XIL Device Porting and Extensibility Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Tables

Preface

This document describes the architecture of, and internal interfaces to, the XIL library. It describes the library's C++ classes and discusses the mechanism for acceleration and porting of new hardware. The functionality of the XIL library is discussed in the documents *XIL Programmer's Guide* and *XIL Reference Manual*.

Who Should Use This Book

This book is designed for people porting hardware to use the XIL imaging library, as well as for people who are writing additional device-independent acceleration code for XIL functions.

Before You Read This Book

It is assumed that the reader is familiar with C++ and the ideas of classes and class inheritance in C++. It is further assumed that the reader has studied the *XIL Programmer's Guide* to become familiar with the capabilities of the XIL library.

What's in This Book?

Chapter 1, "Overview" provides an overview of the XIL library and describes the device-independent classes used to implement the library.

Chapter 2, "More on Writing Device Handlers" provides general information about writing XIL device handlers.

Chapter 3, "I/O Devices" describes how to write I/O device handlers and provides an example implementation of an I/O device handler.

Chapter 4, "Compute Devices" describes how to write compute device handlers and provides an example implementation of a compute device handler.

Chapter 5, "Storage Devices" describes how to write storage device handlers and provides an example implementation of a storage device handler.

Chapter 6, "Compression/Decompression" describes how to add a new compression method and compression hardware, and provides an example implementation of a compressor.

Appendix A, "Sample Molecule" provides an example that illustrates a molecule for performing 16-to-8 bit remapping of memory images.

Appendix B, "XIL Atomic Functions" provides the name of the function that must be supplied in the `XILCONFIG` header comment to associate an implemented function with an API call.

Appendix C, "XilOp Object" lists the number of image sources supported by an XIL function and the `XilOp` member functions that must be used to extract the image sources and to extract an XIL function's parameters from the `XilOp` object.

Related Books

XIL Reference Manual

XIL Programmer's Guide

XIL Test Suite User's Guide

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, 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.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

XIL DDK Directory Structure

The default installation directory for the XIL DDK (Driver Developer Kit) is `/opt/SUNWddk/ddk_2.4/xil`. The structure of the XIL DDK directories is described in Figure P-1 and in the sections that follow.

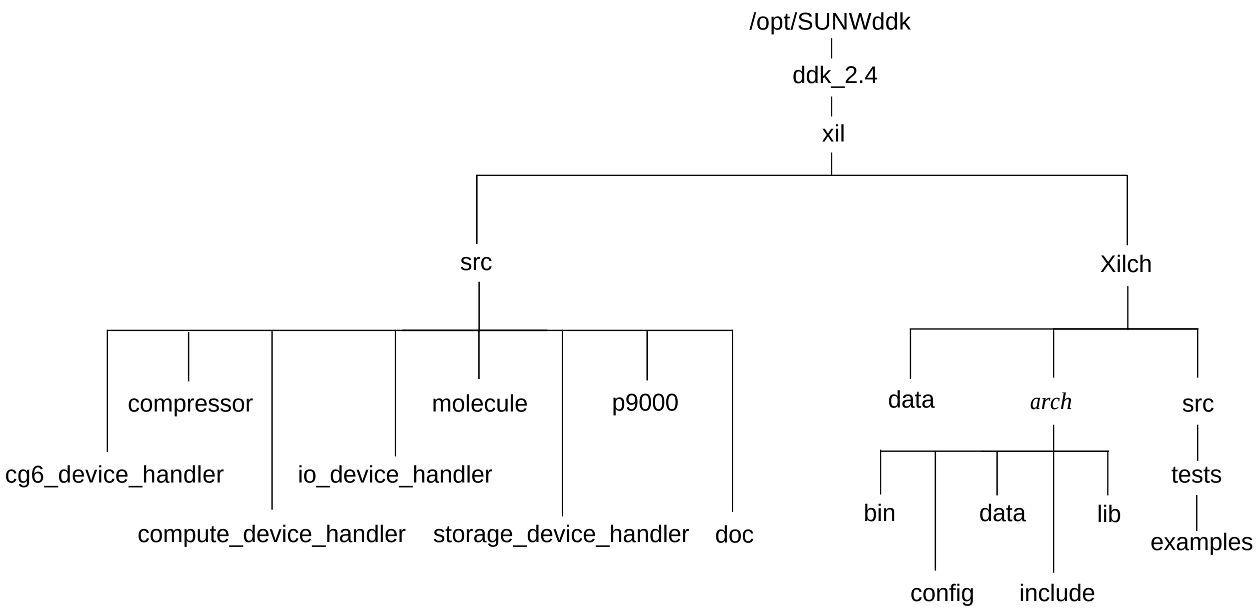

Figure P-1 Directory Structure of XIL DDK Release

src/

The src directory contains seven subdirectories of examples; six of these examples are described in this manual: cg6_device_handler, compressor, compute_device_handler, io_device_handler, molecule, and storage_device_handler.

Note – The directory src/cg6_device_handler contains an example I/O device handler that treats a SPARC GX frame-buffer window as an I/O device. The directory src/p9000, which isn't discussed in this manual, contains an example for an x86-specific module that is based on the SPARC GX example: it treats a p9000 frame-buffer window as an I/O device. The p9000 example isn't discussed in this manual because the p9000 architecture is similar to the CG6 architecture. The p9000 code is included to demonstrate some of the differences you can expect when writing an XIL module for x86.

The src/doc directory contains a source (.po) file used for generating error messages.

Xilch/

The Xilch directory contains the files for the XIL Test Suite, including executables, data files, and examples. The XIL Test Suite is described in the *XIL Test Suite User's Guide*.

Overview 1

Introduction to the XIL Imaging Library

The Solaris™ XIL™ Imaging Library provides a basic set of functions for imaging and video applications. The XIL library is the imaging component of the Solaris Graphics Architecture, a strategy for providing low-level software interfaces known as foundation libraries. Application and API developers can port their code to such foundation libraries. The XIL library is complemented by the XGL™ Graphics Library, which addresses application and API requirements for geometry-based graphics.

Solaris Graphics Architecture

The XIL foundation library is an integral part of the Solaris Graphics Architecture. The Solaris software, using loadable drivers, enables display devices using the Solaris Graphics Architecture to be easily installed and used, without requiring kernel modifications. The Solaris Graphics Architecture, through the XIL, XGL, and OpenWindows™ software, provides a means for third-party hardware and software vendors to develop applications with the knowledge that their investment will see long-term benefits, including access to a range of computing platforms and complete integration into the Solaris environment.

Note – Currently, the GPI interfaces to the library discussed in this book are *uncommitted* interfaces; therefore, they may change in future releases in ways that could require you to change your code. In a release in the near future, the interfaces will be deemed *committed*, and much stricter rules will then apply to interface changes. The API interfaces are *committed*.

Division of Function in the XIL Library

The XIL architecture consists of a high-level application programming interface (API), device-independent *core* code (including the XIL API and GPI layers), which manages the loading and calling of specific device-dependent functions, a graphic porting interface (GPI), which separates device-independent and device-dependent code, and the device-dependent (DD) algorithm implementation. Figure 1-1 illustrates this division of function and shows how these sections relate:

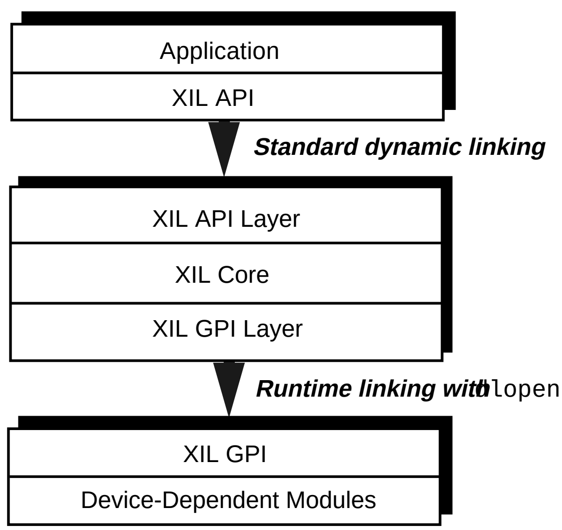

Figure 1-1 The XIL Internal Architecture

This document describes the XIL core (including the XIL API and GPI layers), the graphic porting interface (GPI), and the method needed to supply alternative DD code. In general, porting new hardware to the XIL environment involves providing new implementations of DD modules. The GPI is the interface through which the DD modules are called and is responsible for allowing the creation of new DD implementations without requiring exposure of XIL library source code.

XIL API Layer

The API layer in the XIL library contains the C wrappers on the C++ device-independent classes. It consists of functions that can be categorized in the following way:

- Create and destroy objects
- Set and get object attributes
- Modify image data
- Extract information from an image
- Modify data in non-image objects
- Synchronize operations

The semantics of the functions exposed in the API are described in the *XIL Programmer's Guide* and the *XIL Reference Manual*.

The C++ XIL API level classes are used to implement the API functions described above. These classes provide a device-independent interface to the XIL library imaging functionality and are primarily used to pass information through the GPI to the DD modules. They are listed in Table 1-1 and individually described in the following sections.

Table 1-1 XIL Device-Independent Classes

Class Name	Definition
`XilDebugObject`	The base class from which all other classes derive
`XilGlobalState`	Contains a list of system states and the tree of atomic/molecular operations and their corresponding function pointers
`XilSystemState`	Contains the creation methods for all API classes
`XilObject`	The parent class for all of the XIL API classes
`XilAttribute`	Contains multiple device attributes
`XilInterpolationTable`	Contains an array of 1 x *n* kernels that represent the interpolation filter in either the horizontal or vertical direction

Table 1-1 XIL Device-Independent Classes *(Continued)*

Class Name	Definition
`XilImageType`	Contains an image description without data
`XilImage`	The basic data element for XIL functions
`XilCis`	Contains the compressed image data and compression functions
`XilKernel`	Contains kernel data used in convolution and error diffusion
`XilSel`	A structuring element used in erosion and dilation
`XilDitherMask`	Contains dither matrices for ordered dithering
`XilRoi`	Contains region of interest information for an image
`XilLookup`	Contains data for image conversion and colormap use
`XilColorspace`	Contains information to specify a colorspace
`XilHistogram`	Contains image histogram information
`XilError`	Contains information for reporting errors

Figure 1-2 shows the base part of the XIL library object hierarchy.

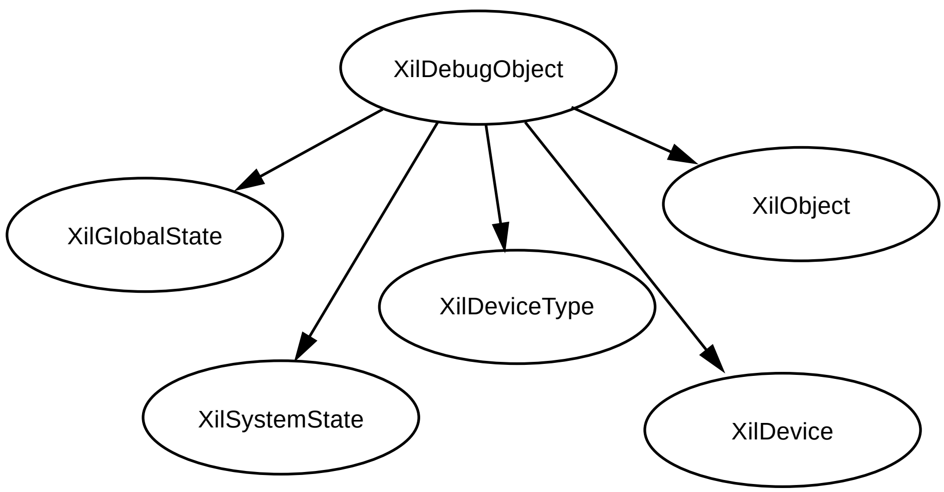

Figure 1-2 XIL Library Object Hierarchy

XIL Base Classes

The `XilDebugObject` *Class*

The `XilDebugObject` class is the base class from which all XIL classes are derived. Its purpose is to allow members to be added to all classes at the same time for debugging purposes. It is currently empty.

Part of the definition of the `XilDebugObject` class is shown below:

Code Example 1-1 Definition of `XilDebugObject` Class

```
class       XilDebugObject {
};
```

The `XilGlobalState` *Class*

This class contains the tree of atomic/molecular operations and their corresponding function pointers. This tree is created initially by the atoms and molecules provided with the XIL library. Each node in the tree contains a list of function pointers. The base of the tree is stored in this class. A new compute handler uses `describeMembers()` to add all of the member functions from the compute handler into the list of function pointers at each node. If the compute handler has a new molecule, then a node is created for that sequence of atoms. See Chapter 4, "Compute Devices," for more detail about compute devices.

A single instance of this class is created when the first `XilSystemState` class is created (during the call to `xil_open()`). Only one `XilGlobalState` is created; it is pointed to by the single global variable `xil_global_state`.

Part of the definition of the XilGlobalState class is shown below:

Code Example 1-2 Definition of XilGlobalState Class

```
class XilGlobalState : public XilDebugObject {
 public:
    XilOpTreeNode* getOpTreeBase();   // Get the base of the tree that holds the
                                      // atomic/molecular operations and their
                                      // corresponding function pointers.
};

extern XilGlobalState* xil_global_state;// The single global variable
```

The actual calls to dynamically load the various modules are done through the `dlopen()` system interface. A private member function of the `XilGlobalState` class implements this and additionally provides code to allow for loading alternate versions of the modules using the `XIL_DEBUG` interface. For information regarding `XIL_DEBUG`, refer to the section "The Development Environment" on page 60 of Chapter 2.

Multiple system states may be created by multiple calls to `xil_open()`. The various system states are created and destroyed through the global state. A singly linked list is used to store the system state objects. The global state destructor uses the list to make sure all the system state objects get destroyed.

The global state also contains the descriptions of atomic and combined image functions, or molecules, in a tree structure. See the description of the `XilOpTreeNode` object in the section "The XilOpTreeNode Class" on page 53 and the discussion on deferred execution and molecules in the section "Deferred Execution" on page 43.

The `XilSystemState` *Class*

The `XilSystemState` class contains all the information for an individual XIL session. The constructor for this class is called by `xil_open()`, which returns the `XilSystemState` handle. Applications must save this handle, since it is a required argument for all object creation routines. At the C++ level, all constructors of the API level XIL objects are members of this class. All of the API objects for an XIL session belong to that session's system state.

Part of the definition of the `XilSystemState` class is shown below:

Code Example 1-3 Definition of `XilSystemState` Class *(1 of 3)*

```
class XilSystemState : public XilDebugObject {

 public:

// notifyError is the main error reporting function. It is called
// by the handler code, normally via the XIL_ERROR macros defined
// in xil/XilError.h
    void notifyError(XilErrorCategory category, char id[],
                    int primary, int line, char* file, XilObject* object);

// API object creation - image creation is overloaded to support
// creation from user data and creation from image type.
// The following GPI functions have the same parameters as their
// corresponding API functions (except the void* parameter of the
// createImage function--in the API the parameter is XilAttribute).

    XilImageType*  createImageType(unsigned int width, unsigned int height,
                              unsigned int nbands, XilDataType datatype);

    XilImage*      createImage (XilImageType* image_type);

    XilImage*      createImage (unsigned int width, unsigned int height,
                              unsigned int nbands, XilDataType datatype);

    XilImage*      createImage (char device[], void* value);

    XilImage*      createImageWindow (Display* display, Window window);

    XilKernel*     createKernel (unsigned int width, unsigned int height,
                              unsigned int keyx, unsigned int keyy,
                              float *data);

    XilDitherMask* createDitherMask(unsigned int width, unsigned int height,
                              unsigned int bands, float* data);

    XilSel*        createSel (unsigned int width, unsigned int height,
                              unsigned int keyx, unsigned int keyy,
                              unsigned int *data);
```

Code Example 1-3 Definition of `XilSystemState` Class *(2 of 3)*

```
    XilLookup*     createLookup (XilDataType input_type,
                              XilDataType output_type,
                              unsigned int nbands, unsigned int num_entries,
                              short offset, void* data);

    XilLookup*     createColorCube(XilDataType input_type,
                              XilDataType output_type,
                              unsigned int nbands, short offset,
                              int multipliers[], unsigned int dimensions[]);

    XilLookup*     createCombinedLookup (XilLookup lookup_list[],
                              unsigned int num_lookups);

    XilHistogram*  createHistogram(unsigned int nbands, unsigned int nbins[],
                              float low_value[], float high_value[]);

    XilColorspace* createColorspace(char* name, unsigned int opcode,
                              unsigned short nbands);

    XilRoi*        createRoi ();

    XilCis*        createCis(char* compression_name);

    XilAttribute*  createAttribute (char* device_name);// the API
                                 // xil_device_create invokes this function

    XilInterpolationTable*createInterpolationTable (unsigned int kernel_size,
                              unsigned int subsamples, float* data);


// Attribute function to control whether the functions for this system
// state are synchronized or allowed to be deferred.
    Xil_boolean getSynchronize();
    void setSynchronize(Xil_boolean onoff);

// Attribute function to control whether image operations are reported
// -1 means check environment variable XIL_DEBUG for “show_action”
// 0 means no show action
// 1 means show action
    int getShowAction();
    void setShowAction(int env_off_on);
```

Code Example 1-3 Definition of `XilSystemState` Class *(3 of 3)*

```
// Entry point to the named object database
    XilObject* getObjectByName(char* name,XilObjectType type);
};
```

The `XilObject` *Class*

`XilObject` is the base class from which all of the API level XIL objects are derived. It contains all the attributes and functions that are generic to those exposed objects. It is an abstract class; no instance of this class is ever created.

Each API level object has a version number, which is updated using the member function `newVersion()` each time that the object is modified. This version number is a 64-bit quantity, and can be returned for any `XilObject` derived class by use of the member function `getVersion()`. A copy of an object returns the same version number. Use of object versions allows intelligent caching of API objects within the implementation.

Part of the definition of the `XilObject` class is shown below:

Code Example 1-4 Definition of `XilObject` Class

```
class XilObject : public XilDebugObject {
 public:
    char* getName ();          // get a copy of the object's name

    void setName (char *name);// set the object's name to the one supplied

    void destroy ();           // destroy objects. Use this rather than "delete"

    XilSystemState* getSystemState();// get a pointer to the system state that
                                      // this object was created under

    // All objects have unique version numbers,
    // which change every time the contents or
    // attributes of the object change. This allows implementations to
    // perform intelligent caching of objects.

    XilVersionNumber getVersion();    // get the version number of this object
```

Code Example 1-4 Definition of `XilObject` Class *(Continued)*

```
    // set the version number of this object to the one supplied. This is
    // used when making identical copies of objects
    void setVersion(XilVersionNumber new_version);// set the version number
                                      // of this object to the one supplied.
                                      // This is used when making identical
                                      // copies

    XilObjectType getObjectType();    // return the XilObjectType of the object

    // each object has a character pointer associated with it for use in
    // error reporting. getErrorString copies “storage_size” characters
    // from the buffer pointed to by this pointer to the “error_storage”
    // buffer. setErrorString sets this pointer to “str”.
    virtual void getErrorString(char* error_storage, int storage_size);
    virtual void setErrorString(char* str);
};
```

The `XilDeviceType` *Class*

The XIL library has the concept of devices—software and hardware—which are represented by the device handler modules. More than one instance of a device may be created. In this case, information that is common to all instances of a device should be held in the device type class. `XilDeviceType` is an abstract class, containing the general information needed at this level. The XIL library subclasses `XilDeviceType` for each of the kinds of devices that are supported.

Note – Each class derived from the `XilDeviceType` class must have its own destructor (even if it takes no action). The base class has a virtual destructor. If the derived class does not supply the destructor, the compiler attempts to in-line the virtual destructor. This results in a warning message and creates out-of-line copies of the destructor, which could result in wasted space.

Part of the definition of the `XilDeviceType` class is shown next:

Code Example 1-5 Definition of `XilDeviceType` Class

```
#include "XilDebugObject.h"

// This class contains all information common to I/O, storage, and compute
// device types. XilDeviceType is an abstract class. The library subclasses one
// instance of an XilDeviceType for each of the kinds of devices it supports.

class      XilDeviceType : public XilDebugObject {
 public:
    virtual void printStatus();       // print any information that the
                                      // device writer thinks might be
                                      // useful for debugging

    virtual void optimizeMemoryUsage(); // routine to request that the
                                      // device free any "optional" system
                                      // memory that it may be using

    virtual ~XilDeviceType();         // destructor

    XilDeviceType(char *dname);       // constructor with name param
};
```

The `XilDevice` *Class*

`XilDevice` is an abstract class that is subclassed for each of the kinds of devices supported by the XIL library (described below). It is similar to `XilDeviceType`, but contains the general information needed for each instance of the device.

Part of the definition of the XilDevice class is shown below:

Code Example 1-6 Definition of XilDevice Class

```
#include "XilDebugObject.h"

// XilDevice
//
// This is the base class for all the various XIL devices
// printStatus() is potentially useful for debugging.

class XilDevice : public XilDebugObject {

    virtual void printStatus();
    virtual ~XilDevice();
};
```

XIL API Level Classes

As described earlier in this chapter, XilObject is the base class for all the API level classes. You cannot instantiate an object from these classes; instead, you must get a copy of or a reference (pointer) to the object. If you get a copy, you are responsible for freeing the allocated data.

Figure 1-3 describes the relationship among the API level objects. The classes are described in the following sections.

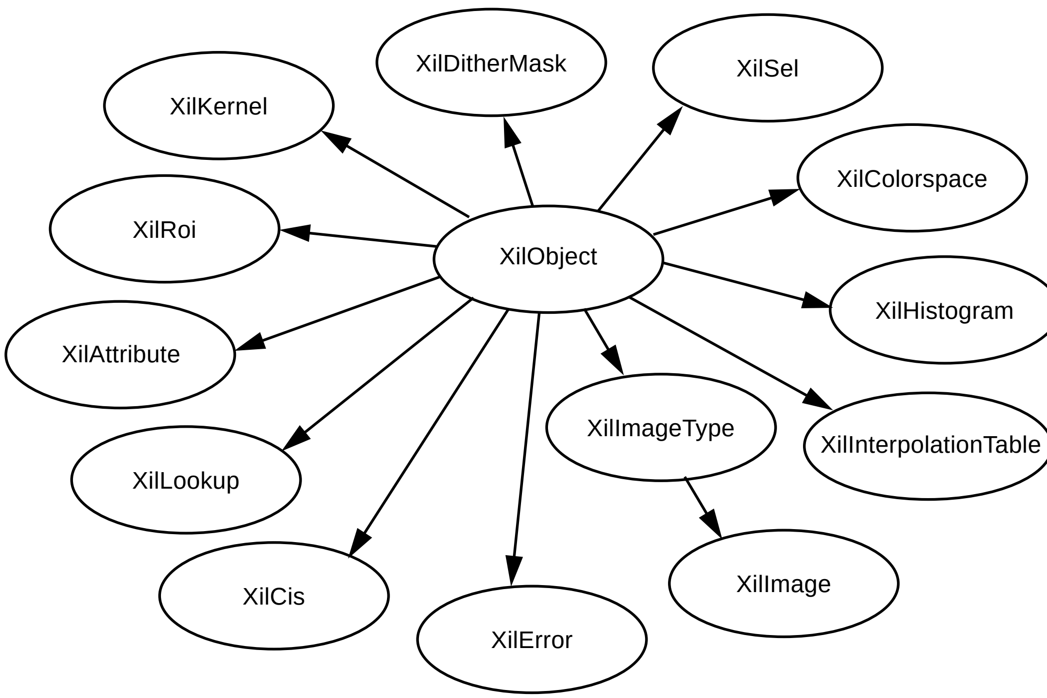

Figure 1-3 API Level Objects

The `XilImageType` *Class*

This class carries information about an image, independent of its associated pixel values. It is used at the API level to return information about the kind of image the application should create to act as a destination from a decompression or device capture, or as a source to a compression or device display. `XilImageType` is subclassed by the XIL library to create the `XilImage` class. There is an API function that creates an image directly from an `XilImageType` object.

Part of the definition of `XilImageType` class is shown next:

Code Example 1-7 Definition of `XilImageType` Class

```
class      XilImageType : public XilObject {
 public:
   unsigned short getWidth();       // the width (extent in x) of the image

   unsigned short getHeight();      // the height (extent in y) of the image

   // overloaded functions to return the x and y image size
   void getSize(unsigned short* width, unsigned short* height);
   void getSize(unsigned int* width, unsigned int* height);

   unsigned short getBands();       // the number of bands in the image

   XilDataType getDataType();       // the data type of the image

   // overloaded functions to return all of the image parameters
   void getInfo(unsigned int* width, unsigned int* height,
                      unsigned int* nbands, XilDataType* datatype);
   void getInfo(unsigned short* width, unsigned short* height,
                      unsigned short* nbands, XilDataType* datatype);

   // functions to set or get a copy of the XilImageType's colorspace
   void setColorspace(XilColorspace* colorspace);
   XilColorspace* getColorspace();
};
```

The `XilImage` *Class*

Derived from the `XilImageType` class, `XilImage` represents an image along with its associated data. The `XilImage` class contains member functions that make up the XIL image functions. It also contains member functions for storage and retrieval of image attributes.

Three of the member functions `getStorage()`, `getMemoryStorage()`, and `requestStorage()` have similar names and deserve an explanation noting their differences.

`getStorage()` returns a pointer to a structure that describes the data storage for the specified storage type. If the current device is not the specified type and cannot emulate the type; then the routine returns `NULL`. `NULL` indicates you must try another storage type. If the routine is successful, it returns a pointer. The structure does not take child image offsets into account, so you must calculate the offsets by using the `offsetX`, `offsetY`, and `offsetBand` fields returned by the `getChildOffsets()` member function. For example, for a byte image that is a child image, you would perform the calculation below to have an accurate pointer to the start of the image data:

```
new_storage.byte.data = storage->byte.data+
storage->byte.scanline_stride*offsetY+
storage->byte.pixel_stride*offsetX+
offsetBand;
```

`getMemoryStorage()` does not allow you to specify the storage type. It uses only the memory storage handler structures defined in `XilMemoryDefines.h`. This routine takes child offsets into account, so there are not extra calculations that you must perform.

`requestStorage()` returns a pointer to a structure that describes the data storage for the specified storage type, but does not force a propagation as `getStorage` does. If the storage has not already been allocated for this image, the routine returns `NULL`. If the requested device does not have the storage or cannot efficiently emulate the requested device, the routine returns `NULL`.

Note that an exported image is required to stay in memory and will not be propagated to another storage type, which may prevent acceleration.

Two member functions enable you to indicate which pixels in the destination image are touched by an operation and to indicate that the XIL core is responsible for freeing the ROI—`setPixelsTouchedRoi()` and `setPixelsTouchedRoi_flag()`.

Operations must intersect the ROI for the source image and destination image to determine the ROI that is written by the operation. This determination is standard procedure and is usually performed by using `XiliGetRoiList()`. For example,

```
(set up for operation)
            .
            .
// get the intersection of source and destination image's ROIs
```

```
intersected_list =
    XiliGetRoiList(&intersected_roi,src1,src2,dest)
            .
            .
(loop over the intersected list of rectangles to perform
operation)
            .
            .
// clean up from operation
intersected_roi->destroy();
intersected_list->destroy();
return(XIL_SUCCESS);
```

For device images, XIL may need the intersected ROI after the operation has completed and the display operation has begun (see Chapter 3, "I/O Devices," for information about device images). Therefore, the intersected ROI should not be destroyed as shown in the code above. Instead, the intersected ROI should be stored (as shown above) and assigned as the "PixelsTouchedRoi" (the pixels that were touched) for the image. Then, a flag should be set that indicates that the XIL core is responsible for freeing the intersected ROI. For example,

```
(set up for operation)
            .
            .
// get the intersection of source and destination image's ROIs
intersected_list =
    XiliGetRoiList(&intersected_roi,dest,src1,src2)

// set up the PixelsTouchedRoi
src1->setPixelsTouchedRoi(intersected_roi);
src1->setPixelsTouchedRoi_flag(TRUE);
            .
            .
(loop over the intersected list of rectangles to perform
operation)
            .
            .
```

```
// clean up from operation
intersected_list->destroy();
return(XIL_SUCCESS);
```

Note – If PixelsTouchedRoi is assigned, the routine *must not* destroy the intersected ROI.

If a routine does not assign the intersected ROI as the PixelsTouchedRoi for the image, the ROI on the display image determines which pixels are touched.

Part of the definition of the `XilImage` class is shown below:

Code Example 1-8 Definition of `XilImage` Class *(1 of 5)*

```
class      XilImage : public XilImageType {
 public:
   // get only attributes (set at create time)
   XilImage *getParent();           // return the pointer to the image's
                                    // parent, NULL if there is no parent.

   void* getMemoryStorage();        // get a pointer to a structure
                                    // that describes the memory
                                    // layout. This function takes child
                                    // image offsets into account.

   void setMemoryStorage(XilMemoryStorage* storage);// set the memory
                                                   // storage of an image

   void* getStorage(char storage_type[]); // get a pointer to a structure
                                    // that describes the data storage.
                                    // This function returns the
                                    // storage information on the parent
                                    // only, but allows the calling process
                                    // to request a particular kind of
                                    // storage.

   XilDeviceInputOutput* getDeviceInputOutput(); // access the input/output
                                    // device, if the image is a device
                                    // image. This will be NULL if it is
                                    // not a device image.
```

Code Example 1-8 Definition of XilImage Class *(2 of 5)*

```
    void getDimensions (unsigned int *x_size,// get the image size parameters
                         unsigned int *y_size,
                         unsigned int *nbands);

    void getChildOffsets(unsigned int *offsetX,// get the child image offsets
                    unsigned int *offsetY,    // if this is a child image.
                    unsigned int *offsetBand);// If this is not a child
                                              // image, the offsets will be 0

    // functions to return whether this image can be read (if it is an
    // output device image, it might not be), or written (if it is an
    // input device image, it probably cannot be).
    Xil_boolean isReadable ();
    Xil_boolean isWriteable ();

    // get/set attributes
    float getOriginX();                   // get the x image origin
    float getOriginY();                   // get the y image origin
    void getOrigin(float* x, float* y);   // get the image origin in floats
    void getOrigin(long* x, long* y);     // get the image origin in longs
    void setOrigin(float x, float y);     // set the image origin

    int getAttribute (char attribute_name[], void**); // get (return a pointer
                                              // to) a user-assigned attribute

    int        setAttribute (char attribute_name[], void*); // set a user-
                                                          // assigned attribute

    int        getDeviceAttribute(char name[], void**); // get (return a
                                              // pointer to) a device-specific
                                              // attribute. This only applies
                                              // to images that are device
                                              // images.

    int        setDeviceAttribute(char name[], void* value); // set a device-
                                              // specific attribute.
                                              // This only applies to images
                                              // that are device images.
```

Code Example 1-8 Definition of `XilImage` Class *(3 of 5)*

```
// functions used by the blackGeneration() function
void setUndercolor (float color);// set the image undercolor value

float getUndercolor ();          // get the image undercolor value

void setBlack (float color);     // set the image black color value

float getBlack ();               // get the image black color value

// ROI functions
XilRoi* getRoi();                // return a copy of the ROI
                                 // assigned to this image. If no ROI is
                                 // assigned, NULL is returned.

XilRoi* getImageSpaceRoi();      // return a pointer to the
                                 // image-origin adjusted, clipped ROI
                                 // assigned to this image. If no ROI was
                                 // initially assigned, a pointer to an
                                 // ROI that encompasses the entire image
                                 // is returned. The calling routine must
                                 // not delete this ROI.

void setRoi(XilRoi* roi);        // set the image’s ROI. An
                                 // internal copy is made of the ROI that
                                 // is passed to this function.

// PIXELS TOUCHED functions
// Functions to set the region of pixels touched by the last routine
// that writes to the image. The flag indicates whether the routine used
// the PTRoi mechanism (This means only new routines will know to set
// the flag. Old routines will default to the old behavior)

void setPixelsTouchedRoi (XilRoi* roi);  // assign the roi as the
                                         // images’s PixelsTouchedRoi

void setPixelsTouchedRoi_flag (Xil_boolean value);// set a flag to
                                         // indicate the image has a valid
                                         // pixelsTouchedRoi
```

Code Example 1-8 Definition of `XilImage` Class *(4 of 5)*

```
// functions pertaining to deferred execution
void sync ();                      // evaluate the contents of
                                   // this image immediately

void toss();                       // indicate to the deferred-
                                   // execution mechanism that we are no
                                   // longer interested in the contents of
                                   // this image.

void setSynchronize (Xil_boolean onoff);// set the synchronization
                                   // of the image. If set to TRUE,
                                   // operations are always executed
                                   // immediately. Setting it to FALSE
                                   // enables deferral of operations
                                   // (and possible optimization)

Xil_boolean getSynchronize ();     // return the current synchronization
                                   // value of the image

// overloaded functions for child image creation
XilImage* createChild (unsigned int xstart, unsigned int ystart,
                       unsigned int width, unsigned int height,
                       unsigned int startband, unsigned int numbands);

XilImage* createChild (unsigned short xstart, unsigned short ystart,
                       unsigned short width, unsigned short height,
                       unsigned short startband, unsigned short numbands);

// overloaded functions for making image copies
XilImage* createCopy (unsigned int xstart, unsigned int ystart,
                      unsigned int width, unsigned int height,
                      unsigned int startband, unsigned int numbands);

XilImage* createCopy (unsigned short xstart, unsigned short ystart,
                      unsigned short width, unsigned short height,
                      unsigned short startband, unsigned short numbands);

// overloaded functions for setting and getting single pixel values
void setPixel(unsigned short x, unsigned short y, float* pixel_value);
void setPixel(unsigned int x, unsigned int y, float* pixel_value);
void getPixel(unsigned short x, unsigned short y, float* pixel_value);
void getPixel(unsigned int x, unsigned int y, float* pixel_value);
```

Code Example 1-8 Definition of `XilImage` Class *(5 of 5)*

```
    // import export stuff
    int export();                     // cause the image data to be moved
                                      // from the library space to
                                      // application space (memory).
                                      // Applications cannot access image
                                      // data that has not been exported.
                                      // The function returns XIL_SUCCESS or
                                      // XIL_FAILURE.

    void import(Xil_boolean change_flag);// cause the image data
                                         // to be moved from application space
                                         // to the XIL library space

    int getExported();                // return one of three possible values:
                                      // 0    if the image is not exported
                                      // 1    if the image is exported
                                      // -1   if the image is not exportable
                                      //      (for example, a device image)

    void* getExportedMemoryStorage();// get a pointer to a structure
                                      // that describes the memory
                                      // layout once the image has been
                                      // exported. This function takes child
                                      // image offsets into account.

    void   setDimensions (unsigned int x_size,// set the image size
                          unsigned int y_size,   // parameters (resize the image)
                          unsigned int nbands);

    void*  requestStorage (char storage_type[]);// get a pointer to a
                                      // to a that describes the data
                                      // storage. This function returns the
                                      // storage information on the parent only
                                      // but allows the calling process to
                                      // to request a particular kind of
                                      // storage. If the request cannot be
                                      // satisfied, NULL is returned.
};
```

The XilKernel Class

The XilKernel class represents a two-dimensional array of floating point values. XilKernel objects are used as parameters in the image convolution and blend functions.

