The distance in data elements to the next pixel of the same band of an image (for example, the distance from the first red pixel to the next red pixel).

plane

One bit of a pixel.

point operation

An image-processing operation in which the value of a point (or **pixel**) in the destination image depends only on the corresponding point in the source image or images. For example, if you want to add two source images to produce a destination image, the value of the pixel in the upper-left corner of the destination image depends solely on the values of the pixels in the upper-left corner of the two source images.

predictive encoding

In predictive encoding, an encoder uses the values of neighboring samples to predict the value of the sample being encoded. The encoder then subtracts this predicted value from the actual value of the sample and encodes the difference. This mode of compression is **lossless**. The XIL library supports a **JPEG** lossless compressor that uses this method.

pseudocolor

The distinction between true color and pseudocolor has to do with the design of a monitor's frame buffer. If the frame buffer uses 8 bits per pixel to store color information, the monitor can display 256 colors simultaneously. What you see on such a monitor is called pseudocolor because the colors that can be shown at any one time are a small subset of the colors the eye can distinguish. If the frame buffer uses 24 bits per pixel to store color information, your monitor can display over 16 million colors (true color).

Pseudocolor is sometimes called indexed color because the values stored in the frame buffer on a pseudocolor system are not the **RGB** values needed to drive the red, green, and blue electron guns in a monitor. Rather, they are indexes into a **colormap**, or color lookup table, which stores 256 sets of RGB values.

QCIF

Quarter Common Interchange Format. QCIF images contain YC_bC_r data and are 176 pixels wide and 144 pixels high (one-fourth the size of **CIF** images). **H.261** codecs are required to support images in this format.

quantization

The technique of scaling down a set of values. In Discrete Cosine Transform (**DCT**)-based encoders, like those that conform to the **JPEG**, **MPEG-1**, and **H.261** standards, quantization is used to ensure that DCT coefficients are represented by the smallest range of numbers needed to produce the desired level of image quality. To bring the coefficients into this range, the quantizer divides each coefficient by the appropriate value from a quantization table and rounds the result to the nearest integer.

rescaling

Sometimes called linear remapping, rescaling involves changing the values in an image using a linear equation. Each value in the image is multiplied by a constant; then a second constant is added to the product. Rescaling is useful for mapping the values in an image from one range to another.

RGB color model

A color model in which colors are built by mixing the three additive primary colors red, green, and blue. In this model, you construct grays by including equal amounts of each primary: (0,0,0) is black and (1,1,1) is white. The RGB color model is closely associated with color CRT monitors because they use this model to produce their colors.

ROI

A region of interest. In the XIL scheme, an ROI is a 1-bit mask that can be specified as an **attribute** of an image. This mask determines what part of a source image can be read or what part of a destination image can modified during an image-processing operation. The ROI for a given operation is the intersection of the ROIs of all the images involved in the operation.

run-length encoding

A compression technique that stores counts of the number of consecutive identical pixels or blocks of pixels in an image.

scanline stride

The distance in data elements to the next pixel on the next horizontal scanline of an image.

separable convolution kernel

Two one-dimensional vectors whose outer product produces a two-dimensional convolutional kernel. (*See also* **convolution kernel**.)

sequential encoding

The encoding of a still image in a single pass through the image data. The sequential encoder first processes a block of pixels in the upper-left corner of the image and then proceeds from left to right and top to bottom until it has processed the entire image.

skip codes

Skip codes are employed during the **interframe compression** process. They instruct the decoder to skip over blocks of pixels in the current frame that are identical to, or very similar to, those in the preceding frame.

slice

MPEG-**1** terminology for a group of consecutive **macroblocks**. Each macroblock in a picture (or image) must be part of a slice. The slice was designed primarily to help a decoder recover from bitstream errors. If a decoder detects an error, one way to recover is to skip to the next slice header.

spatial redundancy

The occurrence of two or more consecutive pixels or blocks of pixels that have the same, or similar, contents. A high level of spatial redundancy in an image can lead to a high rate of compression because only the first pixel or block of pixels in a sequence needs to be encoded fully. A shorthand method can be used to encode succeeding pixels or blocks.

still-image coding

The encoding of a single image.

structuring element

A two-dimensional array of Boolean values used as a parameter to the XIL dilation and erosion functions.

subsampling

A way of mathematically reducing a data set to a subset of its original components. For example, if you have captured a 512-by-512 image, but want to encode it at a resolution of 256-by-256, you can subsample the data before encoding it by discarding every other pixel in both and *x* and *y* directions. On systems that work with YC_bC_r data, it is common to subsample the C_b and C_r **chrominance** information so that each pixel has a unique **luminance** value, but shares chrominance information with one or more nearby pixels.

subtractive color system

In a subtractive color system, primary colors are used as filters to subtract their complements from white light. For example, in the **CMY color model**, the primaries cyan, magenta, and yellow subtract red, green, and blue respectively. That is, if light is reflected from a piece of paper coated with cyan ink, no red is reflected, so you see the color cyan.

temporal redundancy

See **interframe compression**.

thresholding

The XIL library's thresholding function sets all the values (in a band) that fall between a low threshold and a high threshold to a value (called a *map* value) that the programmer specifies.

tiling

One of three image storage configurations XIL supports. The others are whole image and tile stripping. By default, if an image is greater than an XIL-specified threshold, XIL may break up its storage into tiles in memory.

transcoding

The conversion of data in one compressed format to another compressed format. For instance, converting a **JPEG**-compressed image to its **Cell**-encoded counterpart is called transcoding. Transcoding an image usually involves decoding it and then reencoding it.

translation

A geometric operation in which an image is moved up or down and/or left or right.

transposition

A quasi-geometric transformation that involves rotating an image by a multiple of 90 degrees or flipping pixels in an image across a line that passes through the center of the image. The XIL library's transposition function allows for flips across a horizontal or vertical line passing through an image's center and across the main diagonal or antidiagonal.

XIL library

The XIL library is a foundation library for imaging and video support. It provides an implementation of imaging functionality that is common to multiple higher-level interfaces, provides imaging capabilities that are not currently available, and provides a way for ISVs to access low-level and hardware functionality.

YC$_b$C$_r$ color model

When data is stored in YC$_b$C$_r$ format, each pixel is described by a **luminance** value (Y) and two color values. This color model is used in the PAL (European) television format. Also, video analog-to-digital converters often produce data in this format.

Frequently, the color bands of YC$_b$C$_r$ images are subsampled to produce what are called YC$_b$C$_r$ 4:2:2 or YC$_b$C$_r$ 4:1:1 images. In these images, pairs of pixels, or blocks of 4 pixels, share color values; that is, each pixel has its own luminance information, but shares color values with a neighboring pixel or pixels. This subsampling in the color dimension has a very minor impact on image quality.

Index

Z