Part of the definition of the XilKernel class is shown below:

Code Example 1-9 Definition of XilKernel Class

```
class      XilKernel : public XilObject {
 public:
   unsigned short getWidth ();   // return the width (x size) of the kernel

   unsigned short getHeight ();  // return the height (y size) of the kernel

   unsigned short getKeyX ();    // return the x key pixel value of the kernel

   unsigned short getKeyY ();    // return the y key pixel value of the kernel

   float *getValue ();           // return a pointer to the actual kernel data

   XilKernel* createCopy();      // return a copy of the kernel
};
```

The XilRoi Class

XilRoi describes an arbitrary region of interest (ROI). Member functions exist to manipulate and logically combine XIL ROIs. The functions of the XilRoi class are all pure virtual but are implemented in derived classes within the XIL library.

Each ROI is made up of a list of rectangles. If the ROI is empty, then the number of rectangles in the list is 0 (zero). An empty ROI may be returned when the intersection of specified ROIs have no overlapping pixels. In the following case, the intersect routine does not return a NULL pointer; instead, it returns an empty ROI.

```
roi_intersected = roi_intersect(roi2);
if (roi_intersected->getNumRects()==0) {
        // code here for empty ROI
}
else {
        // process list of rectangles
}
```

To process a list of rectangles, take the following steps:

1. Get the list of rectangles from a specific ROI by using the `getRectList()` function of the `XilRoi` class.

2. Move through the list of rectangles by using the `next()` function of the `XilRoiList` class.

```
class XilRoiList : public XilObject {
public:
    virtual Xil_boolean next (
        long *x,                    // rectangle left-most x
        long *y,                    // rectangle topmost y
        unsigned int *x_size,       // rectangle size in x
        unsigned int *y_size = 0);// rectangle size in y
```

The `next()` function expects pointers to variables. The function writes information about the rectangle (starting x, starting y, width in x, and height in y) to these variables. The `next()` function returns a status of TRUE when there is a rectangle whose information has been loaded into the given parameters. Each time TRUE is returned, `XilRoiList` updates its internal state, tracking the current “next” rectangle. The `next()` function returns a status of FALSE when no more rectangles are in the list. If the `next()` function encounters an empty ROI, it returns FALSE on the first call.

Typically, routines operate on an intersected ROI, the area intersected by the source(s) and destination ROIs. Source and destination ROIs are intersected in image space, taking into account the original offsets and size of the image to which they are attached. For example,

```
roi = (dst->getImageSpaceRoi())->
       intersect(src->getImageSpaceRoi());
```

If no ROI is active on an image, then the image has a ROI that is the same size as the image. This semantic is handled internally in XIL by the `getImageSpaceRoi()` function. Once the routine has the intersected ROI, the routine loops over the rectangle list (`roi_list`).

```
roi_list = roi->getRectList();
while(roi_list->next(&over_x,&over_y,
    &over_x_size,&over_y_size)) {

    // process the rectangles

}
```

When the routine completes the operation, it must destroy the `roi_list`.

```
roi_list->destroy;
```

Normally, the ROI is stored via the `setPixelsTouchedRoi()` function of the `XilImage` class. See the section “The XilImage Class” on page 14 for more information.

You can use the `XiliGetRoiList()` routine as a shortcut for forming a `roi_list`. This routine intersects up to four images using image space ROIs. It stores in `outroi` the copy of the final intersected ROI and returns a copy of the rectangle list of that ROI. The prototype is shown next:

```
XilRoiList* XiliGetRoiList (
    XilRoi** outroi,
    XilImage* image1,
    XilImage* image2,
    XilImage* image3=NULL,
    XilImage* image4=NULL);
```

Only image pointers that are non-NULL are valid for the calculations of a rectangle list, and the image pointers are searched up to a NULL. You must provide at least the image1 and image2 pointers (normally destination/source1). If there are more sources, then use image3 and image4. For example, if you have a routine that needs an ROI intersection for two source images and a destination image, `XiliGetRoiList()` would look like:

```
roi_list=XiliGetRoiList(&roi,dest,src1,src2);
```

Part of the definition of the `XilRoi` class is shown next. Notice that the member functions in this class are all pure virtual. Subclasses that contain the functionality for each of these virtual functions are provided in the library. The caller of any function that returns a new ROI is responsible for destroying it.

Code Example 1-10 Definition of `XilRoi` Class *(1 of 4)*

```
class      XilRoi : public XilObject {
 public:
   virtual int addImage (XilImage* image) = 0;// add values that are nonzero
                                    // in the passed image to the ROI.
                                    // The image is expected to be of type
                                    // XIL_BIT. Images are converted to
                                    // rectangles and added to the ROI.

   virtual int addRect (long x, long y,     // add the specified rectangle
           long width, long height) = 0;    // to the ROI
```

Code Example 1-10 Definition of `XilRoi` Class *(2 of 4)*

```
virtual int addRegion (Region region) = 0;// add the specified X region
                                          // to the ROI

virtual XilRoi* affine(float* matrix) = 0;// return a new ROI that is the
                                  // result of the affine transformation
                                  // of this ROI with the specified matrix

virtual XilRoi* createCopy () = 0;    // return a new ROI which is a
                                      // copy of this ROI

virtual void dump () = 0;             // print out debugging information
                                      // describing the ROI

virtual XilImage* getAsImage () = 0;  // return an XIL_BIT image which
                                      // represents the ROI. The image
                                      // needs to encompass the entire
                                      // extent of the ROI, with pixels
                                      // within the ROI set to 1,
                                      // pixels outside set to 0.

virtual Region getAsRegion () = 0;    // return an X Region which
                                      // the ROI

virtual XilRoi* intersect (XilRoi* roi) = 0;// return a new ROI that
                                      // is the result of an
                                      // intersection between “this”
                                      // ROI and the passed ROI

virtual XilRoi* intersect (short* cliplist, int orgx, int orgy) = 0;
                                      // return a new ROI that is the
                                      // result of an intersection between
                                      // “this” ROI and the passed cliplist
                                      // (primarily used for DGA cliplists)

// return a new ROI which is the result of an intersection between
// the current (“this”) ROI and all the passed ROIs
virtual XilRoi* intersect (XilRoi* roi1, XilRoi* roi2) = 0;
virtual XilRoi* intersect (XilRoi* roi1, XilRoi* roi2, XilRoi* roi3) =0;
```

Code Example 1-10 Definition of XilRoi Class *(3 of 4)*

```
    virtual XilRoi* rotate (float angle, float xorigin, float yorigin) = 0;
                                         // return a new ROI that is the result
                                         // of a rotation of “this” ROI about
                                         // the specified origin

    virtual XilRoi* scale (float xscale, float yscale, float xorigin,
                              float yorigin) = 0;   // return a new ROI that
                                         // is the result of scaling of “this”
                                         // ROI about the specified origin

    virtual int subtractRect(long x, long y, long width, long height)=0;
                                         // remove the specified rectangle
                                         // from the ROI

    virtual XilRoi* translate(int x, int y)=0;// return a new ROI that is the
                                         // result of a translation of “this”
                                         // ROI by the specified x and y
                                         // amounts

    virtual XilRoi* transpose (XilFlipType fliptype, float xorigin,
                              float yorigin) = 0;   // return a new ROI that
                                         // is the result of transposing
                                         // “this” ROI about the specified
                                         // origin

    virtual XilRoi* unite (XilRoi* roi) = 0;// return a new ROI that is the
                                         // union of “this” ROI with the
                                         // specified ROI

    virtual Xil_boolean pointInRegion (long x, long y) = 0;
                                         // return TRUE or FALSE, depending on
                                         // whether the specified point is
                                         // within or outside the ROI

    virtual int numRects () = 0;         // return the number of rectangles
                                         // it would take to fully specify
                                         // the ROI

    virtual void boundingBox (long* x1, long* x2, long* y1, long* y2) = 0;
                                         // return a rectangle that bounds
                                         // the whole ROI
```

Code Example 1-10 Definition of XilRoi Class *(4 of 4)*

```
    virtual XilRoiList* getRectList () = 0;  // return a list-of-rectangles
                                             // object for this ROI. This allows
                                             // application code to step through
                                             // the rectangles in an ROI using
                                             // using XilRoiList::next()

    // XilRoi object constructor
    XilRoi(XilSystemState* system_state) : XilObject(system_state,XIL_ROI) {};
};
```

The `XilLookup` *Class*

The XilLookup class describes a table of data that is used to interpret image data. It can be used to modify the data, creating an output image by treating each input image pixel as an array index. The lookup table can have multiple output data for each input value; that is, it can convert a single band image into a multiple band image. In the special case of three bands output, it can be thought of as a colormap. The data elements may be extracted from the lookup and placed into an X colormap.

To support the common occurrence of performing a lookup on a color image with different response curves for each band, XilLookup also supports multiband lookups. The XilLookup object can contain a separate data array for each band in the input image. The number of bands in the output image must match the number of bands in the input.

Part of the definition of the XilLookup class is shown below:

Code Example 1-11 Definition of XilLookup Class

```
class      XilLookup : public XilObject {
 public:
    XilDataType getInputDataType (); // return the datatype of the input

    XilDataType getOutputDataType ();// return the datatype of the output

    unsigned int getNumEntries ();    // return the total number of entries
                                      // in the table or returns 0 for a
                                      // combined (multiband) lookup.
```

Code Example 1-11 Definition of XilLookup Class *(Continued)*

```
    unsigned short getNBands ();      // return the number of bands in the
                                      // output

    void* getData ();                 // return a pointer to the actual data

    short getOffset ();               // return the offset that describes
                                      // the input value corresponding to
                                      // the first table value or returns 0
                                      // for a combined (multiband) lookup.

    void setOffset (short);           // set the offset that describes
                                      // the input value corresponding to
                                      // the first table value. Returns 0 if it
                                      // is a multiband lookup.

    void getValues (short start, unsigned int count, void* data);
                                      // copy 'count' data values from the
                                      // LUT starting at the table entry
                                      // position 'start' into buffer 'data'
f
    void setValues (short start, unsigned int count, void* data);
                                      // copy 'count' data values from the
                                      // buffer 'data' into the LUT starting
                                      // at the table entry position 'start'.
                                      // Generates an error if it is a
                                      // multiband lookup.

    Xil_boolean getIsColorCube ();    // return TRUE if the LUT is formatted
                                      // as a colorcube, FALSE otherwise

    Xil_boolean getColorCubeInfo(int multipliers[], unsigned int dimensions[],
                               short* origin);
                                      // return information on the colorcube
                                      // formatting if this LUT is a colorcube

    XilLookup* convert (XilLookup* dst);// calculate and return a copy of
                                      // the LUT that converts
                                      // between the two LUTs "this"
                                      // and dst. The resulting LUT's input
                                      // datatype will be that of the input
                                      // datatype of "this", and its output
                                      // datatype will be that of the input
```

Code Example 1-11 Definition of `XilLookup` Class *(Continued)*

```
                                        // datatype of dst. The LUT‘s offset
                                        // and number of entries are the same
                                        // as those for “this”. Index N of the
                                        // resulting LUT contains the index of
                                        // the nearest color in dst to the color
                                        // at index N in “this”. Nearest color
                                        // is determined by Euclidean distance.
                                        // Source and destination LUTs must have
                                        // the same input datatypes, output
                                        // datatypes, and number of bands.

    XilLookup* createCopy ();           // return a copy of the LUT

    unsigned short getInputNBands ();// return the number of bands in
                                        // the input

    // Each of the following functions returns a list where each index
    // in the list corresponds to a band in the combined (multiband) lookup.

    unsigned int* getEntriesList ();  // return the list of the number of
                                        // entries for each lookup in a combined
                                        // table

    short* getOffsetsList ();           // return the list of the offset for each
                                        // lookup in a combined table

    void** getDataList ();              // return the list of the data for each
                                        // lookup in a combined table

    XilLookup* getBandLookup (unsigned int band_num);// return an XilLookup
                                        // from a combined lookup
};
```

The `XilCis` Class

The `XilCis` (for compressed image sequence) class is the primary object for compression in the XIL library. It contains member functions to allow access to and movement through compressed data. The `XilCis` is created by loading a specified compressor.

Part of the definition of the `XilCis` class is shown below:

Code Example 1-12 Definition of `XilCis` Class *(1 of 4)*

```
class          XilCis : public XilObject {
 public:
   // get only attributes (set at create time or by compressor)

   char*          getCompressor();      // return the name of a compressor

   char*          getCompressionType(); // return the name of the type of
                                        // compressor

   Xil_boolean    getRandomAccess();    // return TRUE if the compressor
                                        // supports random accessing of
                                        // individual frames of the sequence;
                                        // otherwise, returns FALSE

   int            getStartFrame();      // return the index to the first
                                        // compressed image in the CIS

   int            getReadFrame();       // return the index to the current
                                        // read frame

   int            getWriteFrame();      // return the index to the next
                                        // frame that will be written

   XilImageType*  getInputType();       // return the XilImageType that the
                                        // CIS will accept for compression

   XilImageType*  getOutputType();      // return the XilImageType
                                        // produced by a compressor
```

Code Example 1-12 Definition of `XilCis` Class *(2 of 4)*

```
Xil_boolean    getReadInvalid();     // return TRUE if a bitstream error
                                     // occurs during decompression.
                                     // Otherwise, returns FALSE
                                     // indicating that the CIS is valid
                                     // and able to be decompressed.

Xil_boolean    getWriteInvalid();    // return TRUE if a bitstream error
                                     // occurs during compression.
                                     // Otherwise, returns FALSE
                                     // indicating that the CIS is valid
                                     // and compression can continue.

// get functions: these return something about the state of
// the cis (probably attributes)

int            hasData();            // return the number of bytes of
                                     // compressed data the CIS contains

int            numberOfFrames();     // return the number of complete
                                     // frames the CIS contains

Xil_boolean    hasFrame();           // return TRUE if a complete frame
                                     // exists at the read frame position.
                                     // Otherwise, return FALSE.

// get/set attributes

int            getKeepFrames();      // return the value that has been set
                                     // as the maximum number of frames
                                     // that the CIS should keep in a
                                     // compressor buffer

void           setKeepFrames(int k); // set the number of frames before
                                     // the read frame that the CIS should
                                     // keep in a compressor buffer

int            getMaxFrames();       // return the value that has been set
                                     // as the maximum number of
                                     // compressed frames that the CIS
                                     // will buffer at one time
```

Code Example 1-12 Definition of `XilCis` Class *(3 of 4)*

```
void            setMaxFrames(int m);  // set the upper limit on the number
                                      // of compressed frames that the CIS
                                      // should buffer

int             getFramesToCompress();// return the value that has been set
                                      // as the number of frames that are
                                      // compressed if the source image is
                                      // SEQUENTIAL

void            setFramesToCompress(int number_of_frames);
                                      // set the number of frames to
                                      // compress if the image is
                                      // SEQUENTIAL

Xil_boolean     getAutorecover();     // return TRUE if autorecovery
                                      // has been turned ON. Otherwise,
                                      // return FALSE.

void            setAutorecover(Xil_boolean on_off);// set autorecovery ON
                                      // or OFF. The default is OFF (FALSE).
                                      // If autorecovery is ON, recovery is
                                      // attempted after a bitstream error
                                      // occurs.

int             getAttribute(char attribute_name[], void** value);
                                      // return the value of the specified
                                      // compressor attribute

int             setAttribute(char attribute_name[], void* value);
                                      // set a compressor attribute

// compression/decompression functions
// these functions use this image as a destination so they just insert
// an operation
//
void            flush();              // instruct the compressor to
                                      // complete any pending write
                                      // operations
void            sync();
void            reset();
void            seek(int framenumber, int relative_to);
```

Code Example 1-12 Definition of `XilCis` Class *(4 of 4)*

```
    void           attemptRecovery(unsigned int nframes, unsigned int nbytes);
                                      // attempt recovery after an error
                                      // occurs in a CIS

    // this function cannot be deferred because it interacts with
    // non-library data
    void           getErrorString(char* error_storage, int storage_size);

    // get the XilDeviceCompression pointer for this CIS
    XilDeviceCompression* getDeviceCompression(void);

    // This function is used by device compressions that do not have a
    // reliable way to get back to deferred frames. They MUST get back
    // to the current frame, but can use this function to avoid having any
    // frames older than read_frame - 1. It is in the public part instead
    // of the private in order to avoid having to install XilCisPrivate.h
    // in the install point.
    void           flushPriorDecompressOps(int frame_no);
};

#define XIL_CIS_ERROR(category,id,primary,dc, read_invalid, write_invalid) \
do { \
    if (!(dc)->inMolecule()) \
        (dc)->generateError((category), (id), (primary), \
        (read_invalid), (write_invalid), __LINE__, __FILE__); \
} while (0);

#define XIL_CIS_UNCOND_ERROR(category,id,primary,dc,read_invalid,
write_invalid) \
    (dc)->generateError((category), (id), (primary), (read_invalid), \
        (write_invalid), __LINE__, __FILE__);
```

The `XilError` *Class*

This class describes errors in the XIL library. Its member functions allow programs to get information about the error, to retrieve the object that is associated with the error, and to control the error handling routines.

Part of the definition of the XilError class is shown below:

Code Example 1-13 Definition of XilError Class

```
class XilError {

 public:
   char* getString();        // get a string associated with the error.
                             // This function uses the localization functions
                             // bindtextdomain() and dgettext() to parse the
                             // error id string into a localized message

   char* getId();            // get the error id string

   void setId(char error_string[]);// set the error id string

   int getLine();            // get the line number where the error occurred

   void setLine(int line);   // set the line number where the error occurred

   char* getFile();          // get the file in which the error occurred

   void setFile(char* file); // set the file in which the error occurred

   char* getLocation();      // primarily an internal routine to indicate
                             // where and in which file an error occurred

   XilErrorCategory getCategory();// get the error category define; one of:
                             //  XIL_ERROR_SYSTEM
                             //  XIL_ERROR_RESOURCE
                             //  XIL_ERROR_ARITHMETIC
                             //  XIL_ERROR_CIS_DATA
                             //  XIL_ERROR_USER
                             //  XIL_ERROR_CONFIGURATION
                             //  XIL_ERROR_OTHER

   char* getCategoryString();// get the error category as a string
   void setCategory(XilErrorCategory category); // set the error category

   int getPrimary();         // get the type of error (primary or secondary)

   void setPrimary(int primary);// set the type of error

   XilObject* getObject();   // get the object associated with the error
```

Code Example 1-13 Definition of `XilError` Class *(Continued)*

```
    void setObject(XilObject* object);// set the object associated with error

    XilSystemState* getSystemState(); // get the system state associated with
                                      // the error

    void setSystemState (XilSystemState* sysSt);// set the system state
                                                // associated with the error
};

// defines for different ways of reporting errors
#define XIL_ERROR(sysSt,category,id,primary) \
{ \
    XilSystemState* _state=sysSt; \
    _state->notifyError(category,id,primary,__LINE__,__FILE__,
    (XilObject*)NULL); \
}

#define XIL_OBJ_ERROR(sysSt,category,id,primary,object) \
{ \
    XilSystemState* _state=sysSt; \
    _state->notifyError(category,id,primary,__LINE__,__FILE__,object); \
}

#define XIL_OBJ_STR_ERROR(sysSt,category,id,primary,object,str) \
{ \
    XilSystemState* _state=sysSt; \
    object->setErrorString(str); \
    _state->notifyError(category,id,primary,__LINE__,__FILE__,object); \
    object->setErrorString(NULL); \
}
```

The `XilHistogram` Class

The `XilHistogram` class describes a multidimensional histogram. This object can be used to gather statistical information on images.

Part of the definition of the `XilHistogram` class is shown below:

Code Example 1-14 Definition of `XilHistogram` Class

```
class      XilHistogram : public XilObject {
 public:
   unsigned short getNBands ();         // the number of bands in the
                                        // histogram (this is a
                                        // multi-dimensional object)

   void getNBins (unsigned int *nbins); // the number of bins for each band

   unsigned int getNElements ();        // total number of elements in the
                                        // array

   void getLowValue(float *low_value);  // copy the array of floats that
                                        // define the value of the first bin
                                        // for each band to the user-supplied
                                        // array “low_value”

   void getHighValue(float *high_value);// copy the array of floats that
                                        // define the value of the last bin
                                        // for each band to the user-supplied
                                        // array “high_value”

   void getData (unsigned int *data);   // copy the histogram data into the
                                        // user-supplied buffer “data”

   unsigned int *getDataPtr();          // return a pointer to the actual data
};
```

The `XilColorspace` Class

`XilColorspace` describes a color space of an image in such a way that images may be transformed from one color space to another. The XIL Imaging Library supports ten color spaces, which are created at the time of a call to `xil_open()`. Each of the supported color spaces is assigned an opcode. This `opcode` is referenced by the color conversion routines. The correlation between `opcode` and color space is defined in the `cs.h` file and is shown in Table 1-2. In the table, the string that follows the "CS" prefix is the color space name. For more information regarding supported color spaces, see *XIL Programmer's Guide* or the man page for `xil_colorspace_get_by_name()`.

Table 1-2 `Opcodes` and Their Associated Color Spaces

Color Space	**Opcode**
CSrgblinear	1
CSrgb709	2
CSphotoycc	3
CSycc601	4
CSycc709	5
CSylinear	6
CSy601	7
CSy709	8
CScmy	9
CScmyk	10

In the current release of the XIL library, there is no way to create a color space; all supported color spaces are created as standard objects at start-up time and can be retrieved using the `xil_colorspace_get_by_name()` functions in the core. Future releases of the library will enhance this scheme to allow the creation of device-dependent color spaces, to enable device color management.

Part of the definition of the `XilColorspace` class is shown below:

Code Example 1-15 Definition of `XilColorspace` Class

```
class      XilColorspace : public XilObject {

 public:
   unsigned int getOpcode();     // return the opcode associated with
                                 // the XIL supported color space

   unsigned short getNBands();   // return the number of bands in the
                                 // colorspace

   XilColorspace* createCopy();  // return a copy of the colorspace
};
```

The `XilSel` *Class*

The `XilSel` class describes a structuring element, which is a two-dimensional description of a pixel neighborhood. In the XIL library, the structuring element is described with a two-dimensional boolean (integer) array, with pixels in a neighborhood having true values in the array, and pixels excluded from the neighborhood having false values. Structuring elements are currently used as parameters to the `xil_erode()` and `xil_dilate()` functions.

Part of the definition of the `XilSel` class is shown below:

Code Example 1-16 Definition of `XilSel` Class

```
class       XilSel : public XilObject {
 public:
    unsigned short getWidth ();   // return the width (x size) of the sel

    unsigned short getHeight ();  // return the height (y size) of the sel

    unsigned short getKeyX ();    // return the x key pixel value of the sel

    unsigned short getKeyY ();    // return the y key pixel value of the sel

    unsigned int *getValue ();    // return a pointer to the actual sel data

    XilSel* createCopy();         // return a copy of the sel
};
```

The `XilDitherMask` *Class*

In the simplest case, the dither mask is a two-dimensional array of values that determines how the noise added during the dither process is spread across the image. In the XIL library, the dither mask can have multiple bands, each band with its own matrix. This allows noise to be spread differently for each channel of a true-color image, which can enhance the result of the dither operation. For dithering of multiband images, the number of bands in the dither mask must match the number of bands in the source image.

Part of the definition of the `XilDitherMask` class is shown below:

Code Example 1-17 Definition of `XilDitherMask` Class

```
class      XilDitherMask : public XilObject {
 public:
    unsigned short getWidth ();   // return the width (x size) of the dither
                                  // mask

    unsigned short getHeight ();  // return the height (y size) of the dither
                                  // mask

    unsigned short getBands ();   // return the number of bands of the dither
                                  // mask

    float *getValue ();           // return a pointer to the actual dither
                                  // mask data

    XilDitherMask* createCopy();  // return a copy of the dither mask
};
```

The `XilAttribute` *Class*

The `XilAttribute` class describes the attribute/value pairs of a device. The member functions of this class enable you to access and set a device's attributes.

An `XilAttribute` object can be used to set multiple device attributes simultaneously. This is important when device images are created and when the setting of device attributes incurs substantial overhead.

When you use this object to create a device, you should set only attributes the device understands. If the device does not recognize an attribute that you have set through the `XilAttribute` object, an error is generated. You should set default values for a device's attributes based on the list of attribute/value pairs returned by the `XilAttribute` object.

The definition of the `XilAttribute` class is shown next:

Code Example 1-18 Definition of `XilAttribute` Class

```
// Data structure for the attribute-value pairs

typedef struct __XilKeyValue Pairs {
    char* key;                // store the attribute name
    void* value;              // store the attribute value
} XilKeyValuePairs;

class  XilAttribute : public XilObject {

public:
    // return a pointer to a list of attribute-value pairs that exist for
    // this XilAttribute object. The list is of length “list_length”

    XilKeyValuePairs** getAttributes (unsigned int* list_length);

    // assign an attribute-value pair of this XilAttribute object
    // if the attribute name has already been set, then the specified
    // attribute_value will replace the previous value

    void setValue (char* attribute_name, void* attribute_value);
};
```

The `XilInterpolationTable` *Class*

The `XilInterpolationTable` class supports general interpolation. See *XIL Programmer’s Guide* for a discussion about general interpolation. This class describes an array of 1 x *n* kernels. The array represents the interpolation filter in either the horizontal or vertical direction. The member functions of this class enable you to access the data in an `XilInterpolationTable` object.

The definition of the `XilInterpolationTable` class is shown next.

Code Example 1-19 Definition of `XilInterpolationTable` Class

```
class   XilInterpolationTable : public XilObject {
public:
    unsigned int getKernelSize ();       // return the size of the
                                         // interpolation filter

    unsigned int getSubsamples ();       // return the number of subsamples
                                         // kernel entry

    float* getData ();                   // return pointer to the table data

    XilInterpolationTable* createCopy ();// return a copy of the table
};
```

XIL Core Layer

The Core layer in the XIL library manages the dynamic loading of device handlers, deferred execution, and operation scheduling.

Deferred Execution

The primary problem in achieving adequate performance in an imaging library comes from the way in which the units (or *atoms*) of functionality are arbitrarily combined to perform useful work. The typical imaging case is much more general than, for example, XGL with its well-defined processing pipeline. This has tended to limit the usefulness of general imaging libraries, since any reasonable division of imaging functionality into atoms renders the performance of many applications substandard. The result is that useful libraries tend to be closely tailored to applications and vertical markets.

The use of multiple passes of atoms impedes performance in at least three ways:

- Multiple passes through an image cause the entire image to be paged into memory multiple times. Since in many cases the images are large compared with available physical memory, and the application is often working with multiple images simultaneously, this significantly impairs performance.

Often a pixel operation can be performed in a single CPU clock cycle, so the time spent getting to the data far outweighs the time needed for the operation. Imaging is often a worst case of I/O bound processing.

- Many combinations of atoms can be performed in a single logical step with little penalty. For example, in the case of a rotation followed by a zoom, the backward-mapped algorithms often used to perform the rotation can perform both operations in nearly the same time as the rotation alone.
- The application must often create temporary images to hold intermediate results. Such intermediate images are not needed in customized code and may be avoided if the operations can be combined.

The XIL Library Method

There are several methods that can address these problems. In the XIL library, we have chosen to implement deferred execution and multiple atomic operation replacement. In this approach, the core-layer code keeps track of image dependencies and causes the operations to occur as late as possible. This enables significant performance improvements as described below.

In the library, atomic functions are, by default, deferred as long as possible. To implement this, the API level function creates an instance of the `XilOp` class, adds the API parameters to the `XilOp`, and then places the operation on a tree-like structure that holds deferred operation information. `void` is then returned to the calling routine (the C binding in this case).

The deferred execution data is stored as a directed acyclic graph (DAG), where the nodes are the instances of the `XilOp` class described above. The fact that a destination function depends on its sources is stored, along with the operation and parameters necessary to produce the destination image once the sources are produced. As image results are needed, the parts of the graph that hold that information are evaluated. Their dependent images are generated by performing the operations that have been stored.

Several actions can cause the evaluation of a subgraph:

- The reuse of an image on which other images depend
- The use of a destination image that has the member `synchronized` set
- A call to `xil_set_synchronize()` that turns on synchronization for an image, or for another image that depends on that image.

The DAG is disassembled upon a call to `xil_close()`.

Graph Evaluation and Molecules

When the graph is evaluated, each node's *op* (the operation used to produce the node's destination image) is available and could be used to index into a list of function pointers. In fact, the library does something a little more general than this, and thus gains the ability to accelerate combined operations.

The XIL library stores its function table as an array of trees, each tree having one of the atomic functions as its base. Branches exist from the base node describing each composite operation (*molecule*) that exists. This structure is built from the description of the contents of each compute device handler (described below). As the core code looks at the DAG, it attempts to match the longest sequence of atoms in the DAG to the function table. If the needed molecule is available, it is called; otherwise, the sequence of functions is checked again, leaving off the last function, which is performed atomically.

Each node on the function tree is a list of possible functions, usually using different compute devices. The core code calls the first function, which is assumed to be the optimal (accelerated) one. The accelerated function is allowed to fail gracefully, in which case the second function in the list is called. Typically, the last function in the list is the unaccelerated *memory* port, which is guaranteed to work for all cases. This construction allows an IHV to accelerate a function for only a subset of the input parameters. For example, the code supporting an accelerator that only scales images up can fail gracefully (and cause the memory function to be called) if the scale factors it pulls off the DAG are less than unity. The mechanism for inserting a function into the table is described later in this document.

The core code does not require porting.

Device porters can accelerate either atomic functions or molecules.

Some Considerations

The time needed to determine the sequence of operations from the DAG and choose the appropriate function from the table appears to be trivial compared to typical image operations.

Not all operations can be deferred. An example of this is the `xil_extrema()` function, which supplies the maximum and minimum image values. The library makes no effort to hide the values returned in an opaque structure, the contents of which could be deferred. Thus, the use of `xil_extrema()` causes an evaluation of the source image. In general, only the functions that have as their destination an `XilImage` or `XilCis` object (or create those objects) can be deferred. The complete list of the rules for deferred execution is as follows:

1. Functions that return information based on values in the current image cannot be deferred. These functions are:

   ```
   xil_choose_colormap()
   xil_compress_colormap()
   xil_extrema()
   xil_generate_colormap()
   xil_histogram()
   xil_squeeze_range()
   ```

2. Functions that have nonstandard ROI, origin, or size behavior cannot be deferred. These functions are:

   ```
   xil_affine()
   xil_paint()
   xil_rotate()
   xil_scale()*
   xil_subsample_adaptive()
   xil_subsample_binary_to_gray()
   xil_tablewarp()
   xil_tablewarp_horizontal()
   xil_tablewarp_vertical()
   xil_translate()
   xil_transpose()*
   ```

   * These operations may be deferred under special circumstances. See *XIL Programmer's Guide.*

3. General rules that apply to the other XIL functions are as follows:

   a. The source and destination images must have the same ROI.

   b. The source and destination images must have the same origins.**

   c. The source and destination images must have the same width (xsize) and height (ysize).**

d. The source images cannot have the same parent as the destination image.

** The `xil_copy_pattern()` function is exempt from this rule.

We do not envision a large number of molecules in a typical release. In particular, `display(zoom(decompress()))` molecules have proven to be advantageous. Other display pipelines (`display(zoom())`, `display(dither(zoom()))`, etc.) will prove useful. It is possible, however, for a third party to add molecules that particularly benefit its vertical market, without requiring that other software running on the XIL library be modified.

One goal of deferred execution is that the application need never know when functions are actually performed. Asynchronous error reporting allows this to be the case in general. However, some cases are impossible to hide. Consider the case of a frame grabber used as a source to an operation that is done in response to an external signal. In normal operation, the actual grab would be postponed until the dependency tree was evaluated, possibly several steps further in the program. A possible resolution to this is to make the destination of the grab operation synchronized. This causes the grab to occur when the function call is made, but precludes any optimization of the grab function. In the end, the application must choose whether the operation should be deferred or not, and when the synchronization should occur. With the general rule “no optimization through synchronization,” the application writer can judge an appropriate place to synchronize.

Molecules must behave semantically like the sequence of atomic operations, and produce the same (or nearly the same) results as calling the individual atomic functions. A molecule cannot have a greater precision than the atomic functions that the molecule contains.

Unusual Effects of Deferred Execution

One effect of deferred execution is that in some cases source code may not accurately reflect the actual operations done. Consider the following case, where `im2` is not set to be synchronous, but `display_image` is set to be synchronous:

```
for (i=0; i<N; i++) {
    a[0] = i;
    xil_add_const(im1, a, im2);
}
xil_copy(im2, display_image);
```

In the XIL library, only one add (the last one) is done as a result of this code, since the earlier results are obscured by the later ones. If the final copy were not called, no evaluation of the add would take place at all. In normal code, such cases rarely arise, but one must be careful in benchmarking the library. This is not unlike the situation that occurs with optimizing compilers.

Consider another case where only the final decompress is executed.

```
while (xil_cis_has_frame(cis)) {
    xil_decompress(cis,im2);
}

xil_copy(im2,display_image);
```

Each call to `xil_decompress()` schedules a frame from `cis` to be decompressed into image `im2`. This destination image is not used until the decompress loop is exited. The last decompressed frame is copied to a display image; this is the only operation that is evaluated.

Core Layer Classes

There are a few classes that are defined as part of the core layer. The class `XilOp` is created, and its members are set, in the API layer. It is used, but not modified, in the DD code.

Table 1-3 XIL Core Level Classes

Class Name	Definition
`XilOp`	The class that holds the information required to store the operation on the DAG
`XilOpTreeNode`	Defines the tree node for deferred execution

The `XilOp` Class

The class `XilOp` contains the information that is stored in the DAG representing a specific XIL operation. The source and destination images of atomic functions must be accessible to the operation. These images are stored in the `XilOp` object. Atomic operations may have up to three source images, depending on the function that performs the operation. For example,

```
xil_copy (source1, dest);
xil_multiply (source1, source2, dest);
xil_blend (source1, source2, dest, source3);
```

You can extract the source and destination images of an operation using the following `XilOp` member functions:

- `getSrc1()`, get pointer to source image 1
- `getSrc2()`, get pointer to source image 2
- `getSrc3()`, get pointer to source image 3
- `getDst()`, get pointer to destination image

In the case of `xil_compress`, the source is an image and the destination is a CIS. Therefore, you would use `getDstCis()` instead of `getDst()` to get the pointer to the destination CIS. Likewise for `xil_decompress`, the destination is an image and the source is a CIS. Therefore, you would use `getSrcCis()` instead of `getSrc1()`to get the pointer to the source CIS.

The parameters of XIL functions also are stored in the XilOp object. You can extract them by using the following XilOp member functions:

- getLongParam()
- getPtrParam()
- getFloatParam()
- getObjParam()

The number of image sources supported by an XIL operation and the XilOp member functions that you must use to extract the image sources and to extract an XIL function's parameters from the XilOp object are listed in Appendix C, "XilOp Object."

The arguments to each element (atomic or molecular) in a compute device handler are op and op_count. For example,

```
int XilDeviceComputeTypeMemory::Add8(
    XilOp*  op,         // a pointer into the DAG
    int op_count);      // the number of combined operations to be
                        // performed
```

For an atomic operation, op_count is always 1.

Let's look at an example of extracting from the XilOp object the source image, the destination image, and the parameters for the function xil_fill(). This function has one source image, a destination image, and four parameters.

```
source = op->getSrc1();
dest = op->getDst();
xseed = op->getFloatParam(1);     // get float value xseed
yseed = op->getFloatParam(2);     // get float value yseed
boundary = (float *)(op->getPtrParam(3));// get &boundary[0]
fill_color = (float *)(op->getPtrParam(4));// get &fill_color[0]
```

A molecule is a chain of atomic operations. For a molecule, the chain must be followed properly to extract in a logical order the parameters and images from the `XilOp` object. For a molecule, the `op_count` parameter specifies how many atomic operations exist along the chain for the molecule. This is necessary because each molecule might contain a different number of operations.

The `op` passed to the routine is the `op` associated with the last operation in the chain (the operation that writes its output to a destination image). The molecule must extract the destination image and the function's parameters from the `XilOp` object. Then, to move up the chain of operations, you can use the following functions:

- `getOp1()`, get pointer to `op` that has source1 as its destination
- `getOp2()`, get pointer to `op` that has source2 as its destination
- `getOp3()`, get pointer to `op` that has source3 as its destination

For more information about molecules and a chain of operations, see the section "Adding a New Molecule" on page 130 of Chapter 4.

Part of the definition of the `XilOp` class is shown below:

Code Example 1-20 Definition of the `XilOp` Class *(1 of 3)*

```
union Xil_param {             // union structure for op parameters
    long l;
    float f;
    void *p;
    XilObject* o;
};

class XilOp {
 public:
    XilImage *getSrc1 ();     // get the first input image
    XilImage *getSrc2 ();     // get the second input image
    XilImage *getSrc3 ();     // get the third input image
    XilImage *getDst ();      // get the destination image
    XilCis *getSrcCis ();     // get a pointer to the source cis
    XilCis *getDstCis ();     // get a pointer to the destination cis
    XilOp *getOp1();          // get the first operation
    XilOp *getOp2();          // get the second operation
    XilOp *getOp3();          // get the third operation
    unsigned int getOp ();    // get the op number
```

Code Example 1-20 Definition of the `XilOp` Class *(2 of 3)*

Code Example 1-20 Definition of the XilOp Class *(3 of 3)*

```
    // NOTE: the parameters must be fetched in the format that they were stored
    long getLongParam (int n);        // get the nth parameter as a long
    float getFloatParam (int n);      // get the nth parameter as a float
    void *getPtrParam (int n);        // get the nth parameter as a pointer
    XilObject* getObjParam(int n);    // get the nth parameter as an XilObject

    // function to cause the op to be executed
    void flush();
};
```

The `XilOpTreeNode` *Class*

The `XilOpTreeNode` class contains the data structure and member functions that store the descriptions of compute device capabilities. The use of the member functions of this class is handled automatically by the creation of the `describeMembers.cc` routine generated by the `xilcompdesc` program. See page 128 (step 3) for a discussion of `xilcompdesc`. The core code adds new functions to the tree using the `addFunction()` member when it loads various computation modules. In general, the device-dependent code should not need to access this class explicitly.

Each `XilOpTreeNode` contains both a list of function pointers that implement the (possibly combined) operation that the node represents and a pointer to other `XilOpTreeNode` objects that perform more combined operations. So, for example, the `XilOpTreeNode` for "multiply" will point to the list of function pointers that perform the atomic "multiply" operation, plus potentially a set of pointer to other `XilOpTreeNode` objects that can perform a multiply followed by an add, a multiply followed by another multiply, and so on.

Part of the definition of the `XilOpTreeNode` class is shown below:

Code Example 1-21 Definition of the `XilOpTreeNode` Class

```
class XilOpTreeNode {
 public:
   // Each of the following three functions returns a pointer to the
   // XilOpTreeNode that contains the list of function pointers that
   // perform the operation described by the ‘operator_name’ character
   // string. Combined operation nodes are accessed by successive branches.
   XilOpTreeNode* branch(char operator_name[]);
   XilOpTreeNode* branch2(char operator_name[]);
   XilOpTreeNode* branch3(char operator_name[]);

   int addFunction(XilDeviceComputeType* compute_type,// add a new function
               MemberFuncPtr member_func);   // to the current XilOpTreeNode
                                             // that will implement the
                                             // operation that this node
                                             // describes

   int addMarker(void);      // for multibranch molecules, the end points of
                             // each branch other than the right-most branch
                             // terminates in a marker instead of a function

   Xil_boolean removeFunctions(XilDeviceComputeType* compute_type);
                             // remove all the functions that have been
                             // inserted in the tree for a particular compute
                             // device. This provided so that the partial
                             // loading of a compute device does not happen.
};
```

XIL GPI Layer

The GPI (Graphics Porting Interface) layer is the interface for device-dependent code. In general, porting a device to the XIL library requires subclassing one or more of the base device classes defined below, and then configuring the resulting object files so that they can be loaded at run-time by the library. In addition to enabling third parties to port hardware, the functions and device access in the standard XIL release are provided through this interface as well.

In the XIL library, there are classes that define four types of devices:

- I/O Devices
- Compute Devices
- Storage Devices
- Compression Devices

These devices are represented by the GPI layer classes, which are discussed individually in Chapters 3 through 6:

Table 1-4 XIL GPI Level Classes

Class Name	Definition
`XilDeviceType`	The base class for the device type object
`XilDevice`	The base class for the device object
`XilDeviceComputeType`	The abstract class for compute devices
`XilDeviceInputOutputType`	The abstract class for I/O devices
`XilDeviceInputOutput`	The device-specific base class for I/O
`XilDeviceStorageType`	The abstract class for storage devices
`XilDeviceStorage`	The device-specific base class for storage
`XilDeviceCompressionType`	The abstract class for compression devices
`XilDeviceCompression`	The device-specific base class for compression

I/O Devices

I/O devices include any devices that can produce or display images, such as frame grabbers, image files, and displays. Configured I/O devices appear as "device images" to XIL applications, and may be used as sources and destinations for all XIL imaging operations. These devices are described in Chapter 3, "I/O Devices."

Compute Devices

Compute handlers contain the device-dependent implementation of one or more atoms or molecules. For example, a compute device might implement the geometric operators accelerated by an add-on card, or might provide a combination of frequently used functions in the form of a molecule. A compute device may be hardware specific, or may be a software-only implementation of a superior algorithm. Compute handlers are loaded during the first call to `xil_open()`. These handlers are described in e."

Storage Devices

Storage devices allow images to reside in other places besides host CPU memory. Such a device is typically associated with a compute device, allowing an accelerator to take advantage of image data remaining local to the accelerator during sequential function calls.

The handlers for storage devices are responsible for allocating, deallocating, and describing the data format of the storage on their devices. They are also responsible for data conversion between storage devices. In addition, it is useful to have the storage handler perform single-pixel access for `xil_get_pixel()` and `xil_set_pixel()` to avoid having to convert image data in those cases.

Typically, a compute device handler will cause the storage device handler for the device to be loaded when it first tries to create an image on the device. The CPU memory storage handler is loaded at the time of the first image creation. Storage devices are discussed in detail in Chapter 5, "Storage Devices."

Compression Devices

Compression devices contain most of the utility functions for implementing a method of compression and decompression, even though the actual compress and decompress functions are provided in an associated compute device handler. The compression device performs buffer management and implements the semantics of the `XilCis` object. A compression device for a specified compressor is loaded when `xil_cis_create()` is called. Compression devices are discussed in Chapter 6, "Compression / Decompression."

More on Writing Device Handlers 2

How XIL Device Handlers Work

Each type of device in the XIL library handles a different aspect of imaging device dependence. The inner workings of each type of device are detailed in the following chapters along with an example of each device handler. However, the overall concept behind providing a device handler is similar among different kinds of devices.

To implement a specific device, you must define a derived class from the appropriate `XilDeviceType` class that represents the device. Only one derived class can exist for each device, and therefore only one for each handler. The purpose of the derived class is to:

- initialize the device
- create the derived `XilDevice` class

Note – If you are writing a device handler, be aware that not all XIL application programs call `xil_close()` before exiting. Therefore you should make sure, if possible, that your device handler frees all system resources if an application dies abnormally.

As an example, consider the case of an I/O device called `camera`, which represents the combination of a frame grabber and camera (as shown in Figure 2-1). This example demonstrates the flow of creating an XIL handler, as follows:

1. The subclass `XilDeviceInputOutputTypeCamera` is created by a call to the global function `XilCreateInputOutputType()`, which must exist in the loadable library that contains the handler.
2. The `XilCreateInputOutputTypeCamera` subclass initializes the frame grabber, and holds all the global information that is shared among different instances of the actual device. The `XilCreateInputOutputType()` function is called when the handler is loaded. In this example of an I/O device, this happens the first time the API function `xil_create_from_device()` is called with the `Camera` handler name as the device-name parameter.
3. After initializing, the XIL core code calls code in `XilDeviceInputOutputTypeCamera` that creates an instance of the derived class `XilDeviceInputOutputCamera`. This class contains all the code needed to perform the image acquisition. The second time the application calls `xil_create_from_device()` with the same device name, the second instance of `XilDeviceInputOutputCamera` is created. To the application, this appears as a second device image. The two device images can exist in sequence or simultaneously.

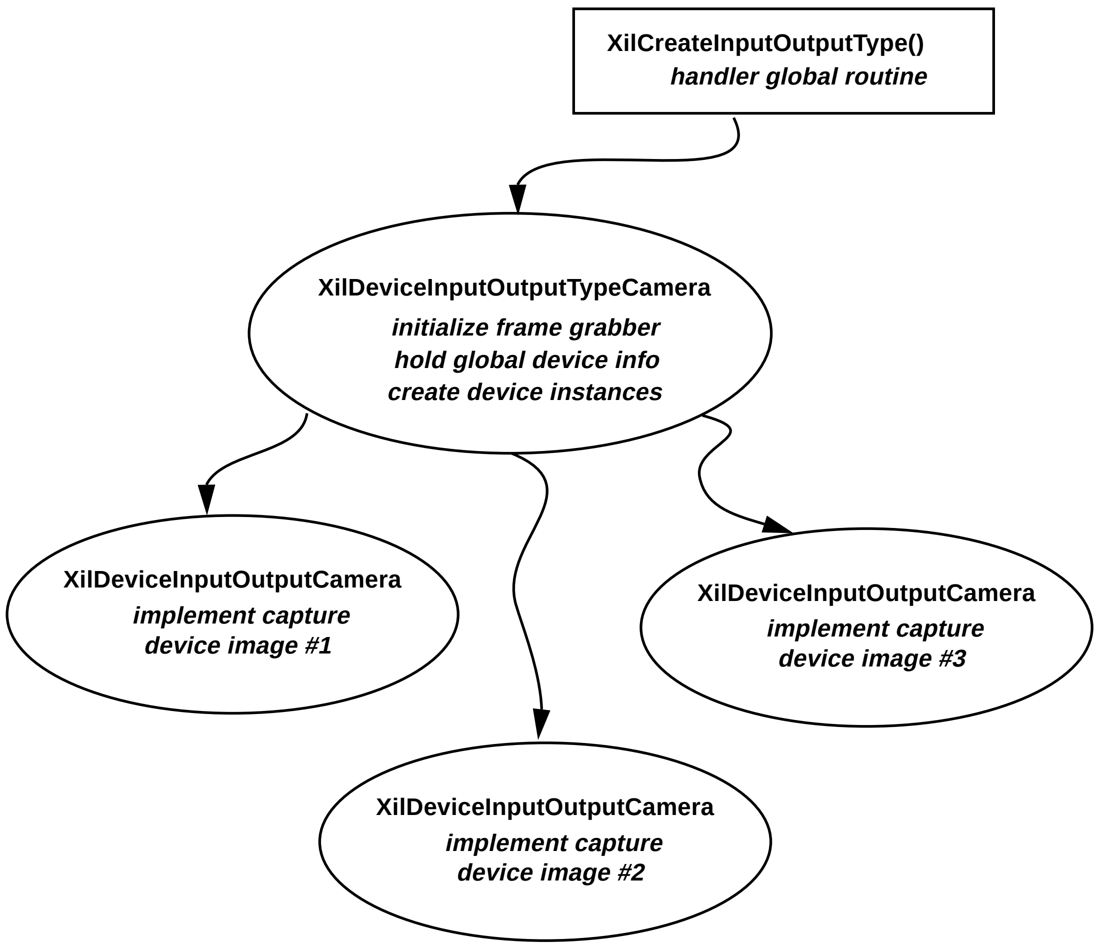

Figure 2-1 An Example of Creating an I/O Handler

The flow of creating a device handler is essentially the same for I/O, storage, and compression handlers (Figure 2-2), but not identical for compute devices.

Figure 2-2 Flow of Creating an I/O, Storage, or Compression Handler

Compute devices have only a single instantiation, which is controlled by the XIL core code. Thus, there is no derived class called `XilDeviceCompute`; only the `XilDeviceComputeType` class exists. Like the other device classes, the `XilDeviceComputeType` class must be subclassed, this time to represent the XIL functions that are being accelerated. The mechanism for allowing the XIL core code to instantiate the compute class is described in detail in Chapter 4, "Compute Devices."

The Development Environment

The porting interface for the XIL library is written in C++. Due to the lack of a stable binary interface for C++ compilers, it is important that device handler code be written with the same compiler as the interface part of the library. Two compilers are supported: SPARCompiler™ C++ v2.0 (or 2.0.1) and ProCompiler™ C++ 2.0.1.

SPARC – Although the *XIL Programmer's Guide* recommends the SPARCompiler C 3.0 or later compiler for building XIL applications, the SPARCompiler C++ 3.0 compiler cannot be used for writing device handler code.

The XIL library contains the XIL Test Suite. It enables you to perform regression tests against proven reference signatures. The XIL Test Suite is described in *XIL Test Suite User's Guide*, which is part of this software release.

The environment variable XIL_DEBUG can be useful in development situations. The options for XIL_DEBUG are described in Table 2-1.

Table 2-1 XIL_DEBUG Options

XIL_DEBUG **Option**	**Definition**
link*xx*	Add the two characters following the option link to the base name of the loadable handlers. This option is especially useful when you want to load a debug version of a handler. For example, link_g causes xilioxlib_g.so.1 to be loaded. If this version does not exist, then the handler with the standard name (xilioxlib.so.1) is loaded.
show_action	Print XIL_ACTION, the name of the device (e.g., XilDeviceComputeMemory) and the name of the function being called to execute each XIL operation (e.g., setvalue8()). For example, XIL_ACTION[XilDeviceComputeMemory]:setvalue8()
set_synchronize	Disable deferred execution.
verbose_cleanup	Print the name and class of each XIL object that you were responsible for destroying but did not. If an object that you were responsible for destroying is still on the list of active XIL objects at xil_close(), then the object's destroy function is invoked. An example of output for this option is Cleaning up an XIL_IMAGE object this object has no assigned name
no_cleanup	Do not destroy extra XIL objects at xil_close(). By default, if an object that you were responsible for destroying is still on the list of active XIL objects at xil_close(), then the object's destroy function is invoked.

Multiple variables may be set at once. For example, you could set XIL_DEBUG to "show_action:set_synchronize."

Installing XIL Device Handlers

By default, XIL looks for the device handlers in the directory

`/opt/SUNWits/Graphics-sw/xil/lib/pipelines`

This location is the default installation point for the Driver Developers Kit (DDK) packages. If XIL does not find the device handlers in this location, it will look in the directory

`/usr/openwin/etc/devhandlers`

The environment variable `XILHOME` overrides where XIL looks for the device handlers. If you set this environment variable, XIL looks for the device handlers only in

`$XILHOME/lib/pipelines`

The file `xil.compute` is a configuration file for handlers and their dependencies. This file resides in the same directory as the XIL runtime libraries (assuming default installation of the Software Developers Kit (SDK) packages):

`/opt/SUNWits/Graphics-sw/xil/lib`

If you have set the environment variable `XILHOME`, the library uses the `xil.compute` file in `$XILHOME/lib`.

Each of the following chapters that discuss I/O, storage, and compute drivers reference the `xil.compute` file and the changes you must make to it to add a new device handler.

Note – Be sure not to overwrite any existing files when you write your device handlers to the `pipelines` directory.

Error Reporting for XIL Device Handlers

All the possible error messages in the XIL library are listed in a file `xil.po`. This file is located in the directory

`/opt/SUNWddk/ddk_2.4/xil/src/doc`

As part of the release process, this file is compiled into another file, called `xil.mo`, which is used to localize error messages for different languages. In the system programmer release, the `xil.po` file is included. Where possible, you should make use of the currently existing error messages.

When you need to use device-specific error messages that are to included in the standard XIL release, you should create a new error file. The current `xil.po` file contains the device-independent error message IDs. These IDs are numbered and prefixed with the string `di-` (for example, `di-312`).

For device-dependent errors, the prefix for the error ID should be the device name for the handler. For example, for a handler with the device name `MYCAMERA`, the error IDs should have the form `MYCAMERA-123`. In this example, the XIL library looks for the device-specific error message number `123` in the directory

`/opt/SUNWits/Graphics-sw/xil/lib/locale/`*current_locale* \
`/LC_MESSAGES/MYCAMERA.mo`

The XIL library is internationalized; that is, it uses functions to extract error messages for a given locale. For information on localization of error messages and the creation of the `.mo` files, see the document *Developer's Guide to Internationalization* (available in AnswerBook).

What Kinds of Ports Are Possible in the XIL Library?

The mechanism for porting in the XIL library allows you to decide which functions would provide the maximum benefit for your customers. If an add-on card is only good at geometric operators, only those functions need to be ported; the memory versions of the remaining functions are called automatically. If the device is a general-purpose imaging accelerator, you may find it reasonable to provide a compute handler for most or all of the possible XIL atomic functions.

If only a compute handler is written, the XIL library expects that an image ends up residing in the CPU memory after each operation. If an accelerator has its own memory, it is often an advantage to allow the image data to reside on the device between operations. This avoids the overhead of having to copy the data back to the CPU after each operation. The XIL library has the concept of a storage handler, which is a set of functions which implements a copy to and from the specific device. If a storage handler is written, the XIL core code allows the image to reside in accelerator memory until another function requests that it be moved somewhere else. Writing a storage handler can greatly speed up a port for certain types of accelerator devices.

Additional molecules may be implemented by combining atomic functions in ways that accelerate specific application areas. Faster implementations of atomic functions can be used in place of the default implementation. While not properly a *device port*, molecules can greatly improve the performance of groups of operations.

For devices that act as either a source or destination image in an operation, the XIL library has the concept of an I/O handler. Once the handler is written, the application programmer can use the I/O device as a source or destination through the device image mechanism.

A single device may be represented by more than one handler. For example, an input frame grabber that has integrated processing support can be described by an I/O handler and an associated compute handler. If it appears as though multiple processing operations will be done often on the grabbed images, a storage handler can be written for the frame-grabber board as well.

Compression devices must implement the compression but may be associated with other compute, storage, or I/O handlers as well.

The following chapters will describe each type of handler in detail.

What Kinds of Ports Are Not Possible in the XIL Library?

The major constraint on porting in the XIL library is that the set of atomic functions may not be extended by the user. All molecules, including those going to I/O hardware, must be made up of groups of the atomic functions that the XIL library defines and implements. The list of available atomic functions is given in Appendix B, “XIL Atomic Functions.”

In addition, the IHV should not change the meaning of existing atomic functions; a new implementation should do exactly what the original version does. The correctness of a new function can be tested using the XIL Test Suite.

Porting of functions not defined by the XIL library must be performed using the mechanism defined by `xil_export()`.

Version Control for XIL Handlers

The XIL core contains a global function:

`xilVersionPtr* XilGetVersion()`

This function returns a pointer to a structure that contains 16-bit unsigned integers containing the major and minor release numbers of the current XIL library. The structure looks like this:

```
typedef struct {
    Xil_unsigned16 majorVersion;
    Xil_unsigned16 minorVersion;
} *xilVersionPtr;
```

The rules for loading handlers are fairly simple:

- The library will not load a module with a `majorVersion` greater than its own. An attempt to load a module greater than the current library version results in an error.
- Currently, the allowable (earlier) module versions that are supported are versions 1.1 and 1.2. Thus, `majorVersion` can only equal 1, and `minorVersion` can equal either 1 or 2.
- The library loads and executes any module with the same `majorVersion` number.

Similar version control rules exist for all of the OGI foundation libraries, including the port for the OpenWindows™ software.

These rules have implications for writers of XIL device handlers. You should write your handler with the earliest version of the Solaris OS that you wish to support. Upgrading to a new OS version by the end user will, in general, not require a new release of XIL device handlers. If you wish to write a handler that requires functionality only available after a specific library release, you must check the `majorVersion` and `minorVersion` numbers to make sure the handler has been loaded by an appropriate version of the library.

In order for the library to properly load handlers, the name of the handler must contain its major version number as a suffix. For example, the standard XIL I/O handler for X11 support is called `xilioxlib.so.1`. For the 1.*x* release of the XIL library, it is sufficient to ensure that each handler name includes the suffix `.1`.

I/O Devices 3

About I/O Devices

In the XIL Imaging Library, I/O devices include any devices that can generate or receive images, such as frame grabbers, image files, and displays. The XIL library supports these types of devices by allowing them to appear as device images to an application. When a device image is used as a source in an operation, an image is captured from the device. When a device image is used as a destination in an operation, an image is written to the device.

The I/O device handler provides an implementation for an image captured from a device and for an image written to a device. The first time a device image is created using the `xil_create_from_device()` API call, the software module containing the handler is loaded. Once the I/O handler is loaded, any compute devices that have only the I/O handler as a dependence are loaded. The I/O handler information is cached so that subsequent creations of new device images from the same device do not require reloading the I/O handler.

The character string representing the name of the device, passed as the second argument to `xil_create_from_device()`, is used to select the appropriate loadable library. Currently, the following API call attempts to load an I/O handler named `/opt/SUNWits/Graphics-sw/xil/lib/pipelines/xiliomy_device.so.1` and fails with an error if this loadable library does not exist:

```
device_image = xil_create_from_device(systemState, “iomy_device”,
NULL);
```

Note – For I/O handlers that are frame buffers, the string returned by the `ioctl` call for `VIS_GETINDENTIFIER` must match the name of the loadable device handler. XIL prepends the unique string returned by this `ioctl` call with `io`. Therefore, XIL is looking for `xilio`*vis_identifier.name*`.so.1`. For more information about graphics device drivers and `ioctl`, see the man page for `visual_io` or the manual *Writing Device Drivers*.

`XilDeviceInputOutputType` *Class*

As described in Chapter 1, "Overview," the abstract class `XilDeviceType` is subclassed by the library to form the *Type* handler for each kind of handler the library supports. For I/O devices, the abstract class `XilDeviceInputOutputType` is defined (see Code Example 3-1). This class must be further subclassed to represent the particular I/O device type represented in the handler. Only one instantiation of this class exists for each type of I/O device created by the device-specific driver.

Code Example 3-1 Definition of `XilDeviceInputOutputType` Class

```
#include "XilDeviceType.h"

class XilDeviceInputOutputType : public XilDeviceType {
 public:

 // This function is used to create instances of the input/output
 // object.

 virtual XilDeviceInputOutput* createDeviceInputOutput(
    XilImage* parent, XilAttribute* attribs)=0;


 // destructor. This should release all resources that were used to
 // make the connection to the device.

 virtual ~XilDeviceInputOutputType();

};
```

When the handler is loaded, the library looks through the symbol table of the loadable library for a specific function that must exist in each I/O handler:

```
XilDeviceInputOutputType* XilCreateInputOutputType()
```

The library invokes this function, which is responsible for doing any global, one-time initialization of the device, and sets up any data that will be used by all instances of this I/O device. This derived "type" class must contain the function `createDeviceInputOutput()`, which creates each instance of the device class. This function is declared with an `XilAttribute*` parameter. The `XilAttribute` object is used for atomically setting multiple device attributes. See the section "The XilAttribute Class" in Chapter 1 for more information.

The section "Sample I/O Handler" on page 75 contains a derived type class `XilDeviceInputOutputTypeCG6`, which instantiates the device class `XilDeviceInputOutputCG6`. Likewise, the section "Sample I/O Device Handler" on page 106 contains a derived type class `XilDeviceInputOutputTypeXlib`, which instantiates the device class `XilDeviceInputOutputXlib`.

Handling Multiple Devices in an I/O Handler

The I/O "type" class (for example, `XilDeviceInputOutputTypeCG6`) is responsible for keeping track of multiple devices. This tracking is accomplished by the use of a linked list of descriptors. Each entry in the list describes a given device, and each frame buffer attached to the system has a descriptor. The descriptors have the same fields but different values. Each window, for example, created on a device (such as a frame buffer) is an instantiation of the device class. The class contains information that maps the device back to a descriptor entry in the linked list. Each device stores its own specific information (such as position/size on the screen). The example in the section "Sample I/O Handler" on page 75 illustrates these concepts.

Consider the case of a system with two CG6 frame buffers and a window on each:

```
XilDeviceInputOutputTypeCG6
    head-> descriptor_CG60    --+
                                |next
            descriptor_CG61    <-+
instantiation 0
XilDeviceInputOutputCG6
    window0
        my_descriptor - descriptor_CG60
        specifics of window0
instantiation 1
XilDeviceInputOutputCG6
    window1
        my_descriptor = descriptor_CG61
        specifics of window1
```

`XilDeviceInputOutput` *Class*

The handler creates a device specific class that derives from `XilDeviceInputOutput`. This is where all of the device-specific information pertaining to a particular instance of the I/O device is stored. For example, an X display object might store information about a display, window, and graphics context here. See Code Example 3-2 for the definition of the base `XilDeviceInputOutput` class. A new instance of this class must be created for each device image.

Code Example 3-2 Definition of `XilDeviceInputOutput` Class *(1 of 3)*

```
//----------------------------------------------------------------------
//
// Description:
//
// The XilDeviceInputOutput class describes one instantiation of a
// particular input/output device. There can be many of these.
// See the example input/output driver for more information.
//
//----------------------------------------------------------------------

class XilDeviceInputOutput : public XilDevice {
```

Code Example 3-2 Definition of `XilDeviceInputOutput` Class *(2 of 3)*

```
public:

// set a device-specific attribute
virtual int setDeviceAttribute (char attribute_name[], void *value)=0;

// get (return a pointer to) a device-specific attribute.
virtual int getDeviceAttribute (char attribute_name[], void **value)=0;

// implement display on this input/output device
virtual void display (XilImage*)=0;

// implement capture on this input/output device
virtual void capture (XilImage*)=0;

// get and set particular image pixel values directly
// from/to the device
virtual void getPixel(unsigned short x, unsigned short y,
      unsigned short band, unsigned short count,
      float* data)=0;
virtual void setPixel(unsigned short x, unsigned short y,
      unsigned short band, unsigned short count,
      float* data)=0;

// return the pointer to the image the input/output device uses
// as its buffer to (display from / capture to)
XilImage* getParent ();

// return the XilImageType that should be returned from the
// xil_create_from_device() call
XilImageType* getImageType();

// return the actual XilImageType that is used internally.
// this may be different from the way it appears to the
// application. For example, the image type for a 24-bit
// frame buffer may be a 3-banded image, whereas the real
// image type would be a 4-band image to make the copy to
// the display faster.
XilImageType* getRealImageType();

// functions to indicate whether this device can be read from/
// written to
Xil_boolean isReadable ();
```

Code Example 3-2 Definition of `XilDeviceInputOutput` Class *(3 of 3)*

```
  Xil_boolean isWritable ();

  // return the op number of the display operation associated with this device
  unsigned short getDisplayOpNumber();

  // return the op number of the capture operation associated with this device
  unsigned short getCaptureOpNumber();

  // flag to indicate whether the buffer image contents are valid
  // (if a molecule has been used to write directly to the display,
  // for example, the buffer image is no longer valid)
  Xil_boolean memoryIsValid();

  // set the memory-valid bit
  void setMemoryValid(Xil_boolean valid);

  //destructor
  virtual ~XilDeviceInputOutput();

  protected:
  XilImage*           parent;
  XilImageType*       imageType;
  XilImageType*       realImageType;
  Xil_boolean         readable;
  Xil_boolean         writeable;
  unsigned short      displayOpNumber;
  unsigned short      captureOpNumber;
  Xil_boolean         memoryValid;
};
```

Device Attribute Functions

I/O devices may define attributes that are used to modify or report their behavior. For example, a frame grabber would use attributes to allow the application to select the type of output image or to select which video input to use. A file input device would use attributes to set the path name.

`setDeviceAttribute()` takes an attribute and a value and performs the device-specific function that the attribute defines.

`getDeviceAttribute()` returns the value associated with the given device-specific attribute.

`getPixel()` returns pixel information for the I/O device.

`setPixel()` sets the device pixel to the given value.

`display()` causes the parent image to be copied to the device.

`capture()` causes the device to copy data into the parent image.

Device image attributes are defined by the port, but some frame-buffer-specific attributes have already been defined by the standard handlers. These attributes are listed in Table 3-1:

Table 3-1 Standard Frame Buffer Attributes

Attribute	**Value**
`XCOLORMAP`	The X colormap of the device image (write only)
`WINDOW`	The X window (read only)
`DISPLAY`	The X display (read only)

Future releases of the XIL library will suggest common attributes for camera-like I/O devices; other I/O devices such as files and printers will have other attributes.

Parent Function

The `parent` member is the image that holds the results of the capture and the source of the display. When a device image is used as a source, the library inserts a device-dependent capture into the current operation sequence and returns the parent image. When a device image is used as a destination, the library inserts a copy from the source to the parent image and then inserts a display operation into the current operation sequence. The `parent` is created by the XIL library and is passed to the constructor for the `XilDeviceInputOutput` subclass. The `parent` is primarily used in the constructor of the derived `XilDeviceInputOutput` class to set up the image type of the device image that is stored in `imageType`.

Image Type Functions

The `imageType` member holds the image type of the device image. The `imageType` describes the size, number of bands, and data type of the image that is required or will be generated by the device image. The `imageType` is created by the handler using the `createImageType()` member of the system state. Sometimes, it is necessary to return to the application an `imageType` different from the actual description of the device. For example, the `imageType` for a 24-bit frame buffer may be a 3-banded image, but the true `imageType` is a 4-band image. The `realImageType` member holds the description of the actual device; normally, the `imageType` is the same as the `realImageType`.

```
realImageType = imageType = parent->getSystemState()->
          createImageType(xsize,ysize,nbands,datatype);
```

Read- and Write-Only Functions

The `readable` and `writeable` members allow a device to appear as read-only or write-only. If the handler wishes for a device to be read-only, these members can be set appropriately when the class is created.

Op Number Functions

The members `displayOpNumber` and `captureOpNumber` are ordinals that describe the type of operation. They provide a unique reference to the specific capture or display operation in order for it to be placed into the DAG. The `displayOpNumber` and `captureOpNumber` members are assigned automatically when the I/O handler is loaded and the device type is created. The number (opcode) is keyed from the function names `display_`*devicename* and `capture_`*devicename*, where *devicename* is the name of the I/O device handler. Each device class instantiation must request and store these opcodes. The opcodes are obtained by using the `XilLookupOpNumber()` function with the appropriate keyname. Shown next is the action for the device `ioSUNWcg6`:

```
displayOpNumber = XilLookupOpNumber("display_ioSUNWcg6");
captureOpNumber = XilLookupOpNumber("capture_ioSUNWcg6");
```

`XilLookupOpNumber()` also is used for compute handlers. See Chapter 4, “Compute Devices,” for more information.

When a device image is the destination or source of an operation, the library inserts a display or capture operation into the current operation sequence of the DAG. The core uses the `getDisplayOpNumber` and `getCaptureOpNumber` member functions to extract the operation number.

Adding an I/O Device

Adding an I/O device is straightforward in the XIL library. The handler writer must follow these steps:

1. Implement the `XilCreateDeviceInputOutput()` function. It must create an instance of the derived `XilDeviceInputOutputType` class.

2. Implement the `XilDeviceInputOutputType` class to provide device creation and initialization. Place all common information for all instances of the device in this class.

3. Implement the `XilDeviceInputOutput` class for the device, including `capture()`, `display()`, `get/setPixel()`, as well as `get/setDeviceAttribute()` if needed.

4. Place the new loadable library file in an application package so that it will be installed in the correct location. See the document *SunOS Application Packaging and Installation Guide* for information on using the package system. Also see Chapter 1, “Overview,” for information about packaging handlers.

   The name of the loadable library must be unique; we strongly suggest using the name `xilio`*device_name*`.so`, where *device_name* is the name that will be used to describe the device in the `xil_create_from_device()` API call. The first part of the *device_name* should be a unique identifier for the company producing the handler; for example, all Sun I/O handlers should contain the string `SUNW` as the first part of the device name.

Sample I/O Handler

This section shows an example I/O device handler that treats a SPARC GX frame-buffer window as an I/O device. It’s an important example because it illustrates how to write an I/O handler that talks directly to hardware using DGA (Direct Graphics Access). The files for the GX example are located in directory `/opt/SUNWddk/ddk_2.4/xil/src/cg6_device_handler`.

A parallel example for an x86-specific module treats a p9000 frame-buffer window as an I/O device. The p9000 example isn’t shown in this manual because the p9000 architecture is similar to the CG6 architecture. The p9000 code is included to demonstrate some of the differences you can expect when writing an XIL module for x86. The files for the p9000 example are located in directory `/opt/SUNWddk/ddk_2.4/xil/src/p9000`.

The GX example shown below has four files:

- `XilDeviceInputOutputTypeCG6.h`
- `XilDeviceInputOutputTypeCG6.cc`
- `XilDeviceInputOutputCG6.h`
- `XilDeviceInputOutputCG6.cc`

The p9000 example, not shown but located in directory `/opt/SUNWddk/ddk_2.4/xil/src/p9000`, also has four files:

- `XilDeviceInputOutputTypeP9000.h`
- `XilDeviceInputOutputTypeP9000.cc`
- `XilDeviceInputOutputP9000.h`
- `XilDeviceInputOutputP9000.cc`

XilDeviceInputOutputTypeCG6.h

Code Example 3-3 XilDeviceInputOutputTypeCG6.h

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      XilDeviceInputOutputTypeCG6.h
//  Project:   XIL
//  Created:   93/08/20
//  RespEngr:  John L. Furlani
//  Revision:  1.1
//  Last Mod:  18:14:19, 07 Sep 1993
//
//  Description:
//    This file contains the description of the CG6 device type
//    class.  The device type is created once for each instance of XIL
//    -- not on a per-window basis.  It contains all of the device
//    information which does not change from window-to-window (like
//    the device mapping).
//
//-----------------------------------------------------------------------
#pragma ident  “@(#)XilDeviceInputOutputTypeCG6.h1.1\t93/09/07  “

#ifndef XILDEVICEINPUTOUTPUTTYPECG6
#define XILDEVICEINPUTOUTPUTTYPECG6

#include <stdlib.h>
#include <sys/cg6fbc.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xil/XilDeviceInputOutputType.h>
#include <xil/XilError.h>
#include <dga/dga.h>

//
//  A structure that describes a mapping of a CG6 device.  Only one
//  mapping is created for all of the windows on the screen.  A list
//  is kept where each node represents a single mapping of a different
//  CG6 device.
//
struct CG6Description {
   int            fd;
   unsigned char* fb_mem;
```

Code Example 3-3 `XilDeviceInputOutputTypeCG6.h` *(Continued)*

```
   int            fb_height;
   int            fb_width;
   int            fb_size;
   fbc*           fb_fbc;
   char name[32];
   struct CG6Description* next;
};

class XilDeviceInputOutputTypeCG6 : public XilDeviceInputOutputType {
public:
    //
    //  The routine called by the XIL core to create the device type.
    //
    virtual XilDeviceInputOutput* createDeviceInputOutput(XilImage* parent,
                                                          void*     data);

    //
    //  Our constructors and destructors...
    //
    XilDeviceInputOutputTypeCG6();
    ~XilDeviceInputOutputTypeCG6();

    //
    //  This routine returns a full description of the given CG6
    //  device.  A list of CG6 descriptions is kept in this class so
    //  multiple windows on the same device will not have multiple
    //  mappings.
    //
    CG6Description* getCG6Description(char* device_name);

private:
    //
    //  The CG6 description list...
    //
    CG6Description* baseCG6Description;
};

#endif
```

XilDeviceInputOutputTypeCG6.cc

Code Example 3-4 XilDeviceInputOutputTypeCG6.cc *(1 of 6)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-------------------------------------------------------------------------
//
//  File:      XilDeviceInputOutputTypeCG6.cc
//  Project:   XIL
//  Created:   93/08/20
//  RespEngr:  John L. Furlani
//  Revision:  1.1
//  Last Mod:  18:14:28, 07 Sep 1993
//
//-------------------------------------------------------------------------
#pragma ident  “@(#)XilDeviceInputOutputTypeCG6.cc1.1\t93/09/07  “

#include <sys/fbio.h>
#include <sys/cg6reg.h>
#include <sys/mman.h>

#include “XilDeviceInputOutputTypeCG6.h”
#include “XilDeviceInputOutputCG6.h”

//
//  The XIL core calls this routine when opening this I/O pipeline to
//  create the DeviceInputOutputType for the CG6 device.
//
XilDeviceInputOutputType *XilCreateInputOutputType()
{
    return(new XilDeviceInputOutputTypeCG6());
}

XilDeviceInputOutputTypeCG6::XilDeviceInputOutputTypeCG6()
{
    //
    //  Initialize the list of CG6 description structure to NULL
    //
    baseCG6Description=NULL;
}

XilDeviceInputOutputTypeCG6::~XilDeviceInputOutputTypeCG6()
{
    CG6Description* temp = baseCG6Description;
```

Code Example 3-4 `XilDeviceInputOutputTypeCG6.cc` *(2 of 6)*

```
    while(baseCG6Description) {
        baseCG6Description = baseCG6Description->next;
        delete temp;
        temp = baseCG6Description;
    }
}

//
//  The XIL core calls this routine when the user calls
//  xil_create_from_window() with an X window that resides on a CG6.
//  For every display window the user opens, this routine is called
//  a new instantiation of XilDeviceInputOutputCG6 is created.
//
XilDeviceInputOutput*
XilDeviceInputOutputTypeCG6::createDeviceInputOutput(XilImage* parent,
                                                     void*     data)
{
    //
    //  Create an instantiation of the DeviceInputOutput for this
    //    window.
    //
   XilDeviceInputOutputCG6* device = new XilDeviceInputOutputCG6(parent,data);
    if(device==NULL) {
        XIL_ERROR(NULL,XIL_ERROR_RESOURCE,"di-1",TRUE);
        return NULL;
    }

    //
    //  Check that it was created successfully by getting the
    //    ImageType which is set in the constructor.
    //
    if(device->getImageType()==NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-1",TRUE);
        delete device;
        return NULL;
    }

    return device;
}

//
//  The routine that creates and manages the CGDescription list with a
```

Code Example 3-4 `XilDeviceInputOutputTypeCG6.cc` *(3 of 6)*

```
//    single node entry per CG6 device on the system.
//
CG6Description*
XilDeviceInputOutputTypeCG6::getCG6Description(char* name)
{
   //
   //  Look through the list to determine if the device has already
   //  been opened.
   //
   CG6Description* tmp = baseCG6Description;
   while(tmp) {
       if(strcmp(name, tmp->name)==NULL) {
           return tmp;
       }
       tmp = tmp->next;
   }

   //
   //  Well, this device hasn't opened yet so we'll go ahead and
   //  create a new description.
   //
   CG6Description* description= new CG6Description;
   if(!description) {
       XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-1",TRUE);
       return NULL;
   }
   strcpy(description->name,name);

   //
   //  Now we actually open the device and get its attributes.
   //
   description->fd = open(description->name, O_RDWR);
   if(description->fd < 0) {
       XIL_ERROR(NULL,XIL_ERROR_RESOURCE,"di-212",TRUE);
       delete description;
       return NULL;
   }

   //
   //  Get the device attributes.
   //
   struct fbgattr attr;
```

Code Example 3-4 `XilDeviceInputOutputTypeCG6.cc` *(4 of 6)*

```
if(ioctl(description->fd, FBIOGATTR, &attr) < 0) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-213",TRUE);
    delete description;
    return NULL;
}

//
//  Be certain the device is really a CG6.
//
if(attr.real_type != FBTYPE_SUNFAST_COLOR) {
    //
    //  Somehow we were called on a non-CG6 framebuffer.
    //    Definite error.
    //
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-214",TRUE);
    delete description;
    return NULL;
}

//
//  Get the information describing the CG6 from the FBIOGXINFO
//  ioctl call.
//
cg6_info cg6_information;
if(ioctl(description->fd, FBIOGXINFO, &cg6_information) < 0) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-213",TRUE);
    delete description;
    return NULL;
}

//
//  Fill our description of the CG6.
//
description->fb_width  = cg6_information.accessible_width;
description->fb_height = cg6_information.accessible_height;
description->fb_size   = cg6_information.vmsize*1024*1024;

//
//  Get the register mappings.
//
description->fb_fbc = (fbc*)
    mmap(NULL, CG6_FBCTEC_SZ, PROT_READ|PROT_WRITE, MAP_PRIVATE,
```

Code Example 3-4 `XilDeviceInputOutputTypeCG6.cc` *(5 of 6)*

```
            description->fd, CG6_VADDR_FBCTEC);
   if(description->fb_fbc == (fbc*) -1) {
       XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-215”,TRUE);
       delete description;
       return NULL;
   }

   //
   //  Map the framebuffer itself
   //
   description->fb_mem = (Xil_unsigned8*)
       mmap(NULL, description->fb_size, PROT_READ|PROT_WRITE, MAP_SHARED,
            description->fd, CG6_VADDR_COLOR);
   if(description->fb_mem == (Xil_unsigned8*) -1) {
       XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-215”,TRUE);
       munmap((caddr_t)description->fb_fbc, CG6_FBCTEC_SZ);
       delete description;
       return NULL;
   }

   //
   //  Wait to ensure the GX is idle and ready for us to set some
   //  registers.
   //
   while(description->fb_fbc->l_fbc_status & L_FBC_BUSY);

   //
   //  Initialize the registers to what we want done.
   //
   description->fb_fbc->l_fbc_misc.l_fbc_misc_blit=L_FBC_MISC_BLIT_NOSRC;
   description->fb_fbc->l_fbc_misc.l_fbc_misc_data=L_FBC_MISC_DATA_COLOR8;
   description->fb_fbc->l_fbc_misc.l_fbc_misc_draw=L_FBC_MISC_DRAW_RENDER;
   description->fb_fbc-
>l_fbc_misc.l_fbc_misc_bwrite0=L_FBC_MISC_BWRITE0_ENABLE;
   description->fb_fbc-
>l_fbc_misc.l_fbc_misc_bwrite1=L_FBC_MISC_BWRITE1_DISABLE;
   description->fb_fbc->l_fbc_misc.l_fbc_misc_bread=L_FBC_MISC_BREAD_0;
   description->fb_fbc->l_fbc_planemask= 0xff;
   description->fb_fbc->l_fbc_pixelmask= 0xffffffff;
   description->fb_fbc->l_fbc_clipcheck= 0;
   description->fb_fbc->l_fbc_rasteroffx= 0;
   description->fb_fbc->l_fbc_rasteroffy=0;
```

Code Example 3-4 `XilDeviceInputOutputTypeCG6.cc` *(6 of 6)*

```
    description->fb_fbc->l_fbc_autoincx= 0;
    description->fb_fbc->l_fbc_autoincy= 0;

    //
    //  Add the newly created description to our description list and
    //    return the new description to the caller.
    //
    description->next= baseCG6Description;
    baseCG6Description= description;

    return description;
}
```

XilDeviceInputOutputCG6.h

Code Example 3-5 XilDeviceInputOutputCG6.h *(1 of 3)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------*-------------------
//
//  File:      XilDeviceInputOutputCG6.h
//  Project:   XIL
//  Created:   93/08/20
//  RespEngr:  John L. Furlani
//  Revision:  1.1
//  Last Mod:  18:13:44, 07 Sep 1993
//
//  Description:
//    This file contains the device instantiation-specific
//    information for the CG6 device.  This object is created on a
//    per-displayimage basis by the XIL core.  It is responsible for
//    the per-window access to the CG6 device.  This includes
//    display/capture/setPixel/getPixel.
//
//------------------------------------------------------------------------
#pragma ident  "@(#)XilDeviceInputOutputCG6.h1.1\t93/09/07  "

#ifndef XILDEVICEINPUTOUTPUTCG6
#define XILDEVICEINPUTOUTPUTCG6

#include <stdlib.h>
#include <sys/cg6reg.h>
#include <sys/cg6fbc.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xil/XilDeviceInputOutput.h>
#include <xil/XilError.h>
#include <dga/dga.h>

class XilDeviceInputOutputCG6 : public XilDeviceInputOutput {
public:
    //
    //  Set and get CG6 device-specific attributes
    //
    virtual int setDeviceAttribute(char attribute_name[], void* value);
    virtual int getDeviceAttribute(char attribute_name[], void**);
```

Code Example 3-5 `XilDeviceInputOutputCG6.h` *(2 of 3)*

```
//
//  The display routine which copies the entire XIL image backing
//  store to the display.  And, the capture routine which updates
//  the XIL image backing store with what's actually on the display.
//
virtual void display(XilImage*);
virtual void capture(XilImage*);

//
//  The single-pixel operations to just modify a few pixels.
//
virtual void getPixel(unsigned short x, unsigned short y,
                      unsigned short band, unsigned short count,
                      float *data);
virtual void setPixel(unsigned short x, unsigned short y,
                      unsigned short band, unsigned short count,
                      float *data);

//
//  The constructor and destructor.
//
XilDeviceInputOutputCG6(XilImage* parent, void* data);
virtual ~XilDeviceInputOutputCG6();

//
//  Our publicly available data which describes the window
//
Display* displayptr;
Window window;
Dga_window infop;
short* cliplist;
int win_x,win_y,win_width,win_height;
int fd;

//
//  Colormap Installation Info
//
Dga_cmap dga_cmap;
Colormap xcmap;

//
//   This flag indicates whether we're on a machine that
```

Code Example 3-5 `XilDeviceInputOutputCG6.h` *(3 of 3)*

```
    //   only has CG6 display devices that can use the double
    //   loads and stores
    //
    Xil_boolean    doubleLoadnStore;

    //
    //  CG6 Specific Info
    //
    short          fb_width;
    short          fb_height;
    int            fb_size;
    unsigned char* fb_mem;
    fbc*           fb_fbc;
    int            fhc_config;
    int*           dac_base;
    int*           tec_base;
    cg6_cmap*      cg6cmap;
};

#endif
```

XilDeviceInputOutputCG6.cc

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(1 of 18)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:XilDeviceInputOutputCG6.cc
//  Project:XIL
//  Created:93/08/20
//  RespEngr:John L. Furlani
//  Revision:1.1
//  Last Mod:18:13:49, 07 Sep 1993
//
//-----------------------------------------------------------------------
#pragma ident"@(#)XilDeviceInputOutputCG6.cc1.1\t93/09/07  "

#include <sys/utsname.h>
#include <sys/mman.h>
#include <X11/Xlib.h>
#include <xil/xilwindow.h>
#include <xil/XilImage.h>
#include <xil/XilOp.h>
#include <xil/XilRoi.h>
#include <xil/XilRoiList.h>
#include <xil/XilColorDefines.h>
#include <xil/xil_memcpy.h>

#include "XilDeviceInputOutputTypeCG6.h"
#include "XilDeviceInputOutputCG6.h"


XilDeviceInputOutputCG6::XilDeviceInputOutputCG6(XilImage* parent,
                                                 void*     data)
{
    //
    //  Initialize the imageType to NULL to indicate the creation of
    //    this device has not succeeded.
    //
    imageType = NULL;

    //
    //  Store a pointer to my parent...
    //
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(2 of 18)*

```
this->parent = parent;

//
//  The CG6 is both readable and writeable.
//
readable = writeable = TRUE;

//
//  Determine if this machine has a sun4 architecture or not.  If
//  it is a sun4 architecture, then we must use the plain memcpy()
//  because CG6's on the sun4 architecture are connected to the P4
//  bus which does not support double loads and stores.  Otherwise,
//  we've got an SBus based CG6.
//
struct utsname uname_info;
if(uname(&uname_info) == -1) {
    XIL_ERROR(NULL, XIL_ERROR_SYSTEM, "di-315", TRUE);
    return;
}
if(!strcmp(uname_info.machine, "sun4")) {
    doubleLoadnStore = FALSE;
} else {
    doubleLoadnStore = TRUE;
}

//
//  Indicate to the XIL core that the memory version of the window
//  is currently not valid which means a capture is required if
//  someone tries to read from the display image.
//
memoryValid = FALSE;

//
//  Initialize the colormap variables for this window.  The xcmap
//  and the dga_cmap are initially set to NULL vales to indicate
//  that the user has not set the X_COLORMAP device attribute.
//  The colormap information is initialized each time the user
//  calls X_COLORMAP.
//
xcmap = 0;
dga_cmap = NULL;
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(3 of 18)*

```
//
//  Save the X Display and the X Window
//
XilWindow* xil_window = (XilWindow*)data;

this->displayptr = xil_window->display;
this->window     = xil_window->window;

//
//  Here we connect to DGA and turn the window we've been given
//    into a DGA window so we can access the hardware directly.
//
int dga_token = XDgaGrabWindow(xil_window->display, xil_window->window);
if(dga_token == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-219",TRUE);
    return;
}

if((infop = ((Dga_window) dga_win_grab(-1, dga_token))) == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-219",TRUE);
    XDgaUnGrabWindow(xil_window->display, xil_window->window);
    return;
}


//
//  Get a pointer to the CG6 type so I can get the information
//    about the device my window is on.
//
XilDeviceInputOutputTypeCG6* io_cg6_type = (XilDeviceInputOutputTypeCG6*)
    xil_global_state->getDeviceInputOutputType("ioSUNWcg6");
if(io_cg6_type == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-188",FALSE);
    return;
}

//
//  Create or get the already created device information from the
//    CG6 type.
//
CG6Description* cg6_description =
    io_cg6_type->getCG6Description(dga_win_fbname(infop));
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(4 of 18)*

```
if(!cg6_description) {
    //
    //  TODO: generate an appropriate error
    //
    return;
}

//
//  Initialize our own variables.
//
this->fd        = cg6_description->fd;
this->fb_width  = cg6_description->fb_width;
this->fb_height = cg6_description->fb_height;
this->fb_mem    = cg6_description->fb_mem;
this->fb_fbc    = cg6_description->fb_fbc;

//
//  Determine what the imageType is going to be for this XIL
//    display image.
//
Window       root_window;
unsigned int x_depth;
unsigned int height;
unsigned int width;
int          x,y;
unsigned int border_width;
Status status = XGetGeometry(displayptr,
                             this->window,
                             &root_window,
                             &x, &y,
                             &width, &height,
                             &border_width, &x_depth);
if(status == 0) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-220",TRUE);
    return;
}

//
//  For the CG6 the imageType and the realImageType are the same.
//
imageType = realImageType =
    parent->getSystemState()->createImageType(width,height,1,XIL_BYTE);
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(5 of 18)*

```
    if(imageType == NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-188",FALSE);
        return;
    }

    //
    //  Lookup and cache the display and capture op numbers.
    //
    displayOpNumber = XilLookupOpNumber("display_ioSUNWcg6");
    captureOpNumber = XilLookupOpNumber("capture_ioSUNWcg6");
}

XilDeviceInputOutputCG6::~XilDeviceInputOutputCG6()
{
   dga_win_ungrab(infop, 1);
   XDgaUnGrabWindow(displayptr,window);
   if (imageType) imageType->destroy();
}

void
XilDeviceInputOutputCG6::setPixel(unsigned short x,
                                  unsigned short y,
                                  unsigned short,
                                  unsigned short,
                                  float*         data)
{
    //
    //  Lock the DGA window so I can access the framebuffer.
    //
    DGA_WIN_LOCK(infop);

    //
    //  We grab the cliplist every time since checking it may have
    //  cause problems with molecules that go directly to the display
    //  which need to know if the window has been modified.  It's
    //  cheap enough.
    //
    cliplist = dga_win_clipinfo(infop);
    dga_win_bbox(infop, &win_x, &win_y, &win_width, &win_height);

    //
    //  Transform the point into screen space.
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(6 of 18)*

```
//
y = y+win_y;
x = x+win_x;

//
//  Loop over the cliplist to figure out which rectangle contains
//  the pixel we're looking for.
//
while(*cliplist != DGA_Y_EOL) {
    if((y >= (unsigned short)cliplist[0]) &&
       (y <= (unsigned short)cliplist[1])) {
        cliplist += 2;
        while(*cliplist != DGA_X_EOL) {
            if((x >= (unsigned short)cliplist[0]) &&
               (x <= (unsigned short)cliplist[1])) break;
            cliplist += 2;
        }
        if(*cliplist != DGA_X_EOL) break;
        cliplist++;
    } else {
        cliplist += 2;
        while(*cliplist != DGA_X_EOL) cliplist += 2;
        cliplist++;
    }
}

if(*cliplist != DGA_Y_EOL) {
    //
    //  Set the pixel now that we've found the right rectangle on
    //    the screen.
    //
    Xil_unsigned8* dst = fb_mem + y*fb_width + x;

    //
    //  Round the pixel up.
    //
    float value = data[0] + .5;

    //
    //  Clip the value properly and set it on the screen.
    //
    if(value > 255.0) {
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(7 of 18)*

```
            *dst = 255;
        } else if (value < 0.0) {
            *dst = 0;
        } else {
            *dst = (unsigned char) value;
        }
    }

    //
    //  Unlock the window...
    //
    DGA_WIN_UNLOCK(infop);
}

void XilDeviceInputOutputCG6::getPixel(unsigned short x,
                                       unsigned short y,
                                       unsigned short,
                                       unsigned short,
                                       float*         data)
{
    //
    //  Lock the DGA window so I can access the framebuffer.
    //
    DGA_WIN_LOCK(infop);


    //
    //  We grab the cliplist every time since checking it may have
    //  cause problems with molecules that go directly to the display
    //  which need to know if the window has been modified.  It's
    //  cheap enough.
    //
    cliplist = dga_win_clipinfo(infop);
    dga_win_bbox(infop, &win_x, &win_y, &win_width, &win_height);


    //
    //  Transform the point into screen space.
    //
    y = y + win_y;
    x = x + win_x;
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(8 of 18)*

```
    //
    //  Loop over the cliplist to figure out which rectangle contains
    //  the pixel we're looking for.
    //
    while(*cliplist != DGA_Y_EOL) {
        if((y >= (unsigned short)cliplist[0]) &&
           (y <= (unsigned short)cliplist[1])) {
            cliplist += 2;
            while(*cliplist != DGA_X_EOL) {
                if((x >= (unsigned short)cliplist[0]) &&
                   (x <= (unsigned short)cliplist[1])) break;
                cliplist += 2;
            }
            if(*cliplist != DGA_X_EOL) break;
            cliplist++;
        } else {
            cliplist += 2;
            while(*cliplist != DGA_X_EOL) cliplist += 2;
            cliplist++;
        }
    }

    if(*cliplist != DGA_Y_EOL) {
        //
        //  Return what's on the screen.
        //
        Xil_unsigned8* src = fb_mem + y*fb_width + x;
        *data = *src;
    } else {
        //
        //  Data point is obscured by another window so return 0.0
        //
        *data = 0.0;
    }

    //
    //  Unlock the window...
    //
    DGA_WIN_UNLOCK(infop);
}
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(9 of 18)*

```
void
XilDeviceInputOutputCG6::display(XilImage* copy_image)
{
    //
    //  NOTE: There is code elsewhere which depends on this function handling
    //        any arbitrary 1-band XIL_BYTE image.
    //

    //
    //  Get the image origin and the child offsets.
    //
    long x_origin, y_origin;
    copy_image->getOrigin(&x_origin,&y_origin);

    unsigned int offset_x,offset_y,offset_band;
    copy_image->getChildOffsets(&offset_x,&offset_y,&offset_band);

    //
    //  Get the memory for the XIL image backing store.
    //
    XilMemoryStorageByte* storage =
        (XilMemoryStorageByte*)copy_image->getMemoryStorage();
    if(storage == NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-140",FALSE);
        return;
    }

    //
    //  Get any ROIs associated with the display image.
    //
    XilRoi* roi = copy_image->getPixelsTouchedRoi();
    if(roi == NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-6",FALSE);
        return;
    }

    //
    //  Lock the DGA window for our use.
    //
    DGA_WIN_LOCK(infop);
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(10 of 18)*

```
//
//  We grab the cliplist every time since checking it may
//  cause problems with codec molecules that go directly to the
//  display which need to know if the window has been modified.
//  It’s cheap enough to not worry.
//
cliplist = dga_win_clipinfo(infop);
dga_win_bbox(infop, &win_x, &win_y, &win_width, &win_height);

//
//  Intersect the ROI list and the window cliplist to generate the
//  actual ROI of pixels we will touch on the display.
//
XilRoi* clipped_roi = roi->intersect(cliplist,
                                     (int)(win_x+x_origin+offset_x),
                                     (int)(win_y+y_origin+offset_y));
if(clipped_roi == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-7”,FALSE);
    DGA_WIN_UNLOCK(infop);
    return;
}

//
//  Get the ROI as a list of rectangles...
//
XilRoiList* roi_list = clipped_roi->getRectList();
if(roi_list == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-8”,FALSE);
    return;
}

//
//  Take into account the child offsets
//
win_x += offset_x;
win_y += offset_y;

//
//  Operate over each rectangle.  The rectangles are guaranteed
//  not to go out outside of the image.
//
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(11 of 18)*

```
    unsigned int   x_size, y_size;
    long           x, y;
    unsigned char* src;
    unsigned char* dst;
    while(roi_list->next(&x,&y,&x_size,&y_size)) {
        //
        //  All of the rectangles must be adjusted by
        //    the image origin.
        //
        x += x_origin;
        y += y_origin;

        src = storage->data +
            y*storage->scanline_stride +
            x*storage->pixel_stride;

        dst = fb_mem +
            (win_y+y)*fb_width +
            (win_x+x);

        //
        //  If this CG6 supports double loads and stores, then we can
        //    use xil_memcpy() to put the data onto the screen.
        //    Otherwise, we must use the plain memcpy() which does not
        //    accelerate the copy by using double loads and stores.
        //
        if(doubleLoadnStore == TRUE) {
            for(int i = 0; i<y_size; i++) {
                xil_memcpy(dst,src,x_size);

                dst += fb_width;
                src += storage->scanline_stride;
            }
        } else {
            for(int i = 0; i<y_size; i++) {
                memcpy(dst,src,x_size);

                dst += fb_width;
                src += storage->scanline_stride;
            }
        }
    }
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(12 of 18)*

```
    //
    //  Unlock the DGA window.
    //
    DGA_WIN_UNLOCK(infop);

    //
    //  Destroy all of the ROIs I created.
    //
    clipped_roi->destroy();
    roi_list->destroy();
}

void
XilDeviceInputOutputCG6::capture(XilImage* copy_image)
{
    //
    //  Get the image origin and the child offsets.
    //
    long x_origin, y_origin;
    copy_image->getOrigin(&x_origin,&y_origin);

    unsigned int offset_x,offset_y,offset_band;
    copy_image->getChildOffsets(&offset_x,&offset_y,&offset_band);

    //
    //  Get the memory for the XIL image backing store.
    //
    XilMemoryStorageByte* storage =
        (XilMemoryStorageByte*)copy_image->getMemoryStorage();
    if(storage == NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-140”,FALSE);
        return;
    }

    //
    //  Get any ROIs associated with the display image.
    //
    XilRoi* roi = copy_image->getImageSpaceRoi();
    if(roi == NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-6”,FALSE);
        return;
    }
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(13 of 18)*

```
//
//  Lock the DGA window for our use.
//
DGA_WIN_LOCK(infop);

//
//  We grab the cliplist every time since checking it may have
//  cause problems with molecules that go directly to the display
//  which need to know if the window has been modified.  It’s
//  cheap enough.
//
cliplist = dga_win_clipinfo(infop);
dga_win_bbox(infop, &win_x, &win_y, &win_width, &win_height);

//
//  Intersect the ROI list and the window cliplist to generate the
//  actual ROI of pixels we will touch on the display.
//
XilRoi* clipped_roi = roi->intersect(cliplist,
                                     (int)(win_x+x_origin+offset_x),
                                     (int)(win_y+y_origin+offset_y));
if(clipped_roi == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-7”,FALSE);
    DGA_WIN_UNLOCK(infop);
    return;
}

//
//  Get the ROI as a list of rectangles...
//
XilRoiList* roi_list = clipped_roi->getRectList();
if(roi_list == NULL) {
    XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-8”,FALSE);
    return;
}

//
//  Take into account the child offsets
//
win_x += offset_x;
win_y += offset_y;
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(14 of 18)*

```
//
//  Operate over each rectangle.  The rectangles are guaranteed
//  not to go out outside of the image.
//
unsigned int   x_size, y_size;
long           x, y;
unsigned char* src;
unsigned char* dst;
while(roi_list->next(&x,&y,&x_size,&y_size)) {
    //
    //  All of the rectangles must be adjusted by
    //    the image origin.
    //
    x += x_origin;
    y += y_origin;

    src = fb_mem +
        (win_y+y)*fb_width +
        (win_x+x);

    dst = storage->data +
        y*storage->scanline_stride +
        x*storage->pixel_stride;

    //
    //  If this CG6 supports double loads and stores, then we can
    //    use xil_memcpy() to put the data onto the screen.
    //    Otherwise, we must use the plain memcpy() which does not
    //    accelerate the copy by using double loads and stores.
    //
    if(doubleLoadnStore == TRUE) {
        for(int i=0; i<y_size; i++) {
            xil_memcpy(dst, src, x_size);

            src += fb_width;
            dst += storage->scanline_stride;
        }
    } else {
        for(int i=0; i<y_size; i++) {
            memcpy(dst, src, x_size);

            src += fb_width;
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(15 of 18)*

```
                dst += storage->scanline_stride;
            }
        }
    }

    //
    //  Unlock the DGA window.
    //
    DGA_WIN_UNLOCK(infop);

    //
    //  Destroy all of the ROIs I created.
    //
    clipped_roi->destroy();
    roi_list->destroy();

    //
    //  Mark the XIL image backing storage as Valid
    //
    setMemoryValid(TRUE);
}

void
install_cmap(Dga_cmap       dga_cmap,
             int            index,
             int            count,
             Xil_unsigned8* red,
             Xil_unsigned8* green,
             Xil_unsigned8* blue)
{
    cg6_cmap* cg6cmap = (cg6_cmap*)dga_cm_get_client_infop(dga_cmap);

    //
    //  Store colors side-by-side
    //
    static Xil_unsigned8 cmap[3*256];
    for(int i=0,j=0; j<count; i+=3,j++) {
        cmap[i]   = red[j];
        cmap[i+1] = green[j];
        cmap[i+2] = blue[j];
    }
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(16 of 18)*

```
    //
    //  cg6 Cmap
    //
    cg6cmap->addr = index << 24;

    int              nument  = (((count<<1)+count)>>2);  // ncolors*3
    volatile u_int* hw_cmap = &cg6cmap->cmap;
    int*             incmap = (int*) cmap;
    for(i=0; i<nument; i++, incmap++) {
        *hw_cmap = *incmap;
        *hw_cmap = *incmap << 8;
        *hw_cmap = *incmap << 16;
        *hw_cmap = *incmap << 24;
    }
}

int
XilDeviceInputOutputCG6::setDeviceAttribute(char  attribute_name[],
                                            void* value)
{
    if(!strcmp(attribute_name, “XCOLORMAP”)) {
    XilColorList*  clist   = (XilColorList*)value;

    if(clist->cmap != xcmap) {
        if(clist->cmap != 0) {
    //
    //  UnGrab Cmap Grabber
    //
    XDgaUnGrabColormap(displayptr, xcmap);
    dga_cmap = NULL;
        }
        xcmap = 0;

        //
        //  Connect to the Cmap Grabber
        //
        Dga_token dga_token = XDgaGrabColormap(displayptr, clist->cmap);
        if(dga_token==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,”di-219”,TRUE);
    return XIL_FAILURE;
        }
        if((dga_cmap =
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(17 of 18)*

```
                ((Dga_cmap) dga_cm_grab(dga_win_devfd(infop),
                                        dga_token))) == NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-219",TRUE);
    XDgaUnGrabColormap(displayptr, clist->cmap);
    return XIL_FAILURE;
        }
        xcmap = clist->cmap;

        //
        //  Mmap Hardware
        //
        if((cg6cmap = (struct cg6_cmap*)
    mmap(NULL, CG6_CMAP_SZ, PROT_READ|PROT_WRITE,
         MAP_PRIVATE, fd, CG6_VADDR_CMAP)) ==
                (struct cg6_cmap*)-1) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-215",TRUE);
    return XIL_FAILURE;
        }

            dga_cm_set_client_infop(dga_cmap, cg6cmap);
    }

    XColor*         colors  = clist->colors;
    Xil_unsigned32 ncolors = clist->ncolors;
    unsigned long  index   = colors[0].pixel;


    //
    // Check to see if the colors are linear and convert
    //
    static Xil_unsigned8  red[256], green[256], blue[256];
    for(int i=(int)(index),j=0; i<(ncolors+index); i++,j++) {
        if(colors[i-index].pixel != i) {
    XStoreColors(displayptr, clist->cmap, colors, ncolors);
    return XIL_SUCCESS;
        }

        red[j]   = colors[j].red>>8;
        green[j] = colors[j].green>>8;
        blue[j]  = colors[j].blue>>8;
    }
```

Code Example 3-6 `XilDeviceInputOuputCG6.cc` *(18 of 18)*

```
    //
    //  Install the new colormap.
    //
        dga_cm_write(dga_cmap, (int)index, ncolors,
                     red, green, blue, install_cmap);

        return XIL_SUCCESS;
    }

    return XIL_FAILURE;
}

int XilDeviceInputOutputCG6::getDeviceAttribute(char attribute_name[],
                                                 void** value)
{
    if (!strcmp(attribute_name,"WINDOW"))
        *value= (void *)window;
    else if (!strcmp(attribute_name,"DISPLAY"))
        *value= (void *)displayptr;
    else if (!strcmp(attribute_name,"FBC"))
        *value= (void *)fb_fbc;
    else if (!strcmp(attribute_name,"DGA_WIN"))
        *value= (void *)infop;
    else
        return XIL_FAILURE;

    return XIL_SUCCESS;
}
```

Sample I/O Device Handler

The following example shows an I/O handler that treats an X11 window as a I/O device. It contains an implementation of `XilDeviceInputOutputTypeXlib` and `XilDeviceInputOutputXlib`, which are classes derived from `XilDeviceInputOutputType` and `XilDeviceInputOutput`, respectively. It also shows sample implementations of the member functions of these classes. This is the code delivered in the XIL library to allow device images to be created from X11 windows.

Code Example 3-7 `XlibCreateType.cc` *(1 of 14)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//------------------------------------------------------------------------
//
//  Description:
//  Contains the member functions of XilDeviceInputOutputTypeXlib
//      and XilDeviceInputOutputXlib.
//
//
//
//------------------------------------------------------------------------
#pragma ident  “@(#)XlibCreateType.cc1.2\t94/03/23  “

#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xil/xili.h>


/*
 * Derived instantiation of XilDeviceInputOutputType class
 * This is the description of the connection to the device
 * There is only one of these
 */
class XilDeviceInputOutputTypeXlib : public XilDeviceInputOutputType {
  public:
    virtual XilDeviceInputOutput* createDeviceInputOutput(
                 XilImage* parent, void* data);
};

/*
 * Derived instantiation of XilDeviceInputOutput class
```

Code Example 3-7 `XlibCreateType.cc` *(2 of 14)*

```
 * This is the description of a particular instantiation of the device
 * There can be many of these
 */
class XilDeviceInputOutputXlib : public XilDeviceInputOutput {
  public:
   virtual int setDeviceAttribute(char attrbute_name[], void* value);
   virtual int getDeviceAttribute(char attribute_name[], void** value);
   virtual void display(XilImage*);
   virtual void capture(XilImage*);
   virtual void getPixel(unsigned short x, unsigned short y,
                         unsigned short band, unsigned short count,
                         float *data);
   virtual void setPixel(unsigned short x, unsigned short y,
                         unsigned short band, unsigned short count,
                         float *data);

    XilDeviceInputOutputXlib(XilImage* parent, void* data);
    virtual ~XilDeviceInputOutputXlib();
  private:
    Display* displayptr;
    Window window;
    XImage* xImage;
    unsigned int x_depth;
    GC gc;
};

/*
 * This is the global function called by XIL to create this kind of
 * I/O device
 */
XilDeviceInputOutputType *XilCreateInputOutputType()
{
    return(new XilDeviceInputOutputTypeXlib());
}

/*
 * This is the routine that creates new images on this particular I/O
 * device
 */
XilDeviceInputOutput* XilDeviceInputOutputTypeXlib::createDeviceInputOutput(
    XilImage* parent, void* data)
{
```

Code Example 3-7 `XlibCreateType.cc` *(3 of 14)*

```
   XilDeviceInputOutputXlib* device;
   device=new XilDeviceInputOutputXlib(parent,data);
   if (device==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_RESOURCE,"di-1",TRUE);
      return NULL;
   }
   if (device->getImageType()==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-188",FALSE);
      delete device;
      return NULL;
   }
   return(device);
}

/*
 * Constructor for an object (image) of this device
 */
XilDeviceInputOutputXlib::XilDeviceInputOutputXlib(
    XilImage* parent,void* data)
{
   XGCValues gc_val;
   XilWindow* window;
   XilDataType depth;
   unsigned int nbands;
   unsigned int width;
   unsigned int height;

   window = (XilWindow*)data;

   /* indicate readability and writeability */
   readable= TRUE;
   writeable= TRUE;

   /* save the display and window */
   this->displayptr= window->display;
   this->window= window->window;

   /* find out the image type */
   {
      Window root;
      int x,y;
      unsigned int border_width;
```

Code Example 3-7 `XlibCreateType.cc` *(4 of 14)*

```
      Status status=XGetGeometry(displayptr,this->window,&root,&x,&y,&width,
                                 &height,&border_width, &x_depth);
      if (status==0) {
         XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-220",TRUE);
         imageType=NULL;
         return;
      }
   }
   switch (x_depth) {
    case 1:
       depth= XIL_BIT;
       nbands= 1;
       imageType=realImageType=
          parent->getSystemState()->createImageType(width,height,1,depth);
       break;
    case 4:
    case 8:
       depth= XIL_BYTE;
       nbands= 1;
       imageType=realImageType=
          parent->getSystemState()->createImageType(width,height,1,depth);
       break;
    case 16:
       depth= XIL_SHORT;
       nbands= 1;
       imageType=realImageType=
          parent->getSystemState()->createImageType(width,height,1,depth);
       break;
    case 24:
       depth= XIL_BYTE;
       nbands= 3;
       imageType=parent->getSystemState()-
>createImageType(width,height,3,depth);
       realImageType=parent->getSystemState()-
>createImageType(width,height,4,depth);
       break;
    default:
       imageType=realImageType=NULL;
       break;
   }
   if (imageType==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-188",FALSE);
```

Code Example 3-7 `XlibCreateType.cc` *(5 of 14)*

```
        return;
    }
    if (realImageType==NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-188",FALSE);
        imageType->destroy();
        imageType=NULL;
        return;
    }

    /* create a graphics context */
    gc_val.foreground= 0;
    gc_val.function= GXcopy;
    this->gc= XCreateGC(displayptr,this-
>window,GCForeground|GCFunction,&gc_val);
    if (!this->gc) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-162",TRUE);
        imageType->destroy();
        imageType=NULL;
        return;
    }

    /* create an X image */
    XWindowAttributes win_attr;
    if (!XGetWindowAttributes(displayptr,this->window,&win_attr)) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-249",TRUE);
        XFreeGC(displayptr,gc);
        imageType->destroy();
        imageType=NULL;
        return;
    }
    xImage=
XCreateImage(displayptr,win_attr.visual,win_attr.depth,ZPixmap,0,0,
                    10, 10, 8, 0);
    if (xImage==NULL) {
        XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-250",TRUE);
        XFreeGC(displayptr,gc);
        imageType->destroy();
        imageType=NULL;
        return;
    }
```

Code Example 3-7 `XlibCreateType.cc` *(6 of 14)*

```
   /* go ahead and throw way the storage that the X image created
    * since we'll be copying directly out of the XIL image
    */
   XFree(xImage->data);
   xImage->data=NULL;

   /* get the display operation numbers */
   displayOpNumber=XilLookupOpNumber("display_ioxlib");
   captureOpNumber=XilLookupOpNumber("capture_ioxlib");
}

/*
 * This is the destructor for the storage device
 */
XilDeviceInputOutputXlib::~XilDeviceInputOutputXlib()
{
   if (imageType) {
      XFreeGC(displayptr,gc);
      xImage->data=NULL;
      XDestroyImage(xImage);
      imageType->destroy();
      if (realImageType!=imageType) realImageType->destroy();
   }
}

/*
 * write image to device
 */
void XilDeviceInputOutputXlib::display(XilImage* copy_image)
{
   unsigned short width,height,nbands;
   XilDataType data_type;
   XilMemoryStorage* storage;
   long x_origin, y_origin;
   XilRoi* roi;
   XilRoiList* roi_list;
   long x,y;
   unsigned int offset_x,offset_y,offset_band;

   /* get information about the image */
   copy_image->getInfo(&width,&height,&nbands,&data_type);
   copy_image->getOrigin(&x_origin,&y_origin);
```

Code Example 3-7 `XlibCreateType.cc` *(7 of 14)*

```
/* get format information and put it into the X image*/
storage= (XilMemoryStorage*)copy_image->getMemoryStorage();
if (storage==NULL) {
   XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-140",FALSE);
   return;
}

/* get the child offset information */
copy_image->getChildOffsets(&offset_x,&offset_y,&offset_band);

switch (copy_image->getDataType()) {
 case XIL_BIT:
   xImage->data= (char*)storage->bit.data;
   xImage->bytes_per_line= (int)storage->bit.scanline_stride;
   break;
 case XIL_BYTE:
   xImage->data= (char*)storage->byte.data-offset_band;
   xImage->bytes_per_line= (int)storage->byte.scanline_stride;
   break;
 case XIL_SHORT:
   xImage->data= (char*)storage->shrt.data;
   xImage->bytes_per_line= (int)storage->shrt.scanline_stride*2;
   break;
 case XIL_FLOAT:
   XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-216",FALSE);
   return;
}
xImage->width= width;
xImage->height= height;

/* figure out the plane mask */
if (x_depth==24) {
   unsigned long mask;
   switch (copy_image->getBands()) {
    case 1:
      mask= 0xFF0000;
      break;
    case 2:
      mask= 0xFFFF00;
      break;
    case 3:
      mask= 0xFFFFFF;
```

Code Example 3-7 `XlibCreateType.cc` *(8 of 14)*

```
          break;
       }
       mask= mask >> ((offset_band-1) * 8);
       XSetPlaneMask(displayptr,gc,mask);
    }


    /* get ROI */
    roi= copy_image->getPixelsTouchedRoi();
    if (roi==NULL) {
       XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-6",FALSE);
       return;
    }

    /* get the roi list from the roi */
    roi_list= roi->getRectList();
    if (roi_list==NULL) {
       XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-8",FALSE);
       return;
    }

    /* operate on each ROI, all of the regions are guaranteed not to go outside
     * the image
     */
    unsigned int x_size,y_size;
    while (roi_list->next(&x,&y,&x_size,&y_size)) {
       x+= x_origin;
       y+= y_origin;
       XPutImage(displayptr,window,gc,xImage,(int)x,(int)y,(int)x+offset_x,
                 (int)y+offset_y,x_size,y_size);
    }

    /* se the plane mask back to normal */
    if (x_depth==24) XSetPlaneMask(displayptr,gc,0xFFFFFF);

    XFlush(displayptr);
    xImage->data= NULL;
    roi_list->destroy();
}
```

Code Example 3-7 `XlibCreateType.cc` *(9 of 14)*

```
/*
 * read image from device
 */
void XilDeviceInputOutputXlib::capture(XilImage* copy_image)
{
   unsigned short width,height,nbands;
   XilDataType data_type;
   XilMemoryStorage* storage;
   long x_origin, y_origin;
   XilRoi* roi;
   XilRoiList* roi_list;
   long x,y;
   unsigned int offset_x,offset_y,offset_band;

   /* get information about the image */
   copy_image->getInfo(&width,&height,&nbands,&data_type);
   copy_image->getOrigin(&x_origin,&y_origin);

   /* get format information and put it into the X image*/
   storage= (XilMemoryStorage*)copy_image->getMemoryStorage();
   if (storage==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-140",FALSE);
      return;
   }

   /* get the child offset information */
   copy_image->getChildOffsets(&offset_x,&offset_y,&offset_band);

   switch (copy_image->getDataType()) {
    case XIL_BIT:
      xImage->data= (char*)storage->bit.data;
      xImage->bytes_per_line= (int)storage->bit.scanline_stride;
      break;
    case XIL_BYTE:
      xImage->data= (char*)storage->byte.data-offset_band;
      xImage->bytes_per_line= (int)storage->byte.scanline_stride;
      break;
    case XIL_SHORT:
      xImage->data= (char*)storage->shrt.data;
      xImage->bytes_per_line= (int)storage->shrt.scanline_stride*2;
      break;
    case XIL_FLOAT:
```

Code Example 3-7 `XlibCreateType.cc` *(10 of 14)*

```
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-216",FALSE);
      return;
   }
   xImage->width= width;
   xImage->height= height;

   /* get ROI */
   roi= copy_image->getImageSpaceRoi();
   if (roi==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-6",FALSE);
      return;
   }

   /* get the roi list from the roi */
   roi_list= roi->getRectList();
   if (roi_list==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-8",FALSE);
      return;
   }

   /* operate on each ROI, all of the regions are guaranteed not to go outside
    * the image
    */
   unsigned int x_size,y_size;
   while (roi_list->next(&x,&y,&x_size,&y_size)) {
      x+= x_origin;
      y+= y_origin;
      XGetSubImage(displayptr,window,(int)x+offset_x,(int)y+offset_y,
                   x_size,y_size,0xffffffff,ZPixmap,xImage,(int)x,(int)y);
   }
   XFlush(displayptr);
   xImage->data= NULL;
   roi_list->destroy();
}

void XilDeviceInputOutputXlib::setPixel(unsigned short x, unsigned short y,
                                         unsigned short band, unsigned short count,
                                         float* data)
{
   float value;
   unsigned long pixel;
   unsigned long mask;
```

Code Example 3-7 `XlibCreateType.cc` *(11 of 14)*

```
/* set the pixel */
switch (x_depth) {
 case 1:
   value= data[0]+.5;
   if (value > 1.0) {
      pixel=1;
   } else if (value < 0.0) {
      pixel=0;
   } else {
      pixel=(unsigned long)value;
   }
   mask=1;
   break;
 case 4:
 case 8:
   value= data[0]+.5;
   if (value > 255.0) {
      pixel=255;
   } else if (value < 0.0) {
      pixel=0;
   } else {
      pixel=(unsigned long)value;
   }
   mask=0xFF;
   break;
 case 24:
   pixel=0;
   switch (count) {
    case 3:
      value= data[2];
      if (value > 255.0) {
         pixel|=0xFF0000;
      } else if (value < 0.0) {
      } else {
         pixel |= (unsigned long)value << 16;
      }
    case 2:
      value= data[1];
      if (value > 255.0) {
         pixel|=0xFF00;
      } else if (value < 0.0) {
      } else {
```

Code Example 3-7 `XlibCreateType.cc` *(12 of 14)*

```
            pixel |= (unsigned long)value << 8;
         }
       case 1:
         value= data[0];
         if (value > 255.0) {
            pixel|=0xFF;
         } else if (value < 0.0) {
         } else {
            pixel |= (unsigned long)value;
         }
      }
      switch (count) {
       case 1:
         mask= 0xFF;
         break;
       case 2:
         mask= 0xFFFF;
         break;
       case 3:
         mask= 0xFFFFFF;
         break;
      }
      mask = mask << (band * 8);
      pixel = pixel << (band * 8);
   }
   XSetPlaneMask(displayptr,gc,mask);
   XSetForeground(displayptr,gc,pixel);
   XDrawPoint(displayptr,window,gc,x,y);
   XSetPlaneMask(displayptr,gc,0xffffffff);
   XFlush(displayptr);
}

void XilDeviceInputOutputXlib::getPixel(unsigned short x, unsigned short y,
                                  unsigned short band, unsigned short count,
                                  float* data)
{
   unsigned long pixel;
   XImage* image;

   image=XGetImage(displayptr,window,x,y,1,1,0xFFFFFF,ZPixmap);
   if (!image) {
      XIL_ERROR(NULL,XIL_ERROR_SYSTEM,"di-316",TRUE);
```

Code Example 3-7 `XlibCreateType.cc` *(13 of 14)*

```
      return;
   }
   pixel= XGetPixel(image,0,0);
   XDestroyImage(image);

   switch (x_depth) {
    case 1:
    case 4:
    case 8:
      data[0]=pixel;
      break;
    case 24:
      pixel >> (band * 8);
      switch (count) {
       case 3:
         data[2]= (pixel & 0xFF) >> 16;
         /* break intentionally omitted */
       case 2:
         data[1]= (pixel & 0xFF) >> 8;
         /* break intentionally omitted */
       case 1:
         data[0]= pixel & 0xFF;
         /* break intentionally omitted */
      }
      break;
   }
}


int XilDeviceInputOutputXlib::setDeviceAttribute(char attribute_name[],
                                                  void* value)
{
    if (!strcmp(attribute_name,"XCOLORMAP")) {
    XilColorList* clist = (XilColorList*)value;
    XStoreColors(displayptr, clist->cmap, clist->colors, clist->ncolors);
        return XIL_SUCCESS;
    }
    return XIL_FAILURE;
}

int XilDeviceInputOutputXlib::getDeviceAttribute(char attribute_name[],
                                                 void **value)
```

Code Example 3-7 `XlibCreateType.cc` *(14 of 14)*

```
{
  if (!strcmp(attribute_name,"WINDOW"))
    *value=(void *)window;
  else if (!strcmp(attribute_name,"DISPLAY"))
    *value=(void *)displayptr;
  else
    return XIL_FAILURE;

  return XIL_SUCCESS;
}
```

Compute Devices 4

About Compute Devices

Compute handlers are the basic operation grouping in the XIL library. A single compute handler can contain the implementation of one or more atomic XIL image functions and/or one or more molecules. (General aspects of molecules are described in Chapter 1, “Overview.”). Compute handlers are device specific; they provide an implementation of the operational capability of a device. In addition, compute handlers integrate with storage handlers (see Chapter 5, “Storage Devices”) and with I/O handlers (see Chapter 3, “I/O Devices”). They also are used to implement the compress and decompress part of XIL compression and decompression (see Chapter 6, “Compression/Decompression”).

Each compute handler must contain the following global function:

```
XilDeviceComputeType* XilCreateComputeType()
```

This function provides device initialization and describes the list of implemented image processing functions that need to be added to the global list of available functions.

`XilDeviceComputeType` is an empty class. Unlike I/O, compression, and storage devices, no instances of a compute device exist. The compute handler subclasses `XilDeviceComputeType`, adding the functions that have been implemented. The default *memory* device has all the functions implemented. In the derived class, you need only implement any additional functions you desire.

4

Implementing an XIL Function

To create a compute device, you must implement an XIL function. There are several examples of this in this document. Appendix A, "Sample Molecule," shows an example of creating a two-atom molecule. The section "Sample Compute Device Handler" on page 134," shows how to create an atom that uses a different storage mechanism from the standard memory storage. The functions that may be implemented are listed in Appendix B, "XIL Atomic Functions."

The language for implementing XIL functions is C++. All image functions are members of the `XilImage` class. Class members, such as `XilImage::getBands()`, are available for use in compute handlers.

The arguments to each element (atomic or molecular) in a compute handler are the `op` and the `op_count`. For the implementation in Appendix A, "Sample Molecule," the prototype looks like this:

```
int
XilDeviceComputeTypeMemory::Rescale16Convert16to8(
    XilOp* op,      // a pointer into the DAG
    int op_count)  // the number of combined ops to be done
```

The `XilOp` class holds the information required to store the XIL operation in the DAG. The `op` parameter is a pointer that represents a specific XIL operation on the DAG. The source and destination images of atomic functions must be accessible to the operation. These images are stored in the `XilOp` object. The parameters of XIL functions also are stored in the `XilOp` object. The `XilOp` class contains member functions that enable you to extract the image and parameter information for an operation. See the section "The XilOp Class" in Chapter 1 for a complete description of this class. The number of image sources supported by an XIL operation and the `XilOp` member functions that you must use to extract the images sources and to extract an XIL function's parameters from the `XilOp` object are listed in Appendix C, "XilOp Object."

Access to the image data is obtained via three member functions of the `XilImage` class:

Table 4-1 Image Data Memory Functions of the `XilImage` Class

Member Function	**Definition**
`getStorage()`	Returns a pointer to a structure that describes the data storage for the specified storage type. It does not take child offsets into account.
`getMemoryStorage()`	Returns a pointer that describes the memory layout. It does not allow you to specify the storage type. It does take child offsets into account.
`requestStorage()`	Returns a pointer to a structure that describes the data storage for the specified storage, but, unlike `getStorage`(), does not force a propagation.

These member functions are discussed in detail in the section “The XilImage Class” in Chapter 1.

An example of the use of `getMemoryStorage`() is as follows:

```
// get source’s memory storage
XilMemoryStorageShort *short_storage;
short_storage = (XilMemoryStorageShort *)src->getMemoryStorage();
if (short_storage==NULL) {
    return XIL_FAILURE;
}
Xil_signed16 *src_base_addr = (Xil_signed16 *)short_storage->data;
unsigned long src_next_pixel = short_storage->pixel_stride;
unsigned long src_next_scan = short_storage->scanline_stride;
```

Almost all image operations in the XIL library will be affected by regions of interest (ROIs). XIL ROIs may be of arbitrary shape. The internal representation in the current release is a series of rectangles, much like the X11 region, except that the XIL library uses 32-bit quantities instead of 16-bit quantities to define the position and extent of the rectangles. The internal function `XiliGetRoiList()` can be used to obtain the intersection of the source and destination ROIs. For a detailed discussion of ROIs, see the section “The XilRoi Class” in Chapter 1.

Note – The values produced by the implementation of an XIL function should match as closely as possible the values produced by the memory port. This has several implications. The results of operations should be clamped; that is, intermediate results should be tested against the maximum and minimum values of the destination data type before they are converted to that data type. For many of the simple functions, any difference from the default version should not be tolerated. More complicated operations, where there are many floating-point or fixed-point calculations done for each pixel, do not always allow pixel-for-pixel accuracy with any sort of reasonable code. Often, new algorithms provide slightly different values. It is up to the implementor of the algorithm to make sure that there are no *systematic* differences between the new implementation and the old one.

The XIL Test Suite can aid you in verifying new implementations of XIL functions. The XIL Test Suite enables you to perform regression tests of new code against verified reference signatures and includes the capability of specifying a tolerance for the comparison. This test suite is described in a separate document, *XIL Test Suite User's Guide*, which is part of this release.

Error handling in an implementation is performed by calling the `notifyError()` member function of the system state. A macro `XIL_ERROR` that simplifies this interface is defined in `XilError.h`. Both compute handler examples use this interface. The method used to add error messages for device-dependent errors is discussed in Chapter 2, "More on Writing Device Handlers."

Adding a Compute Device

When writing a compute device, you can implement support for one or more atomic or molecular functions. The default (memory) compute device contains implementations of all the XIL atomic functions, as well as a few molecules.

Each subclass of a compute handler can implement multiple atomic or molecular functions. Each member function (routine) of a subclass can be an entry point for multiple atoms or molecules. Each routine can perform equivalent functions or can contain common code that branches out for specific parameter values. For example, in the code below, `Affine8` is a common entry point for several types of affine operations on the byte data type.

```
/* XILCONFIG: Affine8= affineNN8() */
/* XILCONFIG: Affine8= affineBL8() */
/* XILCONFIG: Affine8= affineBC8() */
XilDeviceComputeTypeMemory::Affine8(
    XilOp   *op,        // a pointer into the DAG
    int     op_count)   // number of combined operations
{
            .
            .
}
```

Molecules also can share a common entry point. In the example below, the `DecompressDither8` treats the rescale as an optional operation, which can be replaced with some reasonable default value. Therefore, this routine must be written to handle a chain of two or three operations. The code checks the operation number of the second operation using the utility function `XilLookupOpNumber()`, which returns the operation number for a specified routine name.

```
/* XILCONFIG: DecompressDither8= */
    ordereddither8_8(decompress_Codec())
/* XILCONFIG: DecompressDither8= */
    ordereddither8_8(rescale(decompress_Codec())

int
XilDeviceComputeTypeCodecMemory::DecompressDither8(
    XilOp*      *op,        // a pointer into the DAG
    int         op_count)   // number of combined operations
{
    // Pull everything needed off the chain of operations
    // First, the ordered dither operation
    dst = op->getDst();
    cube = (XilLookup *)op->getObjParam(1);
    dmask = (XilDitherMask *)op->getObjParam(2);
```

```
    // move to next operation along the chain
    op = op->getOp1();

    // Now, look at the next operation number and test for the
    // optional rescale operation
    rescale8_opnum == XilLookupOpNumber("rescale8")
    if (op->getOp() == rescale8_opnum) {
        rescale = (float *)op->getPtrParam(1);
        offset = (float *)op->getPtrParam(2);
        // move to next operation along the chain
        op = op->getOp1();
    } else {
        // default values for optional rescale
        rescale = 0;
        offset = 0;
    }

    // Now, the decompress operation
    dc = (XilDeviceCompressionCodec*)
        (op->getSrcCis())->getDeviceCompression();
    dc->seek((int)op->getLongParam(1));
            .
            .
}
```

The next example illustrates that you can have multiple routines in the same file. And, a file can contain both atoms and molecules and the `XILCONFIG` lines can be in any order (meaning they do not have to be in the order the routines are presented in the file).

```
/* XILCONFIG: SubtractAdd8 = add8(subtract8()) */
/* XILCONFIG: Add8 = add8() */
/* XILCONFIG: MultiplyAdd8 = add8(multiply8()) */

XilDeviceComputeTypeMemory::Add8(XilOp *op, int op_count)
{
            .
            .
}
```

```
XilDeviceComputeTypeMemory::MultiplyAdd8(XilOp *op, int
op_count)
{
            .
            .
}

XilDeviceComputeTypeMemory::SubtractAdd8(XilOp *op, int
op_count)
{
            .
            .
}
```

The compute device implementation follows these steps:

1. Subclass the `XilDeviceComputeType` class, including the functions for this specific compute device. As an example, consider the case of a device that can add 8-bit images. The implementation would include a subclass like:

```
class XilDeviceComputeTypeMyDevice: public XilDeviceComputeType{
    public:
        int MyAdd8(XilOp* op, int count);
};
```

2. After the implementation is compiled, link it into the execution table so that it will be referenced properly by the core code. The XIL library provides a nearly automatic means of doing this. For you to use this method, the implementation file must contain a comment line of the following type:

   ```
   /* XILCONFIG: MyAdd8 = add8() */
   ```

   The `add8()` indicates the XIL atom that the implementation `MyAdd8` represents. Remember that multiple atoms and/or molecules may be implemented within a single routine and within a single file. For each atom and molecule, an appropriate `XILCONFIG` line must be present. The atomic functions available for implementing are described in Appendix B, "XIL Atomic Functions."

3. After you have added the configuration lines, run the executable `xilcompdesc`. This executable automatically generates a file that, when compiled, will provide a routine to correctly integrate the implementation of the subclass into the XIL execution table. The executable

   `$XILHOME/bin/xilcompdesc` *classname* [*files*]

   parses the given files (or the standard input) looking for the configuration lines described above. It produces a C++ source file on standard output that is the implementation of a member function of the given class called `describeMembers()`. `describeMembers()` should be invoked inside `XilCreateComputeType()`. When invoked, `describeMembers()` adds all the member functions from this compute handler into the available function tree. For example,

```
// routine called by the XIL core to initialize the band_memory
// compute device
XilDeviceComputeType* XilCreateComputeType()
{
    XilDeviceComputeTypeBandMemory* device;

    // create an instantiation of the device
    device= new XilDeviceComputeTypeBandMemory();

    // register with the core the functions that this device
    // implements
    if (device->describeMembers()==XIL_FAILURE) {
        delete device;
        return NULL;
    } else {
        return device;
    }
}
```

4. Finally, there is a file `/opt/SUNWits/Graphics-sw/xil/lib/xil.compute` that lists the compute handlers and their dependencies (see Chapter 2, “More on Writing Device Handlers,” for more information on

installing handlers). You must edit this file to give the library instructions on the order in which to load the compute modules (see the following section “Loading Compute Handlers”). The format looks like this:

```
computememory
computeSUNWgx ioSUNWgx
computeMYCOMPANYmyhandler
```

Loading Compute Handlers

As part of `xil_open()`, the library parses the `xil.compute` file, looking for the appropriate modules in `/opt/SUNWits/Graphics-sw/xil/lib`. The name of the compute handler is listed first, followed by any other handlers on which the compute handler is dependent. The compute handlers that have no dependents are loaded as they are reached in the `xil.compute` file.

In the following example, the first line loads all the functions in `xilcomputememory.so`, and the second line has a dependent, so `xilcomputeSUNWgx.so` and `xilioSUNWgx.so` are not loaded. The third line loads `xilcomputeMYCOMPANYmyhandler.so.`

```
computememory
computeSUNWgx ioSUNWgx
computeMYCOMPANYmyhandler
```

If this handler contains, for example, the `add8()` function, this new version replaces the existing memory version (which was loaded by the first line) in the function tree. Actually, each node in the function tree is kept as a linked list of function pointers. When the new version of `add8()` is encountered, it is simply added to the head of the list. When the `add8()` atom occurs in the course of processing, the first function in the list is called. If, for some reason, the function returns `XIL_FAILURE`, the second function in the list is called, and so forth down the list, until the memory version is called. This strategy allows a function implementation to limit the range of parameters for which it works, leaving undesired parameter sets to the memory version, which will work for all legal parameters.

Let’s consider another example with compute handlers that have dependents:

```
computeSUNWgx  ioSUNWgx
computeCell_SUNWgx Cell ioSUNWGX
```

The compute handlers that have dependents are loaded when all the dependents have been loaded. For instance, the `computeSUNWgx` handler is dependent on the `ioSUNWgx` handler. When this I/O handler is loaded and initialized, then the compute handlers in `xil.compute` that have `ioSUNWgx` as their remaining dependency are loaded. Since `ioSUNWgx` is the only dependent of `computeSUNWgx`, this compute handler is loaded at that time. For example,

```
xil_create_from_device(state, “ioSUNWgx”, NULL);
// causes ioSUNWgx handler to be loaded
// once the I/O handler is loaded, then computeSUNWgx is loaded
```

The `computeCell_SUNWgx` handler still has `Cell` as a dependent, so it would not be loaded yet.

Adding a New Molecule

The XIL core code determines the function that will be called for each combination of atomic functions. The first thing you must do when adding a new molecule is to decide what the molecule is to do.

XIL supports both single and multiple branch molecules. Single branch molecules can be described in the form:

```
function_1(function_2(...(function_N())...))
```

An example of code that can be written as a molecule is:

```
xil_add(im1, im2, im3)
xil_add(im3, im4, im5)
```

This code can be rewritten as a molecule of the form `xil_add(xil_add())`

The output of the first `add` is used as an input to the second one.

An example of code that can be written as a multiple branch molecule is:

```
xil_add(im1,im2,im3)
xil_add(im4,im5,im6)
xil_subtract(im3,im6,im7)
```

Both sources in the subtract operation have dependencies. The left branch, which branches into source1 of the subtract operation, is the first add operation. The right branch, which branches into source2 of the subtract operation, is the second add operation.

The routines use the existing interface `getOp1()` to follow the left-most branch of the chain `op` operations. Two new interfaces on the `XilOp` class exist to access the right branch chain of ops: `getOp2()`, associated with source2, and `getOp3()`, associated with source3. See the section "Manipulating Molecules" in this chapter for more information.

Also, you can have a linear molecule along the right branch of the chain of operations. For example,

```
xil_add(im1,im2,im3);
xil_subtract(im4,im3,im5);
```

This code can be rewritten as a molecule of the form:

`xil_subtract(,xil_add())`

Adding a molecule is one example of adding a compute device. In the file that contains the implementation of a single branch molecule, the following configuration line must be included:

`/* XILCONFIG:` *SingleBranchMoleculeName* `=` *atom2*`(`*atom1*`()) */`

where *atom2* and *atom1* refer to existing atomic functions listed in Appendix B, "XIL Atomic Functions."

In the file that contains the implementation of a multiple branch molecule, the following configuration line must be included:

```
/* XILCONFIG: MultiBranchMoleculeName = atom1(atom2(),atom3()) */
```

In the file that contains the implementation of a linear molecule along the right branch of the chain of operations, the following configuration line must be included:

```
/* XILCONFIG: RightBranchMoleculeName = atom1(,atom2);
```

The configuration is done using `xilcompdesc` in the manner described previously for other compute handlers. The file `/opt/SUNWits/Graphics-sw/xil/lib/xil.compute` must be updated in order for the compute handler to be loaded.

Like other compute handlers, molecules may execute on processors other than the host CPU. If it would be advantageous to allow the image data to remain in nonhost memory, a storage handler for the device is required. Storage device handlers are described in Chapter 5, "Storage Devices."

Appendix A, "Sample Molecule," shows an example of a molecule made up of `rescale16` followed by `convert16to8`, using the standard memory storage handler. It is important to note that, like new atoms, molecules should strive to give the same answers as would be obtained through the atomic path. As an example, consider a molecule `subtract_const8(add_const8())` that first adds a constant value to an 8-bit image and then subtracts away a possibly different constant value. It might seem reasonable to simply subtract the two constants from each other and add the difference to the source image. However, this could cause the molecule to behave differently from the atomic path. If the initial add causes the image values to exceed the maximum value for the data type (255), the result of the initial add must be clamped to that maximum value. Subtracting the second constant would subtract from 255, not from the source value plus the first constant. Since either the molecular or atomic path may be taken, depending on how the application is written, it is vital that they behave alike.

Manipulating Molecules

A molecule is a chain of atomic operations. For a molecule, the chain must be properly followed to extract in a logical order the parameters and images from the `XilOp` object. The `op_count` parameter is useful to determine how many atomic operations exist along the chain, in case molecules of varying lengths share the same routine.

The `op` passed to the routine is the `op` associated with the last operation in the chain (the operation that writes its output to a destination image). The molecule must extract the destination image and the function's parameters from the `XilOp` object. Then, to move up the chain of operations, you can use the following functions:

- `getOp1()`, get pointer to op that has source1 as its destination
- `getOp2()`, get pointer to op that has source2 as its destination
- `getOp3()`, get pointer to op that has source3 as its destination

For information about the `XilOp` class, see the section "The XilOp Class" in Chapter 1.

Molecules and I/O Devices

I/O handlers may have their `capture()` or `display()` functions included as part of a molecule. For example, the final destination of a decompression molecule might be the display to enable digital video.

Note – Whenever the device is the final step in a molecule, it is important to mark the buffer memory as invalid by calling `setMemoryValid(FALSE)` from the device. Subsequent writes to the device may cause a `capture()` from the device to set the buffer.

In this case, the compute handler containing the molecule has a dependency on the I/O handler, since the latter contains the information to initialize the I/O device and contains the attributes. The `xil.compute` file contains this dependency information. The compute handlers that have dependents are loaded when all the dependents have been loaded.

Consider the following example:

```
computememory
computeSUNWgx ioSUNWgx
computeMYCOMPANYmyhandler
computeMYCOMPANYmyotherhandler ioMYCOMPANYmyiodevice
```

The `computeSUNWgx` handler contains several molecules that depend on the `gx` I/O handler. Before the `computeSUNWgx` handler is loaded, the `ioSUNWgx` handler must be loaded and initialized. The `computeMYCOMPANYmyotherhandler` handler contains molecules that depend on `ioMYCOMPANYmyiodevice` I/O handler. Before the `computeMYCOMPANYmyotherhandler` handler is loaded, the `ioMYCOMPANYmyiodevice` handler must be loaded and initialized.

Sample Compute Device Handler

This example illustrates a compute handler for images that are stored in a band-sequential format. The atomic function that the compute handler implements is `add8()`, the arithmetic addition of two 8-bit images. This compute handler requires a corresponding storage handler that stores images in band-sequential format. The example at the end of Chapter 5, "Storage Devices," illustrates such a handler. The files in this example include:

- `XilDeviceComputeTypeBandMemory.h` and `XilDeviceComputeTypeBandMemory.cc`, which describe and implement the compute handler classes
- `Add8BandMemory.cc`, which is the band-sequential implementation of `add8()`
- `band_memory_utils.cc`, which contains utility functions for handling child images

XilDeviceComputeTypeBandMemory.h

Code Example 4-1 XilDeviceComputeTypeBandMemory.h

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

class XilDeviceComputeTypeBandMemory : public XilDeviceComputeType {
 public:
   // constructor
   XilDeviceComputeTypeBandMemory()
   : XilDeviceComputeType("XilDeviceComputeBandMemory") {};

   // destructor
   ~XilDeviceComputeTypeBandMemory();

   // local describeMembers routine
   int describeMembers();

   // routines that this compute device implements
   int Add8(XilOp* op, int count);
};
```

XilDeviceComputeTypeBandMemory.cc

Code Example 4-2 XilDeviceComputeTypeBandMemory.cc

```
//
// foo
//

#include <xil/xili.h>
#include "XilDeviceComputeTypeBandMemory.h"

XilDeviceComputeTypeBandMemory::~XilDeviceComputeTypeBandMemory() {}

// routine called by the XIL core to initialize the band_memory
// compute device
XilDeviceComputeType* XilCreateComputeType()
{
   XilDeviceComputeTypeBandMemory* device;

   // create an instantiation of the device
   device= new XilDeviceComputeTypeBandMemory();
   if (device==NULL) {
      XIL_ERROR(NULL,XIL_ERROR_RESOURCE,"di-1",TRUE);
      return NULL;
   }

   // register with the core the functions that this device implements
   if (device->describeMembers()==XIL_FAILURE) {
      delete device;
      return NULL;
   } else {
      return device;
   }
}
```

Add8BandMemory.cc

Code Example 4-3 Add8BandMemory.cc *(1 of 6)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-------------------------------------------------------------------------
//
//  File:      Add8BandMemory.cc
//  Project:   XIL
//  Created:   93/04/30
//  RespEngr:  Chuck Mosher
//  Revision:  1.2
//  Last Mod:  11:34:47, 23 Mar 1994
//
//  Description:
//
//  This routine performs the arithmetic addition of 2 8-bit images
//  that are in band-sequential memory format
//
//
//-------------------------------------------------------------------------
#pragma ident “@(#)Add8BandMemory.cc1.2\t94/03/23  “

#include <xil/XilDefines.h>
#include <xil/XilError.h>
#include <xil/XilImage.h>
#include <xil/XilOp.h>
#include <xil/XilRoi.h>
#include <xil/XilRoiList.h>

#include “../storage_device_handler/XilBandMemoryDefines.h”

//
// Class declaration of this particular compute function
//
class XilDeviceComputeTypeBandMemory : public XilDeviceComputeType {
 public:
   int Add8(XilOp* op, int count);
   ~XilDeviceComputeTypeBandMemory();
};
//
// Global function which returns band-sequential storage information.
// This could also be handled locally by the compute functions.
//
```

Code Example 4-3 `Add8BandMemory.cc` *(2 of 6)*

```
XilBandMemoryStorage *getBandMemoryStorage(XilImage *);

//
// Line which describes which atomic function (or set of functions) that
// this routine implements.  The utility function “xilcompdesc” will cause
// this description to be included in the local version of describeMembers.cc

/* XILCONFIG: Add8 = add8() */

//
// The compute routine
//
XilDeviceComputeTypeBandMemory::Add8(
    XilOp *op,    // a pointer into the DAG.
    int)          // unused parameter (number of combined ops to be done).
{
   /* region location */
   long x,y;

   /* region size */
   unsigned int x_size,y_size;

   /* the images */
   XilImage *src1,*src2,*dest;

   /* the base addresses */
   Xil_unsigned8 *src1_base_addr,*src2_base_addr,*dest_base_addr;

   /* pointer to the current band */
   Xil_unsigned8 *src1_band,*src2_band,*dest_band;

   /* pointer to the current scanline */
   Xil_unsigned8 *src1_scanline,*src2_scanline,*dest_scanline;

   /* pointer to the current pixel */
   Xil_unsigned8 *src1_pixel,*src2_pixel,*dest_pixel;

   /* next band offset (in bytes), range=1..65535 */
   unsigned long src1_next_band,src2_next_band,dest_next_band;

   /* next scanline offset (in bytes), range=1..(65535*255) */
   unsigned long src1_next_scan,src2_next_scan,dest_next_scan;
```

Code Example 4-3 `Add8BandMemory.cc` *(3 of 6)*

```
/* x offset of image origin, range=1..65535 */
long src1_x_origin,src2_x_origin,dest_x_origin;

/* y offset of image origin, range=1..65535 */
long src1_y_origin,src2_y_origin,dest_y_origin;

/* the number of bands, range=1..255 */
unsigned int nbands;

/* loop counters */
unsigned int pixel_count, scanline_count, band_count;

/* get information about source 1 */
src1= op->getSrc1();
src1->getOrigin(&src1_x_origin,&src1_y_origin);

/* get source 1's memory storage */
XilBandMemoryStorageByte *storage;
storage= (XilBandMemoryStorageByte *)getBandMemoryStorage(src1);
if (storage==NULL) {
   return XIL_FAILURE;
}
src1_base_addr= (Xil_unsigned8 *)storage->data;
src1_next_band= storage->band_stride;
src1_next_scan= storage->scanline_stride;
delete storage;

/* get information about source 2 */
src2= op->getSrc2();
src2->getOrigin(&src2_x_origin,&src2_y_origin);

/* get source 2's memory storage */
storage= (XilBandMemoryStorageByte *)getBandMemoryStorage(src2);
if (storage==NULL) {
   return XIL_FAILURE;
}
src2_base_addr= (Xil_unsigned8 *)storage->data;
src2_next_band= storage->band_stride;
src2_next_scan= storage->scanline_stride;
delete storage;
/* get information about the destination */
dest= op->getDst();
```

Code Example 4-3 `Add8BandMemory.cc` *(4 of 6)*

```
    dest->getOrigin(&dest_x_origin,&dest_y_origin);

    /* get the destination’s memory storage */
    storage= (XilBandMemoryStorageByte *)getBandMemoryStorage(dest);
    if (storage==NULL) {
       return XIL_FAILURE;
    }
    dest_base_addr= (Xil_unsigned8 *)storage->data;
    dest_next_band= storage->band_stride;
    dest_next_scan= storage->scanline_stride;
    delete storage;

    /* get the number of bands (same for all images) */
    dest->getDimensions(NULL,NULL,&nbands);

   /* get the ROI (and the list of rectangles) that comprises the intersection
       of the ROIs of src1, src2, and dest */
    XilRoi* roi;
    XilRoiList* roi_list= XiliGetRoiList(&roi,dest,src1,src2);
    if (roi_list==NULL) {
       XIL_ERROR(src1->getSystemState(), XIL_ERROR_SYSTEM, “di-12”, FALSE);
       return XIL_FAILURE;
    }


    /*
     *  Now that we’ve intersected to determine the pixels that will
     *  be touched in the destination, set the pixelsTouchedRoi on
     *  the image.
     */
    dest->setPixelsTouchedRoi(roi);
    dest->setPixelsTouchedRoi_flag(TRUE);

    /* operate on each ROI, all of the ROI’s are guaranteed not to go outside
     * any of the images
     */
    while (roi_list->next(&x,&y,&x_size,&y_size)) {
       // adjust starting addresses to take image origins into account
       src1_band= src1_base_addr+((y+src1_y_origin)*src1_next_scan)+
                      ((x+src1_x_origin));
       src2_band= src2_base_addr+((y+src2_y_origin)*src2_next_scan)+
                      ((x+src2_x_origin));
```

Code Example 4-3 `Add8BandMemory.cc` *(5 of 6)*

```
    dest_band= dest_base_addr+((y+dest_y_origin)*dest_next_scan)+
                   ((x+dest_x_origin));

    band_count = nbands;
    do { /* each band */

       /* point to the first scanline of the band */
       src1_scanline= src1_band;
       src2_scanline= src2_band;
       dest_scanline= dest_band;

       scanline_count=y_size;
       do { /* each scanline */

          /* point to the first pixel of the scanline */
          src1_pixel= src1_scanline;
          src2_pixel= src2_scanline;
          dest_pixel= dest_scanline;

          pixel_count= x_size;
          do { // each pixel

              /* result cannot be greater than MAXBYTE */
              int result = (int)(*src1_pixel + *src2_pixel);
  *dest_pixel = ((result>>8) ? (0xff) : (result & 0xff));

             /* move to next data element */
             src1_pixel++;
             src2_pixel++;
             dest_pixel++;
          } while (--pixel_count);

          /* move to the next scanline */
          src1_scanline+=src1_next_scan;
          src2_scanline+=src2_next_scan;
          dest_scanline+=dest_next_scan;
       } while (--scanline_count);

       /* move to the next band */
       src1_band+=src1_next_band;
       src2_band+=src2_next_band;
       dest_band+=dest_next_band;
```

Code Example 4-3 `Add8BandMemory.cc` *(6 of 6)*

```
        } while (--band_count);
    }

    /* get rid of the roi_list
     * (the roi stored in dest “pixelsTouchedRoi”
     * will be destroyed by the xil core)
     */
    roi_list->destroy();

    return(XIL_SUCCESS);
}
```

band_memory_utils.cc

Code Example 4-4 band_memory_utils.cc *(1 of 3)*

```
//
//  Utility routine(s) that are needed by the band_memory compute routines
//
#include <xil/XilDefines.h>
#include <xil/XilError.h>
#include <xil/XilImage.h>
#include <xil/XilOp.h>

#include “../storage_device_handler/XilBandMemoryDefines.h”


//
//  Device storage information returned by the getStorage() function
//  is always with respect to the parent image.  IHVs must write a
//  routine similar to this one to handle the case of child images.
//  This routine adjusts the image data pointer and offset information
//  in case the image being requested is a child.
//
XilBandMemoryStorage*
getBandMemoryStorage(XilImage *image)
{
    XilBandMemoryStorage  *storage;
    XilBandMemoryStorage  *parent_storage;

    // get the parent storage description
    parent_storage= (XilBandMemoryStorage*)image->getStorage(“band_memory”);
    if (parent_storage==NULL) {
        return NULL;
    }

    //
    // Allocate the storage description to return to the compute routine.
    // The compute routine will then be responsible for deleting it.
    //
    // Note that this is done differently in the XIL memory storage driver
    // and associated memory compute routines (the memory storage driver
    // just passes back a reference that the compute routine does not delete).
    // It is up to the IHV writing these functions to decide how they want
    // to implement this.
    //
```

Code Example 4-4 band_memory_utils.cc *(2 of 3)*

```
    storage = new XilBandMemoryStorage;
    if (storage==NULL) {
        return NULL;
    }

    //
    //  Get image offset information
    //
    unsigned int offsetX, offsetY, offsetBand;
    image->getChildOffsets(&offsetX,&offsetY,&offsetBand);

    if(offsetX || offsetY || offsetBand) {
    //
        //  If this is a child, take offsets into account
    //
        XilDataType datatype= image->getDataType();
        switch (datatype) {
          case XIL_BIT:
           storage->bit.scanline_stride= parent_storage->bit.scanline_stride;
            storage->bit.band_stride= parent_storage->bit.band_stride;
            storage->bit.offset= (unsigned char)
              (((unsigned long)parent_storage->bit.offset +
           offsetX)%(unsigned long)8);
            storage->bit.data= parent_storage->bit.data +
                    parent_storage->bit.band_stride*offsetBand +
                    parent_storage->bit.scanline_stride*offsetY  +
                offsetX;
            break;

          case XIL_BYTE:
         storage->byte.scanline_stride= parent_storage->byte.scanline_stride;
            storage->byte.band_stride= parent_storage->byte.band_stride;
            storage->byte.data= parent_storage->byte.data +
                    parent_storage->byte.band_stride*offsetBand +
                    parent_storage->byte.scanline_stride*offsetY  +
                offsetX;
            break;

          case XIL_SHORT:
         storage->shrt.scanline_stride= parent_storage->shrt.scanline_stride;
            storage->shrt.band_stride= parent_storage->shrt.band_stride;
            storage->shrt.data= parent_storage->shrt.data +
```

Code Example 4-4 `band_memory_utils.cc` *(3 of 3)*

```
                     parent_storage->shrt.band_stride*offsetBand +
                     parent_storage->shrt.scanline_stride*offsetY  +
                 offsetX;
             break;

           case XIL_FLOAT:
             storage->flt.scanline_stride= parent_storage->flt.scanline_stride;
             storage->flt.band_stride= parent_storage->flt.band_stride;
             storage->flt.data= parent_storage->flt.data +
                     parent_storage->flt.band_stride*offsetBand +
                     parent_storage->flt.scanline_stride*offsetY  +
                 offsetX;
             break;
         }

     }
     else {
     //
     // can just copy the parent description
     //
         memcpy(storage, parent_storage, sizeof(XilBandMemoryStorage));
     }

     return(storage);
}
```

Storage Devices 5

About Storage Devices

Storage device handlers allow images to reside in places besides host CPU memory or in a format different from the standard memory layout. They are always associated with a compute device, allowing an accelerator to take advantage of image data remaining local to the accelerator during sequential function calls. Since accelerators usually have their own idea of how image data memory is laid out, storage handlers allow reformatting of data as it is copied between devices.

The handlers for storage devices are responsible for allocating, deallocating, and describing the data format of the storage on their device. A particular function from a compute device will request the image in a specific storage type (format) via a call to the `getStorage()` member of the `XilImage` class. The storage handler attempts to satisfy that request and return a description of the image's data layout in the requested format. There is a close relationship between a compute device and the associated storage device.

In addition to the above functions, it is useful to have the storage handler perform single-pixel access for `xil_get_pixel()` and `xil_set_pixel()` to avoid having to copy all of the image data in that case.

A storage device is not required to be able to handle all images, but can limit the sort of images it will store based on any parameter. For example, a storage device may only be capable of storing 8-bit images, or images that are 320-by-240 pixels in size. Processing functions that request this restricted storage must either know of these restrictions and use alternate storage devices for noncompliant images, or correctly handle the failure of a storage request by attempting alternative storage.

Aside from storing data, the main job of a storage handler is format conversion. The interface provided via `getStorage()` is an attempt at a compromise between forcing every device to convert to a single interchange format and requiring each handler to convert between every possible format. The former forces an intermediate copy between devices, while the latter is actually impossible, since the potential list of device formats is unlimited.

Much like I/O devices, storage devices are loaded the first time they are needed. Typically, a compute device handler will cause the storage device handler for a device to be loaded when it first tries to create an image of the associated storage type. The CPU memory storage handler is loaded at the time of the first image creation.

Each storage handler must contain the following global function:

```
XilDeviceStorageType* XilCreateStorageType()
```

This function is called by the handler loader and performs device initialization and sets up any data that will be used by all instances of the storage device.

`XilDeviceStorageType` *Class*

This class describes the connection to a storage device. There is only one instance of this class for each type of storage device, which is created by the device-specific driver. In addition to implementing this class, the driver can add any device-dependent data or functions that are not needed in every instance of a storage handler.

`XilDeviceStorageType` is the abstract class shown below:

Code Example 5-1 Definition of `XilDeviceStorageType` Class

```
class XilDeviceStorageType : public XilDeviceType {
 public:
 //
 // This is the propagate-to-this-device call.
 //
 // This is the function that creates new images on this particular device,
```

Code Example 5-1 Definition of `XilDeviceStorageType` Class

```
 // or moves images from memory to this device. The ‘typename’ field
 // tells the name of the storage type that the ‘image’ is currently.
 // If this storage device knows about the internals of the device type
 // specified by ‘typename’, then it creates a new storage device of
 // this type, copies the image data into it, and returns pointers to
 // both the new storage device and the device dependent ‘description’
 // of the storage. Otherwise it should return NULL. All storage devices
 // need to know how to propagate from memory. The ‘take_ownership’
 // field tells the device driver whether it should delete the data
 // when the storage object is destroyed. See the example storage driver
 // for more information.
 //
 virtual XilDeviceStorage * propagateDeviceStorage (XilImage *image,
        char typename[], void *description,
        int take_ownership)=0;

 // destructor. This should release all resources that were used to
 // make the connection to the device.

 virtual ~XilDeviceStorageType();
};
```

The handler creates a device-specific class that derives from `XilDeviceStorageType`. All device-specific information unique to this instance of the object should be stored here. In the storage handler example on page 154, this class is used to derive a storage handler that supports band-sequential data. The class `XilDeviceStorageTypeBandMemory` is derived from `XilDeviceStorageType`.

`propagateDeviceStorage()` in the *type* class implements image data transfer *to* the device. If the handler knows how to read image storage from the device `typename`, it must create the image storage, copy the image data from the named device, and return a pointer to the local image storage. If `take_ownership` is true, the device storage handler should delete the image data storage when the image is destroyed. If the handler does not know how to propagate from the specified device, it should return `NULL`.

All device storage handlers must know how to propagate to and from the *memory* device, and may know about other devices as well. When a propagation is requested, the core code first tries to use the source and destination devices to see if either knows how to copy directly between the two devices. If neither knows how to copy directly, then two propagations are used: one from the source device to standard memory and a second from standard memory to the destination device.

The `propagateDeviceStorage()` function returns derived instances of the storage object for a particular device.

`XilDeviceStorage` *Class*

This class describes one instance of a particular storage device. Many of these can exist.

The abstract class for `XilDeviceStorage` looks like this:

Code Example 5-2 Definition of `XilDeviceStorage` Class

```
//------------------------------------------------------------------------
//
// Description:
//
//  Definition of the interface to the XilDeviceStorage class
//
//------------------------------------------------------------------------

class XilDeviceStorage : public XilDevice {
 public:
 //
 // This function allows this device to emulate other device types, so
 // that images can be shared across devices that know about each other
 // without needing to copy the data. It returns a pointer to the
 // device dependent structure that describes the internal information
 // about how to access the image as if it were the specified type.
 // The image should either be of the specified type or the storage device
 // driver should be capable of efficiently emulating the specified type.
 // If the storage device cannot emulate the requested type than this
 // function should return NULL.
 //
 virtual void* requestStorageInfo (char typename[])=0;
 //
 // This is the propagate-from-this-device call.
 //
 // This is the function that moves images from the current device to
```

Code Example 5-2 Definition of `XilDeviceStorage` Class *(Continued)*

```
  // the named device. If this storage device knows about the internals
  // of the device type specified by ‘typename’, then it creates a new
  // storage device of the requested type, copies the image data into it,
  // and returns a pointer to it. If the storage device cannot convert
  // directly to the requested type then it should return NULL.
  //
  // All storage devices need to know how to convert to a ‘memory’ type image.
  //
  virtual XilDeviceStorage* propagateDeviceStorage (char typename[])=0;

  //
  // functions to get and set particular image pixel values without
  // forcing a propagation of the image to memory
  //
  virtual void getPixel(unsigned short x, unsigned short y,
     unsigned short band, unsigned short count,
     float* data)=0;
  virtual void setPixel(unsigned short x, unsigned short y,
     unsigned short band, unsigned short count,
     float* data)=0;

  //
  // destructor. This should release all resources that were used to
  // make this particular storage device.
  //
  virtual ~XilDeviceStorage();
};
```

The device storage driver should create a device-specific class derived from `XilDeviceStorage`. The instance of this derived class represents the storage of data for a single image.

`requestStorageInfo()` should return a device-dependent structure that describes the memory layout for the specific image data storage as if it were the specified type. If the image is not of the specified type, or cannot be efficiently treated as if it is of the specified type, `requestStorageInfo()` should return `NULL`. This additional complexity allows an image that resides on a device that can be memory-mapped to expose the device-resident image as a memory image.

The device-dependent storage information need not include things like the image width and height, since that information is available in the device-independent image class.

`propagateDeviceStorage()` in the *device* class copies data from the current device into the named device. If the handler knows about the device type specified by `typename`, it should create a new storage device of the requested type, copy the data into it, and return a pointer to the new storage class. As mentioned above, it should return `NULL` if it cannot convert the data to the requested storage type, but it must be capable of converting to the *memory* storage type.

`getPixel()` and `setPixel()` are used to implement the corresponding API-level functions. They are part of the storage class to prevent having to propagate the image in order to return single pixel values. The pixel information these functions return is an array of floating point data with a size equal to the number of bands in the image.

One note of caution concerning the `getStorage()` member of `XilImage`: it does not respect the existence of children. This means that if `getStorage()` is called on a spatial or band child, `getStorage()` returns the storage information for the parent image, not the child. Since the caller usually wants the information corresponding to the child image, `getStorage()` is not usually called directly, but rather through a utility function. In the storage handler example on page 154, this utility function is called `getBandMemoryStorage()`. It calls `getStorage("band_memory")`, which returns the parent data. It then constructs the appropriate data structure for the requested child image.

Adding a Storage Device

The decision to implement a storage handler should be driven by the performance enhancement expected from allowing the image data to reside on an accelerator. If an accelerator performs several atomic operations that are likely to be called in sequence, it will undoubtedly be advantageous to provide a storage handler to support the compute handler for the accelerator. If, however, the accelerator only provides functionality for a single operator (for example, a JPEG decompressor), the advantage of keeping the image data local to the accelerator is minimal, since the next usage of the image would cause the propagation to the memory format anyway. Storage handlers also can be used to allow local data on an I/O device, but, in the same way, it is of little advantage if the I/O device cannot perform subsequent operators.

Storage devices that modify the format of the image without tying it to an actual hardware device can be useful as well. (The storage handler example on page 154 is this sort of storage device). However, before such a storage handler can be used, there must exist compute handlers that ask for this storage format.

Adding a storage device is relatively simple compared to compute or compression devices. The handler writer must perform the following steps:

1. Subclass the `XilDeviceStorageType` class to represent the desired storage type. If there is initialization needed for the storage device, it should go here. The `propagateDeviceStorage()` defined here should create instances of the class derived below, from `XilDeviceStorage`. If `take_ownership` is `TRUE`, the data associated with the image should be deallocated when the `XilDeviceStorage` instance's destructor is called.

2. Subclass the `XilDeviceStorage` class to represent the desired storage. As a minimum, the implementation of `propagateDeviceStorage()` from the derived `XilDeviceStorage` class must be able to propagate the image data to and from the *memory* storage format. The internal representation of the XIL memory layout is pixel interleaved (except for bit images), exactly the same as exported data. This exported format is described in the *XIL Reference Manual* under `xil_get_memory_storage()`. If the writer of the handler has in-depth knowledge of the layout of some other storage format, the function may also allow propagation to and from that additional format.

3. The implementation should be placed in the file `$XILHOME/lib/pipelines/xil`*device_name*`.so`, where *device_name* is the name of the storage device (see the section on handler installation in

Chapter 2, "More on Writing Device Handlers"). Storage devices are not referenced as dependencies in the `xil.compute` configuration file, but are referenced directly by name via the `getStorage()` parameters.

Sample Storage Device Handler

This example illustrates a storage handler for memory images that are stored in band-sequential format. (The standard XIL memory operators expect images stored in a pixel-sequential format). The example is fairly complete, and illustrates the majority of the features of a storage driver. Note, however, that this example only works for 1-band images (of any type), or `XIL_BIT` images (XIL already stores them in band-sequential format). For the handler to be fully functional, the `copyMemory2BandMemory()` and `copyBandMemory2Memory()` routines would need to be implemented. This example contains two files:

- `XilBandMemoryDefines.h`, which describes structures needed by the storage handler
- `XilDeviceStorageTypeBandMemory.cc`, which implements the storage handler

XilBandMemoryDefines.h

Code Example 5-3 XilBandMemoryDefines.h

```
#ifndef XIL_BANDMEMORYDEFINES_H
#define XIL_BANDMEMORYDEFINES_H

//
// Definition of band-sequential memory storage description
// Other storage drivers may need other information as well (file descriptors,
// accelerator ids, etc.)
//

typedef struct __XilBandMemoryStorageBit {
   Xil_unsigned8* data;           /* pointer to the first byte of the image */
   unsigned short scanline_stride; /* the number of bytes between scanlines */
   unsigned long band_stride;      /* the number of bytes between bands */
   unsigned char offset;          /* the number of bits to the first pixel */
} XilBandMemoryStorageBit;

typedef struct __XilBandMemoryStorageByte {
   Xil_unsigned8* data;           /* pointer to the first byte of the image */
   unsigned long scanline_stride;  /* the number of bytes between scanlines */
   unsigned long band_stride;      /* the number of bytes between bands */
} XilBandMemoryStorageByte;

typedef struct __XilBandMemoryStorageShort {
   Xil_signed16* data;            /* pointer to the first word of the image */
   unsigned long scanline_stride;  /* the number of shorts between scanlines */
   unsigned long band_stride;      /* the number of shorts between bands */
} XilBandMemoryStorageShort;

typedef struct __XilBandMemoryStorageFloat {
   float* data;                  /* pointer to the first float in the image */
   unsigned long scanline_stride;  /* the number of floats between scanlines */
   unsigned long band_stride;      /* the number of floats between bands */
} XilBandMemoryStorageFloat;

typedef union __XilBandMemoryStorage {
   XilBandMemoryStorageBit   bit;
   XilBandMemoryStorageByte  byte;
   XilBandMemoryStorageShort shrt;
   XilBandMemoryStorageFloat flt;
} XilBandMemoryStorage;
```

Code Example 5-3 `XilBandMemoryDefines.h` *(Continued)*

```
#endif
```

XilDeviceStorageTypeBandMemory.cc

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(1 of 17)*

```
#include <strings.h>
#include <xil/XilDeviceStorage.h>
#include <xil/XilImage.h>
#include <xil/XilDeviceStorageType.h>
#include <xil/XilError.h>

#include “XilBandMemoryDefines.h”

//------------------------------------------------------------------------
//
// This is an implementation of a band-sequential (or band-interleaved)
// memory storage driver.  The example is fairly complete, and illustrates
// the majority of the features of a storage driver.  Note however that
// this example only works for 1-band images (of any type), or XIL_BIT
// images (since they are already stored in band-sequential format).  In
// order to become fully functional, the copyMemory2BandMemory() and
// copyBandMemory2Memory() routines would need to be implemented.
//
//------------------------------------------------------------------------


//
// Derived instantiation of XilDeviceStorageType class
// This is the description of the connection to the device
// There is only one of these
//
class XilDeviceStorageTypeBandMemory : public XilDeviceStorageType {
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(2 of 17)*

```
 public:
   virtual XilDeviceStorage* propagateDeviceStorage(XilImage* image,
              char type_name[], void* description, int take_ownership);
 private:
   XilDeviceStorageTypeBandMemory();
   ~XilDeviceStorageTypeBandMemory();
   XilBandMemoryStorage* allocateStorage(XilImage* image);
   XilBandMemoryStorage* convertMemoryStorage(XilImage* image,
                 XilMemoryStorage* memory_storage,
                 Xil_boolean* need_copy);

   // this is a friend for the purpose of calling the constructor
   friend XilDeviceStorageType* XilCreateStorageType();
};

//
// Derived instantiation of XilDeviceStorage class
// This is the description of a particular instantiation of the device
// There can be many of these
//
class XilDeviceStorageBandMemory : public XilDeviceStorage {
 public:
   virtual void* requestStorageInfo(char typename[]);
   virtual XilDeviceStorage* propagateDeviceStorage(char typename[]);
   virtual void getPixel(unsigned short x, unsigned short y,
                         unsigned short band, unsigned short count,
                         float *data);
   virtual void setPixel(unsigned short x, unsigned short y,
                         unsigned short band, unsigned short count,
                         float *data);
 private:
   XilDeviceStorageBandMemory(XilImage* image, XilBandMemoryStorage*
description, int take_ownership);
   ~XilDeviceStorageBandMemory();
   XilDataType dataType;// datatype of image
   XilBandMemoryStorage storage;// device storage description
   XilMemoryStorage memory_storage;// used for emulating memory images
   int owner;// whether device owns data or not
   XilImage* parent;// pointer to parent image description
   friend class XilDeviceStorageTypeBandMemory;
};
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(3 of 17)*

```
//
// prototypes of device utility functions (not implemented in this example)
//
void
copyMemory2BandMemory(XilMemoryStorage *, XilBandMemoryStorage *, XilImage *);

void
copyBandMemory2Memory(XilBandMemoryStorage *, XilMemoryStorage *, XilImage *);



//
// This is the global function called by XIL to create this kind of storage
device
//
XilDeviceStorageType* XilCreateStorageType()
{
   XilDeviceStorageType* device_type;

   // create the storage type
   device_type=new XilDeviceStorageTypeBandMemory();
   if (device_type==NULL) {
      return NULL;
   }
   return device_type;
}

//
// This is the constructor for the storage device
// It is called only once at startup
//
XilDeviceStorageTypeBandMemory::XilDeviceStorageTypeBandMemory()
{
   // there is no data to initialize at the moment
}

//
// This is the destructor for the storage device
// It is called only once at shutdown
//
XilDeviceStorageTypeBandMemory::~XilDeviceStorageTypeBandMemory()
{
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(4 of 17)*

```
}


//
// Routine that creates new images on this particular device, or moves images
// from memory to this device.  This is the propagate-to-this-device call.
//
XilDeviceStorage* XilDeviceStorageTypeBandMemory::propagateDeviceStorage(
   XilImage* image, char *type_name, void* description, int take_ownership)
{
   // if an image is not being passed in, allocate one
   if (type_name==NULL) {
      type_name="band_memory";
      description= (void*) allocateStorage(image);
      if (description==NULL) {
         return NULL;
      }
   }
   else if (strcmp(type_name,"memory")==NULL){
          // This is a memory image -- we can convert it
          Xil_boolean need_copy;
          XilBandMemoryStorage* band_description= (XilBandMemoryStorage *)
              convertMemoryStorage(image, (XilMemoryStorage*) description,
&need_copy);
          if (band_description==NULL) {
              // Could not allocate the storage for the converted image
              return NULL;
          }

          if(need_copy == TRUE) {
              // Could not convert the image in place - need to copy it
              copyMemory2BandMemory((XilMemoryStorage *)description,
               band_description,
               image);
          // however, since this is not implemented, clean up and fail
          delete band_description;
          return NULL;
          }
      else {
              /* Didn't require copy */
          }
          description= band_description;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(5 of 17)*

```
   }
   else if (strcmp(type_name,"band_memory")) {
          // This is already a band_memory image -- don't need to do anything
   }
   else {
      // This is an image of a type that this handler cannot interpret -- fail
      return NULL;
   }

   // create the storage object
   XilDeviceStorage* device= new XilDeviceStorageBandMemory(image,
                (XilBandMemoryStorage*)description,
                 take_ownership);
   if (device==NULL) {
      if(take_ownership)
     delete ((XilBandMemoryStorage*)description)->bit.data;
      delete description;
      return NULL;
   }

   // Now that the image is on this device, delete the external reference to it
   delete description;

   return device;
}

//
// Routine that actually allocates the storage on this device
//
XilBandMemoryStorage*
XilDeviceStorageTypeBandMemory::allocateStorage(XilImage* image)
{
   unsigned short width,height,nbands;
   XilDataType datatype;
   unsigned long size;

   // allocate the storage description
   XilBandMemoryStorage* storage= new XilBandMemoryStorage;
   if (storage==NULL) {
      return NULL;
   }
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(6 of 17)*

```
   // fill it in
   image->getInfo(&width,&height,&nbands,&datatype);
   switch (datatype) {
    case XIL_BIT:
       storage->bit.scanline_stride=(unsigned short)(width+XIL_BIT_ALIGNMENT-
1)/
        XIL_BIT_ALIGNMENT*(XIL_BIT_ALIGNMENT/8);
       storage->bit.band_stride=storage->bit.scanline_stride*height;
       storage->bit.offset= 0;
       size= storage->bit.band_stride*nbands;
       break;
    case XIL_BYTE:
       storage->byte.scanline_stride= width*nbands;
       storage->byte.band_stride=storage->byte.scanline_stride*height;
       size= width*height*nbands*sizeof(Xil_unsigned8);
       break;
    case XIL_SHORT:
       storage->shrt.scanline_stride= width*nbands;
       storage->shrt.band_stride=storage->shrt.scanline_stride*height;
       size= width*height*nbands*sizeof(Xil_signed16);
       break;
    case XIL_FLOAT:
       storage->flt.scanline_stride= width*nbands;
       storage->flt.band_stride=storage->flt.scanline_stride*height;
       size= width*height*nbands*sizeof(float);
       break;
    default:
       return NULL;
   }

   // allocate the actual storage
   storage->bit.data= new Xil_unsigned8[size];
   if (storage->bit.data==NULL) {
      delete storage;
      return NULL;
   }

   return storage;
}
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(7 of 17)*

```
//
// Constructor for an object (image) of this device
//
XilDeviceStorageBandMemory::XilDeviceStorageBandMemory(
   XilImage* image,
   XilBandMemoryStorage* storage,
   int take_ownership)
{
   // copy the description
   dataType= image->getDataType();
   switch (dataType) {
    case XIL_BIT:
      this->storage.bit= storage->bit;
      break;
    case XIL_BYTE:
      this->storage.byte= storage->byte;
      break;
    case XIL_SHORT:
      this->storage.shrt= storage->shrt;
      break;
    case XIL_FLOAT:
      this->storage.flt= storage->flt;
      break;
   }

   owner=take_ownership;
   parent= image;
}

//
// Destructor for an object (image) of this device
//
XilDeviceStorageBandMemory::~XilDeviceStorageBandMemory()
{
   // throw away the data if owner
   if (owner) {
      switch (dataType) {
       case XIL_BIT:
         delete storage.bit.data;
         break;
       case XIL_BYTE:
         delete storage.byte.data;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(8 of 17)*

```
          break;
        case XIL_SHORT:
          delete storage.shrt.data;
          break;
        case XIL_FLOAT:
          delete storage.flt.data;
          break;
      }
   }
}

//
// Routine that allows this device to emulate other device types, so
// images can be shared across devices that know about each other without
// needing to copy them
//
void* XilDeviceStorageBandMemory::requestStorageInfo(char typename[])
{
   if (strcmp(typename,"band_memory")==NULL) {
      // image is of requested type, don't need to do anything
      return &storage;
   }
   else if ((strcmp(typename,"memory")==NULL) &&
      ((parent->getBands() == 1) || (dataType == XIL_BIT))) {
      // 1-banded band_memory images have the same storage representation as
      // memory images.  Also, 1-bit memory images are already band-sequential.
      // In either of these cases, we can emulate a memory image by passing back
      // the appropriate storage description.
      switch (dataType) {
        case XIL_BIT:
          memory_storage.bit.scanline_stride= storage.bit.scanline_stride;
          memory_storage.bit.band_stride= storage.bit.band_stride;
          memory_storage.bit.offset= storage.bit.offset;
          memory_storage.bit.data= storage.bit.data;
          break;
        case XIL_BYTE:
          memory_storage.byte.scanline_stride= storage.byte.scanline_stride;
          memory_storage.byte.pixel_stride= 1;
          memory_storage.byte.data= storage.byte.data;
          break;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(9 of 17)*

```
        case XIL_SHORT:
           memory_storage.shrt.scanline_stride= storage.shrt.scanline_stride;
           memory_storage.shrt.pixel_stride= 1;
           memory_storage.shrt.data= storage.shrt.data;
           break;
        case XIL_FLOAT:
           memory_storage.flt.scanline_stride= storage.flt.scanline_stride;
           memory_storage.flt.pixel_stride= 1;
           memory_storage.flt.data= storage.flt.data;
           break;
        default:
           return NULL;
       }
       return &memory_storage;
    }
    else {
       // the band-sequential memory storage device cannot emulate any
       // other storage types at this time
       return NULL;
    }
}


//
//  Routine to move images from the current device to the named device
//  This is the propagate-from-this-device call
//
XilDeviceStorage* XilDeviceStorageBandMemory::propagateDeviceStorage(char
type_name[])
{
    XilDeviceStorage* new_storage_device;
    XilMemoryStorage* memory_storage;

    if (strcmp(type_name,"band_memory")==NULL) {
       // it is already on the specified device
       return NULL;

    } else {
       // get the storage type so we can access the device that the image
       // will be going to
       XilDeviceStorageType* storage_type;
       storage_type=xil_global_state->getDeviceStorageType(type_name);
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(10 of 17)*

```
      if (storage_type==NULL) {
         return NULL;
      }
      XilDeviceStorage* storage_device;

      // if it is the memory device then create a new memory image and
      // copy if necessary
      if(strcmp(type_name,"memory")==NULL) {

          if((parent->getBands() == 1) || (dataType == XIL_BIT)){
             // it is one band or XIL_BIT memory image -- don't need to copy it
              XilMemoryStorage* mem_storage = new XilMemoryStorage;

              mem_storage->byte.data            = storage.byte.data;
             mem_storage->byte.scanline_stride = storage.byte.scanline_stride;
              mem_storage->byte.pixel_stride    = 1;

          // Let the memory device take it over
              storage_device=
                  storage_type->propagateDeviceStorage(parent,
                                                       "memory",
                                                       mem_storage,
                                                       TRUE);
              if(storage_device==NULL) {
                  return NULL;
              }

          } else {
              // it is a multi-band, non-bit memory image -- must copy

              // create the memory image
              storage_device=
                 storage_type->propagateDeviceStorage(parent,NULL,NULL,TRUE);
              if(storage_device==NULL) {
                  return NULL;
              }

              // get the storage information
              memory_storage=
                  (XilMemoryStorage*)storage_device-
>requestStorageInfo("memory");
              if (memory_storage==NULL) {
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(11 of 17)*

```
                return NULL;
            }

        // do the copy
            copyBandMemory2Memory(&storage,memory_storage,parent);
        // however, since this is not implemented, clean up and fail
        delete memory_storage;
        return NULL;
        }
    } else {
        // it is an image type that this handler does not know how to
    // propagate to -- must copy it

    // first ask the other handler if it knows how to propagate from
band_memory
        storage_device= storage_type->propagateDeviceStorage(parent,
                  "band_memory",&storage,TRUE);
        if (storage_device==NULL) {
        // The other handler doesn't know about band_memory:
        //   Create a memory image and propagate to that.  Then tell
        //   the other handler to propagate from memory to itself
        //   (all handlers need to be able to do this)

        //
            // create the intermediate memory image:
        //
        //   get access to the memory storage device
            storage_type = xil_global_state->getDeviceStorageType("memory");
        //   create an instance of a memory image
            storage_device=storage_type-
>propagateDeviceStorage(parent,NULL,NULL,TRUE);
            if (storage_device==NULL) {
                return NULL;
            }
            // get the memory storage information of the memory image
            memory_storage= (XilMemoryStorage*)storage_device-
>requestStorageInfo("memory");
            if (memory_storage==NULL) {
                return NULL;
            }
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(12 of 17)*

```
          // copy from band_memory to memory
              copyBandMemory2Memory(&storage,memory_storage,parent);
          // however, since this is not implemented, clean up and fail
          delete memory_storage;
          return NULL;

          // tell the memory device handler to propagate to the requested type
              new_storage_device=storage_device-
>propagateDeviceStorage(type_name);
              if (new_storage_device==NULL) {
               return NULL;
            }
            return new_storage_device;
         }
      }
      delete this;
      return storage_device;
   }
}

void XilDeviceStorageBandMemory::getPixel(unsigned short x, unsigned short y,
                                 unsigned short band, unsigned short count,
                                 float* data)
{
   switch (parent->getDataType()) {
    case XIL_BIT:
      {
         Xil_unsigned8* pixel_byte;
         Xil_unsigned8 pixel_bit;
         pixel_byte= storage.bit.data +
                     band*storage.bit.band_stride +
        y*storage.bit.scanline_stride;
         pixel_byte= pixel_byte + ((long)storage.bit.offset+(long)x)/(long)8;
         pixel_bit= (Xil_unsigned8)(1 <<
(((long)storage.bit.offset+(long)x)%(long)8));
         for (unsigned short i=0; i<count; i++) {
            *data++= (*pixel_byte & pixel_bit) ? 1.0 : 0.0;
            pixel_byte= pixel_byte+storage.bit.band_stride;
         }
      }
      break;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(13 of 17)*

```
     case XIL_BYTE:
       {
          Xil_unsigned8* pixel;
          pixel= storage.byte.data +
     band*storage.byte.band_stride +
     y*storage.byte.scanline_stride + x;
          for (unsigned short i=0; i<count; i++) *data++ = *pixel++;
       }
       break;
     case XIL_SHORT:
       {
          Xil_signed16* pixel;
          pixel= storage.shrt.data +
     band*storage.shrt.band_stride +
     y*storage.shrt.scanline_stride + x;
          for (unsigned short i=0; i<count; i++) *data++ = *pixel++;
       }
       break;
     case XIL_FLOAT:
       {
          float* pixel;
          pixel= storage.flt.data +
     band*storage.flt.band_stride +
     y*storage.flt.scanline_stride + x;
          for (unsigned short i=0; i<count; i++) *data++ = *pixel++;
       }
       break;
   }
}

void XilDeviceStorageBandMemory::setPixel(unsigned short x, unsigned short y,
                                 unsigned short band, unsigned short count,
                                 float* data)
{
   switch (parent->getDataType()) {
     case XIL_BIT:
       {
          Xil_unsigned8* pixel_byte;
          Xil_unsigned8 pixel_bit;
          pixel_byte= storage.bit.data+
                      band*storage.bit.band_stride+
        y*storage.bit.scanline_stride;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(14 of 17)*

```
          pixel_byte= pixel_byte + ((long)storage.bit.offset+(long)x)/(long)8;
          pixel_bit= (Xil_unsigned8)(1 <<
((long)storage.bit.offset+(long)x)%(long)8);
          for (unsigned short i=0; i<count; i++) {
             if (*data < .5) {
                *pixel_byte &= ~pixel_bit;
             } else {
                *pixel_byte |= pixel_bit;
             }
             pixel_byte= pixel_byte+storage.bit.band_stride;
             data++;
          }
       }
       break;
     case XIL_BYTE:
       {
          Xil_unsigned8* pixel;
          pixel= storage.byte.data +
    band*storage.byte.band_stride +
    y*storage.byte.scanline_stride + x;
          for (unsigned short i=0; i<count; i++) {
             if (*data > 254.5) {
                *pixel= 255;
             } else if (*data < .5) {
                *pixel= 0;
             } else {
                *pixel= (Xil_unsigned8)(*data + .5);
             }
             pixel++;
             data++;
          }
       }
       break;
     case XIL_SHORT:
       {
          Xil_signed16* pixel;
          pixel= storage.shrt.data +
    band*storage.shrt.band_stride +
    y*storage.shrt.scanline_stride + x;
          for (unsigned short i=0; i<count; i++) {
             if (*data > 32766.5) {
                *pixel= 32767;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(15 of 17)*

```
            } else if (*data < -32768.5) {
               *pixel= -32768;
            } else if (*data > 0.0) {
               *pixel= (Xil_signed16)(*data + .5);
            } else {
               *pixel= (Xil_signed16)(*data - .5);
            }
            pixel++;
            data++;
         }
      }
      break;
    case XIL_FLOAT:
      {
         float* pixel;
         pixel= storage.flt.data +
    band*storage.flt.band_stride +
    y*storage.flt.scanline_stride + x;
         for (unsigned short i=0; i<count; i++) *pixel++ = *data++;
      }
      break;
   }
}


XilBandMemoryStorage*
XilDeviceStorageTypeBandMemory::convertMemoryStorage(XilImage* image,
                 XilMemoryStorage* memory_storage,
                 Xil_boolean* need_copy)
{
   // allocate a storage description
   XilBandMemoryStorage* storage = new XilBandMemoryStorage;
   if(storage == NULL)
       return NULL;

   // fill it in
   XilDataType dataType= image->getDataType();
   switch (dataType) {
    case XIL_BIT:
      storage->bit.data= memory_storage->bit.data;
      storage->bit.offset= memory_storage->bit.offset;
      storage->bit.scanline_stride= memory_storage->bit.scanline_stride;
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(16 of 17)*

```
      storage->bit.band_stride= memory_storage->bit.band_stride;
      break;
    case XIL_BYTE:
      storage->byte.data= memory_storage->byte.data;
      storage->byte.scanline_stride= memory_storage->byte.scanline_stride;
      storage->byte.band_stride= memory_storage->byte.scanline_stride *
            image->getHeight();
      break;
    case XIL_SHORT:
      storage->shrt.data= memory_storage->shrt.data;
      storage->shrt.scanline_stride= memory_storage->shrt.scanline_stride;
      storage->shrt.band_stride= memory_storage->shrt.scanline_stride *
            image->getHeight();
      break;
    case XIL_FLOAT:
      storage->flt.data= memory_storage->flt.data;
      storage->flt.scanline_stride= memory_storage->flt.scanline_stride;
      storage->flt.band_stride= memory_storage->flt.scanline_stride *
            image->getHeight();
      break;
   }

   // check if a copy is necessary
   if ((image->getBands() == 1) || (dataType == XIL_BIT)) {
      // 1-banded band_memory images have the same storage representation as
      // memory images.  Also, 1-bit memory images are already band-sequential.
      // In either of these cases, we do not need to reformat (copy) the data.
      *need_copy = FALSE;
   }
   else {
      // otherwise, we do
      *need_copy = TRUE;
   }

   return(storage);
}


void
copyMemory2BandMemory(  XilMemoryStorage *memory_storage,
                        XilBandMemoryStorage *storage,
                        XilImage *parent)
```

Code Example 5-4 `XilDeviceStorageTypeBandMemory.cc` *(17 of 17)*

```
{
}

void
copyBandMemory2Memory(  XilBandMemoryStorage *storage,
                        XilMemoryStorage *memory_storage,
                        XilImage *parent)
{
}
```

Compression/Decompression 6

This chapter explains how to add a new compression method and compression hardware.

In the XIL library, compression and decompression are implemented using loadable handlers. The library defines a generic compression/decompression interface; different compressors implement this interface and store the implementation in dynamically loadable libraries. The handlers are loaded at runtime when they are requested, much like the I/O handlers. From the viewpoint of the application, this allows a variety of compression techniques to be used in a similar fashion.

The public semantics of compression and the description of the compressors in the XIL library are explained in the *XIL Programmer's Guide*. The information in this chapter assumes you are familiar with those concepts.

Implementation of Compression

The implementation of compression is somewhat different from other handlers in that it is divided into two handlers: a compression handler and a compute handler. Compression handlers contain most of the utility functions for implementing a method of compression and decompression, even though the actual compress and decompress functions are provided in an associated compute handler. The compression handler performs buffer management and implements the semantics of the `XilCis` object.

Figure 6-1 shows the relationship of the classes that must be used to implement compression.

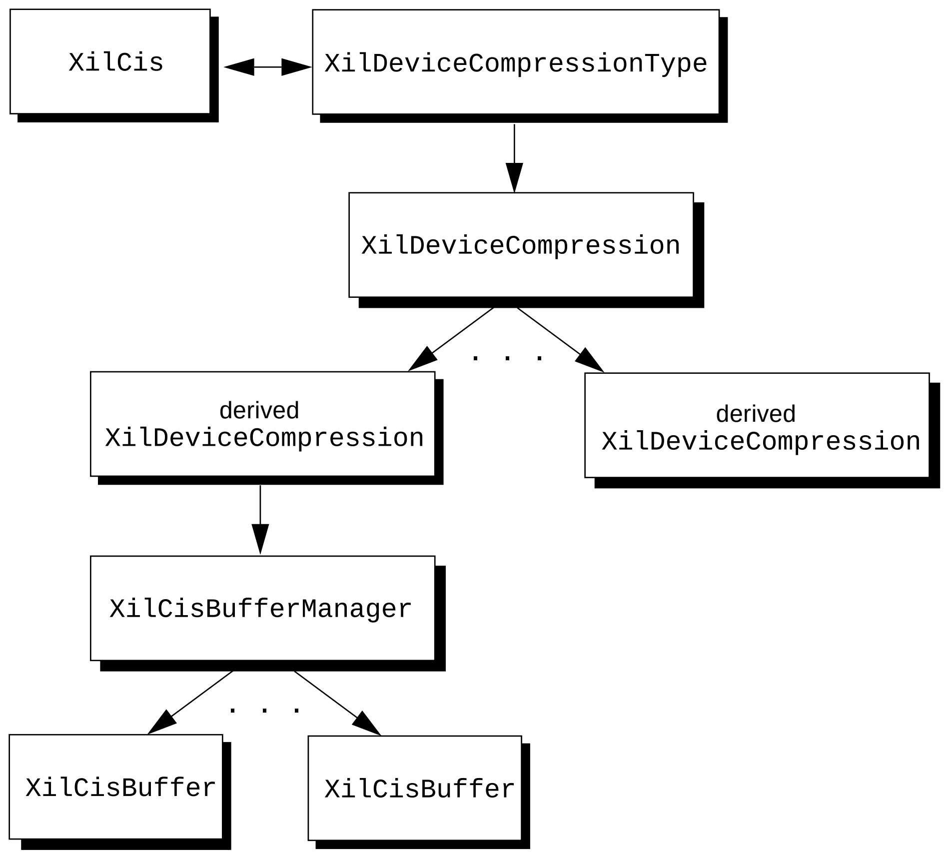

Figure 6-1 Relationship of Classes

The class `XilDeviceCompression` contains functions that manipulate compressed data buffers to enable compression. The actual `compress()` and `decompress()` functions belong to an associated compute handler. This is similar to the situation with I/O devices and the associated compute handlers containing molecules for that I/O device.

The API interface object that holds compression information is called the Compressed Image Sequence, or CIS. The CIS object is created by naming a specific compression via the API call:

```
XilCis xil_cis_create(XilSystemState system_state,
    char* compressor_name)
```

Calling this function causes the named compression handler to be loaded. For example, if the `compressor_name` parameter is `MyCompressor`, the core code looks for a loadable library named `xilMyCompressor.so`, which should contain the device compression classes.

`XilDeviceCompressionType` *Class*

When the compression handler is loaded, the XIL core looks for a specific function to call to create a class derived from `XilDeviceCompressionType` for the specific compression. Each compression handler must contain the following function:

```
XilDeviceCompressionType* XilCreateCompressionType()
```

This function creates a class derived from `XilDeviceCompressionType`. A single instance of this class holds all the information common to a specific compression type. For example, any non-varying tables used in compression could be placed here. The compression handler example on page 214 creates a derived class `XilDeviceCompressionTypeIdentity`; the call to `XilCreateCompressionType()` instantiates this derived class. Compression attribute names are shared between instances of the `XilDeviceCompression` class, so they are defined here. The primary purpose of this class is to create instances of `XilDeviceCompression` through the `createDeviceCompression()` member function.

The `XilDeviceCompressionType` class public data is shown below:.

Code Example 6-1 Definition of `XilDeviceCompressionType` Class

```
class XilDeviceCompressionType : public XilDeviceType {
 public:
 virtual XilDeviceCompression* createDeviceCompression(
       XilCis* xcis )=0;
// constructor for base class
// expects compression name and compression type
XilDeviceCompressionType::XilDeviceCompressionType(char *cname, char *ctype);
```

Code Example 6-1 Definition of `XilDeviceCompressionType` Class *(Continued)*

```
// register attribute is set up to remember only the last registered version
// of an attribute. Since the compression-specific attributes are registered
// last, this allows a compression to override the default implementation
// of an attribute.
protected:
    void registerAttr(char* name, setAttrFunc set, getAttrFunc get);
};
```

The constructor for the class derived from `XilDeviceCompressionType` should call the `XilDeviceCompressionType` constructor with the compressor name and compression type attributes of the CIS. This initializes the compress and decompress function names for this handler, appending *cname* to the `compress_` or `decompress_` basename. Also, the `registerAttr()` calls are made at this time. These calls register the codec functions that are called when the application calls `xil_cis_set_attribute()` and `xil_cis_get_attribute()` for a particular attribute string. Attributes are shared among actual instances of the device compression class (but attribute values are specific to an instance). Following is an example of the constructor for the Identity codec with an attribute `COMPRESSION_QUALITY` that can be set:

```
typedef void (XilDeviceCompression::*setAttrFunc) (void *value);
typedef void* (XilDeviceCompression::*getAttrFunc) ();

XilDeviceCompressionTypeIdentity::XilDeviceCompressionTypeIdentity()
: XilDeviceCompressionType("Identity","IDENTITY") {
    registerAttr("COMPRESSION_QUALITY",
    (setAttrFunc)XilDeviceCompressionIdentity::setCompressionQuality);
    (getAttrFunc)XilDeviceCompressionIdentity::getCompressionQuality);
}
```

The initialization of compressor name and compression type, and the registration of attributes that the base class provides is sufficient for most compression types.

The member function `createDeviceCompression()` is used to create each instance of the compression device. It is a pure virtual function in `XilDeviceCompressionType`; that is, no default implementation exists. A reasonable implementation of `createDeviceCompression()` must be made part of the compression-specific class derived from `XilDeviceCompressionType`.

The `ok()` function of the `XilDeviceCompression` class must be called to ensure that the device compression instantiation was successful. The status of the constructor can be tracked by a local flag. This function returns a pointer to the device compression object if the constructor was successful. If the constructor failed, then it returns `NULL`. If `destroy` is `TRUE`, then this function will call the destructor before returning. This function should also call the `ok()` function of the base class to ensure it was successful. Shown next is an example of an `ok()` function:

```
XilDeviceCompressionIdentity*
XilDeviceCompressionIdentity::ok(Xil_boolean destroy) {
    if(this == NULL) {
        return NULL;
    } else {
        // check base class constructor
        if(XilDeviceCompression::ok(FALSE) == this && isok ==
TRUE) {
            return this;
        } else {
            // either failure in base class or in this constructor
            if(destroy == TRUE) delete this;
            return NULL;
        }
    }
}
```

`XilDeviceCompression` *Class*

More than one CIS may exist for a particular compression type. For each CIS, there is an instance of the class derived from `XilDeviceCompression`. The instance of the derived class is created using the `createDeviceCompression()` function described above. This class implements the various utility functions needed to control the compressed data. The definition of the `XilDeviceCompression` class is shown next.

Code Example 6-2 Definition of `XilDeviceCompression` Class *(1 of 3)*

```
class XilDeviceCompression : public XilDevice {
public:
// Sufficient default implementation.
// Functions in this grouping should have an adequate default
// implementation in the base class for a variety of compression types.
// They support calls from the XilCis class.

 virtual int setInputType(XilImageType* type);// called if inputType is
                                             // unknown
 char* getCompressor();
 char* getCompressionType();
 Xil_boolean getRandomAccess();
 XilImageType* getInputType();
 XilImageType* getOutputType();             // may have to parse to get this
 XilImageType* getOutputTypeHoldTheDerivation(); // no parsing
 XilCisBufferManager* getCisBufferManager();
 XilCis* getCis();
 int getFramesToCompress();
 void setFramesToCompress(int number_of_frames);
 int getAttribute(char *name, void** value)
 { return comp_type->getAttr(this, name, value); }
 int setAttribute(char *name, void* value)
 { return comp_type->setAttr(this, name, value); }
 void setInMolecule(Xil_boolean on_off)
 { in_molecule = on_off; }
 Xil_boolean inMolecule()
    { return in_molecule; }
 void destroy() {delete this; };

 // Dependent on XilCisBufferManager
 // These functions reflect the actual state of the cis, as opposed to the state
 // the user sees (if operations are deferred).
 int getStartFrame();
 int getReadFrame();
 int getWriteFrame();

 // No-action default implementation
 // Functions in this grouping take no action for the default implementation,
```

Code Example 6-2 Definition of `XilDeviceCompression` Class *(2 of 3)*

```
// which will be sufficient for simple compression types
virtual int decompressHeader(void);
virtual void flush(void);

// Dependent on XilCisBufferManager, the default is no-typed frames.
// Functions in this grouping call functions within XilCisBufferManager
// to perform the action for compression types with no notion of history.
// If codec has history (typed frames that are known as key frames), these
// functions must be implemented in the derived class.
virtual void seek(int framenumber,          // Seek to the given frame number
      Xil_boolean history_update = TRUE);   // Maintain valid history if
                                            // history_update flag is TRUE
virtual int adjustStart(int new_start_frame);// Call to adjust the beg.
                                             // of the CIS

// Dependent on XilCisBufferManager ordinal numbering default.
// Functions in this grouping call functions within the XilCisBufferManager to
// perform the action for compression, where XilCisBufferManager considers
// frames in the order they appear in the CIS.
// If this does not apply for the compression type, as in MPEG1, these
// function must be implemented in the derived class.
virtual void* getBitsPtr(int* nbytes, int* nframes);
virtual int hasData();
virtual int numberOfFrames();
virtual Xil_boolean hasFrame();
virtual void putBitsPtr(int nbytes, int nframes, void* data,
      XIL_FUNCPTR_DONE_WITH_DATA done_with_data = NULL);
virtual void putBits(int nbytes, int nframes, void* data);

// Error reporting function
// Defaults to notifyError provided for each systemState
// which is sufficient for most compression types
void generateError(XilErrorCategory category, char* id,
                   int primary, Xil_boolean read_invalid,
                   Xil_boolean write_invalid,
                   int line, char* file);
void generateError(XilErrorCategory category, char* id,
                   int primary, int line, char* file);

// Error recovery function
// Activated by xil_cis_attempt_recovery
// Defaults to no action
```

Code Example 6-2 Definition of `XilDeviceCompression` Class *(3 of 3)*

```
 virtual void attemptRecovery(unsigned int nframes, unsigned int nbytes,
                      Xil_boolean &read_invalid, Xil_boolean &write_invalid);

 // These functions MUST be implemented in the derived class
 // Functions that are specific to the compression type
 virtual void reset(void);
 virtual int deriveOutputType(void);
 virtual int findNextFrameBoundary();

// Function called to ensure the device compression instantiation was
// successful
 XilDeviceCompression* ok(Xil_boolean destroy = TRUE);

 protected:
    virtual int getMaxFrameSize() = 0;        // called to figure maximum
                                              // frame size
    virtual void burnFrames(int nframes) = 0;// update history going forward

};
```

This class has several virtual functions with default implementations. These implementations will work for most compression types that do not have interframe encoding. For compression techniques that require interframe encoding, the implementor must replace these functions with appropriate ones that obey the semantics.

There is a library of buffer management classes that is used to implement the default versions of these members. The CIS buffer manager is described in the section "The CIS Buffer Manager" on page 189.

Base Class Implementations

This section discusses the functions that use the base class implementations. These functions support calls from the `XilCis` class.

The following functions return class variables that have been initialized during the creation of the codec:

- `getCompressor()`
- `getCompressionType()`

- `getRandomAccess()`
- `getInputType()`
- `getOutputType()`
- `getOutputTypeHoldTheDerivation()`
- `getCisBufferManager()`
- `getCis()`
- `getFramesToCompress()`
- `setFramesToCompress()`
- `getAttribute()`
- `setAttribute()`

Note that `getAttribute()` and `setAttribute()` essentially invoke the codec function that was registered (in the type constructor) with `char *name`.

`setInMolecule()` sets a flag that indicates if the currently active codec function is a molecule. At the entry point of the molecule, this flag should be set to `TRUE`. At the entry point of an atomic compress or decompress, this flag should be set to `FALSE`. This function is used by the error reporting.

`destroy()` calls the destructor of the derived class when the CIS is destroyed via `xil_cis_destroy()`.

The following three functions query the `XilCisBufferManager` object for the actual state of the CIS (opposed to the state the user sees):

- `getStartFrame()`
- `getReadFrame()`
- `getWriteFrame()`

The return values will not include operations that have been deferred. The return values are the current values the `XilCisBufferManager` object has for the start, read, and write frame of the CIS.

Sufficient Default Implementation

`setInputType()` provides the mechanism to track the input and output types of a CIS. The type includes the width, height, number of bands, and the data type of the CIS. This function is called automatically by the `XilCis` class when the first `xil_compress()` is scheduled. Once the input type of the CIS has been defined with nonzero values, it cannot be changed. Therefore, any subsequent `xil_compress()` calls with different values for the width, height, number of bands, or data type than the original values generates an error condition. When no calls to `xil_compress`() are made, the `deriveOutputType()` function activates the type tracking after it determines the type from the bitstream. See the section“Functions That Must Be Implemented” on page 186 for a description of `deriveOutputType()`.

No Action for the Default Implementation

The following two functions take no action for the default implementation:

- `decompressHeader()`
- `flush()`

int decompressHeader(void);

`decompressHeader()` is called when the function `xil_decompress()` is scheduled. This function takes no action for the default implementation. However, `decompressHeader()` can be implemented to parse the bitstream and make attributes for the current frame available to the application. The implementation of `decompressHeader()` is useful when there are attributes of the CIS that are easily located by parsing the frame. For example, each frame in an H261 bitstream has an attribute bit that flags the source of the image as a document Camera. In an application, you may want to direct document Camera images to a different destination than non-document Camera images from the CIS. Therefore, the handler could have a `decompressHeader()` function that stores the attribute value from the current frame. Then, the application can use the `xil_cis_get_attribute()` function to get the value of the attribute and take appropriate action before the scheduled decompress gets executed.

void flush(void);

`flush()` is called when the function `xil_flush()` is scheduled. This function takes no action for the default implementation. The CIS will be synchronized so that any pending compress operations are scheduled. This function should be implemented for a codec that buffers frames internally (such as MPEG). The buffered frames need to be made available in some form for the output CIS.

Determine the CIS Read Position

Note – seek() calls a function within XilCisBufferManager to perform the action for compression types with no typed frames. If a codec has history (typed frames that are known as key frames), this function must be implemented in the derived class.

void seek(int framenumber, Xil_boolean history_update);

seek() determines the CIS read position (this is always to a frame number). The framenumber parameter is determined by the XilCis class, which tracks the read frame. The history_update parameter of the seek() function is determined by the function that is coupled with seek(), which is handled by the XilCis class, as shown next:

```
xil_cis_seek(cis,0,0);     // Set up XilCis read_frame
xil_cis_has_frame(cis);    // Call deviceCompression->seek();
                        // then call deviceCompression->hasFrame()
```

The value history_update indicates whether the seek must maintain valid history. TRUE indicates that the history is necessary; FALSE indicates it is not necessary. Note that the history_update parameter is useful only for compression types that have key frames (history). The default value for history_update is FALSE, which assumes the compression type has no key frames. Therefore, seeks are based only on position.

Shown next is the default implementation of the seek() function in the XilDeviceCompression class, which shows that the actual seek is performed by the XilCisBufferManager::seek() function:

```
frames_to_burn = cbm.seek(framenumber,XIL_CIS_ANY_FRAME_TYPE);
if (frames_to_burn > 0)
    burnFrames(frames_to_burn);
```

The value returned, frames_to_burn, is the number of frames that must be processed to reach the requested position. For more information on the XilCisBufferManager::seek() function, see the section “Seek a Specific

Frame within the CIS” on page 203. `burnFrames()` must be implemented for each class derived from `XilDeviceCompression`. The parsing `burnFrames()` performs is compression specific. See the section “Functions That Must Be Implemented” on page 186 for a description of `burnFrames()`.

The following table explains the meanings for the possible values of `frames_to_burn`.

Value	Meaning
Negative	An error occurred
Zero (0)	The read position of the CIS is at the desired frame
Positive	The number of frames which must be processed to reach the requested position

Adjust the Start of a CIS

`int adjustStart(int new_start_frame);`

`adjustStart()` is called by the `XilCis` class when the start of a CIS must be adjusted. An adjustment is activated by the CIS attributes defined by `xil_cis_set_keep_frames()` and `xil_cis_set_max_frames()`. These two functions are described in *XIL Programmer’s Guide*. The default implementation of the `adjustStart()` function adjusts the start frame based only on the input parameter for the frame number.

Note – This default implementation is not sufficient for a compression type with key frames (history), which may be kept in the CIS prior to the start frame. See the section “Adjust Start Frame within Buffer Lists” on page 206 that discusses the `XilCisBufferManager::adjustStart()` function.

Compression Types with Ordinal Numbering

The following functions depend on the `XilCisBufferManager` class to perform the requested action:

- `getBitsPtr()`
- `hasData()`
- `numberOfFrames()`

- `hasFrame()`
- `putBitsPtr()`
- `putBits()`

The default implementations of these functions as defined in the `XilCisBufferManager` class work correctly for compression types with ordinal numbering. In other words, when there are five frames in the CIS, the frames are numbered 0-4, in order. However, for a codec that has out-of-order frames (such as MPEG), the codec must determine if the five frames in the CIS are really frames 0-4 by tracking the temporal reference of each frame.

The tracking of the temporal reference of each frame requires extra parsing, which is not implemented in the default functions listed above. Therefore, if a codec has out-of-order frames, these functions must be implemented in the derived class.

See the section “XilCisBufferManager Class” on page 192 for a description of each of these functions.

Error Reporting

```
void generateError (XilErrorCategory category,
       char* id, int primary, Xil_boolean read_invalid,
       Xil_boolean write_invalid, int line, char* file);
```

`generateError()` is called by the derived `XilDeviceCompression` class to register the state of its error. If the error occurs during the reading (decompress) of the bitstream, the `read_invalid` parameter should be set to `TRUE`. If the error occurs during the writing (compress) to the bitstream, the `write_invalid` parameter should be set to `TRUE`. This error function calls a corresponding function in the `XilCis` class to store the information for the read / write invalid flags and current CIS state.

Error Recovery

`attemptRecovery()`, by default, takes no action. It is a hook that is provided to enable you to manage a response to an illegal bitstream and to go beyond error reporting.

Functions That Must Be Implemented

The following functions must be implemented in the derived class:

- `reset()`
- `getMaxFrameSize()`
- `deriveOutputType()`
- `burnFrames()`
- `findNextFrameBoundary()`

`void reset(void);`

`reset()` is called when the codec is reset via `xil_cis_reset()`. This function must clear the state of the CIS so that it is the same as a newly created CIS. Clearing the state of the CIS involves many of the same actions that are performed by the class constructor. In the XIL codec implementations, each class has a private function, `initValues()`, that performs the common actions for start-up and reset.

One of the common actions is to set up the default input type of the CIS. The value of the variable fields within an input type must be 0 (zero). The value of the non-variable fields within an input type must be their restricted value. For example, "MyCodec" operates only on byte images, but the byte images can be of varying size and number of bands. Therefore, the input type has only one non-variable field, and its value must be `XIL_BYTE`. The other fields are variable fields, and their values are 0 (zero), as shown in the following code:

```
int XilDeviceCompressionMyCodec::initValues() {
    // set up input/output type
    XilImageType* t = cis->getSystemState()->
        createImageType(0,0,0,XIL_BYTE);
    outputType = inputType = t;

    // initialize any state
    width = 0;
    height = 0;

    return XIL_SUCCESS;
}
```

`reset()` must destroy the current input and output type, call the `initValues()` function, and then call the base class `reset()` function. The base class `reset()` function performs the reset for the `XilCisBufferManager` class. For example,

```
void XilDeviceCompressionMycodec::reset() {
    if (inputType != outputType)
        outputType->destroy();
    inputType->destroy();

    initValues();
    XilDeviceCompression::reset();
}
```

`int deriveOutputType(void);`

`deriveOutputType()` is called when the input/output type of the CIS is unknown (for example when the data has been loaded into the CIS from an external file, rather than compressed). In this case, the bitstream must be parsed to determine the fields for the type: xsize, ysize, number_bands, and datatype. The parsed type must be passed to the `setInputType()` function of the base class. `setInputType()` will compare the parsed type against the variable fields for this CIS and report any errors. If no errors exist, `setInputType()` stores the parsed type as the input/output type of the CIS. For example,

```
// get pointer to current frame
bp32 = (Xil_unsigned32*)cbm.nextFrame();

image_width = *bp32++;
image_height = *bp32++;
image_bands = *bp32++;

if(image_width && image_height && image_bands) {
   newtype = cis->getSystemState()->createImageType(image_width,
            image_height, image_bands, XIL_BYTE);

    // set up input/output type and check variable fields
    setInputType(newtype);
}
```

int findNextFrameBoundary();

findNextFrameBoundary() is activated by the XilCisBufferManager class when frame boundaries have not been determined for a CIS. This function parses the CIS bitstream, using special interface functions within the XilCisBufferManager class, until the end of the current frame is found, or until no more available data in the CIS exists. The function then returns a status to indicate its success or failure at finding the end of the current frame. See the section “Determine if a Complete Frame Exists” on page 201” for a detailed discussion of findNextFrameBoundary().

void burnFrames(int nframes);

burnFrames() must process through the number of frames specified by the input parameter, nframes. The XilDeviceCompression object must process the bitstream in accordance with the device’s dependence on history or interframe data. Burning a frame can be as simple as parsing through until the end-of-frame marker is found, or it may invoke many of the same functions as a decompress. The burnFrames() function should use the nextFrame() and decompressedFrame() functions in the XilCisBufferManager class (see the section “Guarantee a Complete Frame for the Codec to Decompress” on page 198” and the section “After a Frame is Decompressed” on page 198. Below is an example that has a fixed size for each frame; in this case burning is quite simple:

```
void
XilDeviceCompressionIdentity::burnFrames(int nframes)
{
    Xil_unsigned8* bp = (Xil_unsigned8*)cbm.nextFrame();
                .
                .
    for(int i=0; i<nframes; i++) {
        bp += frame_size;
        cbm.decompressedFrame(bp);
    }
```

int getMaxFrameSize();

getMaxFrameSize() must determine the worst case compression for the given input type of a CIS. This function should return the maximum number of bytes for any frame in the CIS, which must include anything that appears between the start of a frame and the start of the next frame (for example, headers and markers). This function is used by the `XilCisBufferManager` class to determine the space needed for compression and to test for at least one frame in the current buffer.

For the Identity example at the end of this chapter, the `getMaxFrameSize()` function is the (`xsize*ysize*nbands`) + "12", for the three header words:

```
int XilDeviceCompressionIdentity::getMaxFrameSize(void) {
    return ((int)inputType->getWidth()*
            (int)inputType->getHeight()*
                 inputType->getBands() +12);
}
```

The CIS Buffer Manager

The CIS buffer manager maintains a list of buffers in which compressed data is stored. There is a `XilCisBufferManager` object associated with each `XilDeviceCompression` object (see Figure 6-1).

What follows is a description of the CIS buffer manager interface. The default implementation of the virtual functions in the `XilDeviceCompression` class make use of the CIS buffer manager. If the CIS buffer manager works for your compression technique, you should make use of it; overriding the default virtual functions in the `XilDeviceCompression` class is possible without it, but re-implementing the functionality of the CIS buffer manager will certainly increase the effort needed to implement a new compression technique.

`XilCisBuffer` *Class*

An `XilCisBuffer` object acts as a buffer for compressed data. It can be used to create a buffer of a particular size or to make use of an already-existing buffer. The object contains the number of complete frames and number of bytes currently contained in the buffer, the size of the buffer, and an index to the first frame in the buffer. It also contains a flag that indicates whether the buffer may contain a partial frame at its end.

Note – An `XilCIsBuffer` object contains only frames. It does not support a bitstream that mixes frame data with non-frame data (for instance, audio). If frame data mixed with non-frame data is supported by your codec, then the non-frame data must be grouped along with its associated frame, which can be either the previous or following frame. The codec is responsible for handling and processing the non-frame data. If you are mixing frame and non-frame data, the maximum frame size for the compression type must include the maximum size of the attached non-frame data.

The `XilCisBuffer` object maintains pointers to the start of the current frame being written to, the current frame being read from, and the next available byte in the buffer. `XilCisBuffer` may be allowed control over its buffer; in this case it may destroy the buffer if needed. Otherwise, the buffer is expected to be allocated elsewhere, and a callback function may be provided to free the external storage.

`XilCisBuffer` also keeps a list of objects that contain information about each frame within the buffer. These `XilFrameInfo` objects contain information such as the starting byte of a frame within the buffer and the number of bytes in the frame. This list is built up by the compressor each time it compresses a frame, and by the decompressor each time a frame is decompressed. A pointer to the current position in the list is also held by the `XilCisBuffer` object.

The public part of the `XilCisBuffer` class is shown below:

Code Example 6-3 The `XilCisBuffer` Class

```
class XilCisBuffer {

public:

 XilCisBuffer(unsigned buf_size, int approx_nframes);
 XilCisBuffer(unsigned nbytes, int nframes, Xil_unsigned8* buf,
    int frame_id, XIL_FUNCPTR_DONE_WITH_DATA done_data,
    int approx_nframes = 0);

 ~XilCisBuffer();
 XilCisBuffer * ok(); // constructor creation OK function

 //--------------------- Byte Addition Functions -------------------------
```

Code Example 6-3 The `XilCisBuffer` Class

```
 void addByte( int b );
 void addBytes( Xil_unsigned8* b, unsigned nbytes );

 void addShort(int s) { addByte((s) >> 8); addByte(s); }
 void addShorts(int* s, unsigned m_shorts );

 //------------------------ Public Attributes Access --------------------

 int getNumFrames() const { return num_frames; }
 int getStartFrameId() const { return start_frame_id; }
 int getNumBytes() const { return wptr - buffer; }
 unsigned getBufferSize() const { return buffer_size; }
 int getNumBytesInWFrame() const { return wptr - wfptr; }
 int getNumBytesInRFrame();
 Xil_unsigned8* getNumBytesToFrame(int end_id, int* nbytes);

 //--------------------- Other Member Functions -----------------------

 int frameAtRfptr() const;
 int frameAfterRfptr(int max_frame_size, Xil_boolean need_EOF = FALSE) const;
 int numAvailBytes() const;

 //----------- Merging PB --------------------
 int removeStartFrame();
friend class XilCisBufferManager;

};
```

For device compressions that use the implementation of the `XilCisBuffer`, the only functions that are important are the byte addition functions:

- `addByte()` inserts the given byte
- `addBytes()` inserts *n* bytes starting from the given pointer
- `addShort()` inserts the given short
- `addShorts()` inserts *n* shorts starting from the given pointer

These functions allow a compressor that uses the `XilCisBufferManager::nextBuffer()` function to add bytes into the current buffer.

The other functions in the `XilCisBuffer` class are used by the `XilCisBufferManager` class (discussed next). These functions are shown in Code Example 6-3 in the case that you want to reimplement them.

`XilCisBufferManager` *Class*

The `XilCisBufferManager` class manages the multiple `XilCisBuffer` objects that make up a CIS. This class maintains a list of buffers in which the compressed data is kept. Three important positions exist within this list of buffers: the start of the list, the buffer which certain operations will read from, and the current write buffer. The `XilCisBufferManager` class is shown next:

Code Example 6-4 The `XilCisBufferManager` Class *(1 of 3)*

```
class XilCisBufferManager {
public:

 XilCisBufferManager(int mfs, int nfpb); // constructor

 void   reset();
 XilCisBufferManager* ok();              // constructor creation OK function
 void setXilDeviceCompression(XilDeviceCompression* dc);
 XilDeviceCompression* getXilDeviceCompression();

// functions to set or get the maximum frame size or number of frames per buffer
 int setFrameSize(int fs);           // set maximum frame size (mfs)
 void setNumFramesPerBuffer(int nfpb)// set number of frames per buffer (nfpb)
 int getFrameSize();                 // get mfs
 int getNumFramesPerBuffer();        // get nfpb

// functions to get attributes of a frame
 int getSFrameId();                  // get start frame ID
 int getRFrameId();                  // get read frame ID
 int getWFrameId();                  // get write frame ID
 int getRFrameType();                // get read frame type
 void* getRFrameUserPtr();           // get read frame user pointer
 int setRFrameUserPtr(void* uptr);   // set read frame user pointer

// compress frame into CIS, method 1
 XilCisBuffer* nextBuffer();
 int compressedFrame(int type = XIL_CIS_DEFAULT_FRAME_TYPE);
```

Code Example 6-4 The `XilCisBufferManager` Class *(2 of 3)*

```
// compress frame into CIS, method 2
 Xil_unsigned8* nextBufferSpace();
 int doneBufferSpace(int nbytes, int type = XIL_CIS_DEFAULT_FRAME_TYPE);

// function to guarantee a complete frame is available for codec to decompress
 Xil_unsigned8* nextFrame(Xil_unsigned8** r_buffer_end = NULL,
        Xil_boolean need_EOF = FALSE);

// function that is called by the decompressor when it is done with a frame
 void decompressedFrame(Xil_unsigned8* bfptr,
        int type = XIL_CIS_DEFAULT_FRAME_TYPE, void* user_ptr = NULL);

// functions to put data into a buffer of the CIS buffer manager
 void putBits(int nbytes, int nframes, void* data);
 void putBitsPtr(int nbytes, int nframes, void* data,
        XIL_FUNCPTR_DONE_WITH_DATA = NULL);

// function to return a pointer to data that has been compressed or loaded into
// the current read buffer of the CIS buffer manager
 void* getBitsPtr(int* nbytes, int* nframes);

// functions to return data and frame information about the CIS
 int hasData();
 int numberOfFrames();
 Xil_boolean hasFrame();

// functions to determine if a complete frame exists in the current read buffer
 Xil_unsigned8* getNextByte();
 Xil_unsigned8* getNextBytes(int* nbytes);
 int foundNextFrameBoundary(Xil_unsigned8* frame_ptr);

// function to return any bytes that may have been over-read
 Xil_unsigned8* ungetBytes(Xil_unsigned8* curr_ptr, int nbytes);

// functions to seek a specific frame within a CIS
 int seek(int framenumber, int type = XIL_CIS_ANY_FRAME_TYPE);
 void setSeekToStartFrameFlag(Xil_boolean value)
      {seek_to_start_frame_flag = value;}
// function to adjust the start frame within buffer lists
 int adjustStart(int framenumber, int type = XIL_CIS_ANY_FRAME_TYPE);
```

Code Example 6-4 The `XilCisBufferManager` Class *(3 of 3)*

```
// this function is a special case of seek()
 int seekBackToFrameType(int type);

// function to add end-of-sequence marker for MPEG
 int addToLastFrame(Xil_unsigned8* data, int nbytes);

// functions to allow MPEG to implement its own getBits function
 Xil_unsigned8* getRBuffer() { return ((Xil_unsigned8 *) r_buffer); }
 Xil_unsigned8* getNumBytesToFrame(int end_id, int* nbytes);
 int moveEndStartOneBuffer();

// functions to handle errors and recovery
 void byteError(Xil_unsigned8* bptr); // called instead of decompressedFrame
 int nextSeek(int framenumber, int type = XIL_CIS_ANY_FRAME_TYPE);
 int prevSeek(int framenumber, int type = XIL_CIS_ANY_FRAME_TYPE);
 void nextKnownFrameBoundary(Xil_unsigned8* cptr, Xil_unsigned8** fptr,
       int* num_frames);
 void errorRecoveryDone(Xil_unsigned8* fptr, int num_frames,
       Xil_boolean fixed);
 void foundFrameDuringRecovery(Xil_unsigned8* fptr);

};
```

An `XilCisBufferManager` object has a frame size and a number of frames per buffer value associated with it. Whenever the manager deems it necessary to create a new `XilCisBuffer` object with its own data storage, it will create the new object such that it has a buffer of size (`max_frame_size * num_frames_per_buf`) bytes. For `XilCisBuffer` objects with external storage, the memory buffer is allocated by the application and is whatever size the application provides.

Attributes of a Frame

The elemental unit of a CIS is a frame. The codec can access three attributes of a frame through the `XilCisBufferManager` class, as follows:

- The frame's ID (frame_id), which is an ordinal number that refers to the position in the CIS, starting at 0 and increasing monotonically.

- The frame's type (frame_type), which is a positive integer assigned to the frame when the frame was compressed or decompressed. This attribute is useful for codecs that understand the concept of key frames in the bitstream. This attribute defaults to `XIL_CIS_DEFAULT_FRAME_TYPE`.
- The frame's user data pointer (user_ptr), which is a pointer to data allocated by the decompressor that is associated with the frame using `malloc()`. For instance, an MPEG codec might use this attribute to store the frame's display ID, since this is different from the ordinal frame_id.

The `XilCisBufferManager` class keeps track of the following special frames:

- Write frame, which is the frame_id of the write position in a CIS. It may be equal to -1 if an unknown number of frames has been loaded into the CIS.
- Start frame, which is the frame_id of the first position in the CIS. It is initialized as 0, but the start of the CIS may be adjusted to minimize data storage requirements. See the discussion of `adjustStart()` in the section "Adjust the Start of a CIS" on page 184.
- Read frame, which is the frame_id of the read position in a CIS. It is advanced when data is read from the CIS, either for a decompress or for an output (`getBits`) operation. The read frame can be positioned by an explicit seek from the application. Refer to the discussion of `seek()` in the section "Determine the CIS Read Position" on page 183.

The Constructor and Associated Functions

The constructor for the `XilCisBufferManager` class requires two parameters:

- `mfs`, the maximum frame size for the compression type
- `nfpb`, the number of frames per buffer

The value of `mfs` must be the worst case scenario (for example, the largest number of bytes that a compressed frame might require, including header, markers, etc.). The value of `nfpb` is used with the value of `mfs` to determine the number of bytes allocated for each `XilCisBuffer` object (`mfs` * `nfpb`).

`ok()` checks the success of the constructor in a similar fashion to the `XilDeviceCompression::ok` function. `ok()` returns a NULL pointer if a failure occurs; otherwise, it returns a pointer to the `XilCisBufferManager` object.

setXilDeviceCompression() is called by the base class XilDeviceCompression during the instantiation. This function registers the derived XilDeviceCompression class, which is necessary for access to that class's findNextFrameBoundary() function. getXilDeviceCompression() returns the registered pointer.

Reset the Codec

reset() is called by the base class XilDeviceCompression when the codec must be reset. reset() handles the freeing of currently buffered data in a CIS. The pointers for the start, read, and write positions are reset to 0, and any local variables are initialized to their default values. The CIS is empty and ready for a new bitstream.

Set/Get Maximum Frame Size and Number of Frames per Buffer

The following functions are used to set/get the maximum frame size and the number of frames per buffer:

- setFrameSize() sets the maximum frame size per buffer
- getFrameSize() gets the maximum frame size per buffer
- setNumFramesPerBuffer() sets the number of frames per buffer
- getNumFramesPerBuffer() gets the number of frames per buffer

The codec is allowed only to increase the maximum frame size, not decrease it. A request to decrease its value generates an error.

Method One of Adding Data to a CIS Bitstream

One method of adding data to a CIS bitstream is to use the nextBuffer() and compressedFrame() functions.

XilCisBuffer* nextBuffer();

The nextBuffer() function is called by the compressor to request space for a compressed frame. The XilCisBufferManager object checks the current XilCisBuffer to see if max_frame_size bytes are available. If they are not, XilCisBufferManager allocates a new XilCisBuffer object. The pointer to the appropriate buffer is returned to the compressor. The derived

XilDeviceCompression class must use the XilCisBuffer functions shown in the following code example to add data to the CIS. The buffer tracks its own pointer to the added bytes/shorts. Next, the example compressor adds the header bytes to the CIS for width, height, and nbands.

```
// write the image parameters into the byte-stream
cisbuf->addBytes((Xil_unsigned8*)&cis_width,
sizeof(cis_width));
cisbuf->addBytes((Xil_unsigned8*)&cis_height,
    sizeof(cis_height));
cisbuf_addBytes((Xil_unsigned8*)&cis_bands, sizeof(cis_bands));
```

int compressedFrame(int type = XIL_CIS_DEFAULT_FRAME_TYPE);

When the compressor has finished adding data to the CIS bitstream, it must call compressedFrame(). If the frame that was compressed needs to have a type associated with it, you should pass the frame's type as a parameter to the compressedFrame() function. Otherwise, the default value for a frame type is assigned to the frame.

Note – You must use nextBuffer() with compressedFrame().

Method Two of Adding Data to a CIS Bitstream

Method two of adding data to a CIS bitstream is to use the nextBufferSpace() and doneBufferSpace() functions.

Xil_unsigned8* nextBufferSpace();

for the compressor to call the nextBufferSpace() function is called by the compressor to return a pointer to an available buffer. The pointer is of type Xil_unsigned8*, which means the compressor is responsible for adding data one frame at a time and tracking its own pointer. There are no calls to the XilCisBuffer class.

```
int doneBufferSpace(int nbytes, int type =
                    XIL_CIS_DEFAULT_FRAME_TYPE);
```

When the compressor has finished adding data to the CIS bitstream, it must call the `doneBufferSpace()` function. The first parameter you must pass to this function is the number of bytes (`nbytes`) added to the buffer. Also, there is an optional frame type parameter for `doneBufferSpace()`.

`doneBufferSpace()` may cancel a call to `nextBufferSpace()` if the value of `nbytes` for `doneBufferSpace()` is -1.

Note – You *must* use `nextBufferSpace()` with `doneBufferSpace()`.

Guarantee a Complete Frame for the Codec to Decompress

`nextFrame()` is the only function that guarantees a complete frame is available for the codec to decompress. The prototype of `nextFrame()` is:

```
Xil_unsigned8* nextFrame(Xil_unsigned8** r_buffer_end,
               Xil_boolean need_EOF);
```

This function returns a pointer to the start of the frame. If a non-NULL value for the optional parameter `r_buffer_end` is supplied, the function will be loaded with a pointer to the last byte in the current buffer. This pointer can be used by the decompressor to protect against bad bitstreams. If you supply a non-NULL value for the optional parameter `r_buffer_end` and set the parameter `need_EOF` to `TRUE`, `r_buffer_end` will be loaded with a pointer to the end of the frame. Requesting a pointer to the end of the frame may be very expensive with regard to time because the frame must be pre-parsed. Normally just a pointer to the end of the buffer is sufficient to protect against reading past valid memory.

After a Frame is Decompressed

`decompressedFrame()` is called by the decompressor when it is done with a frame. The prototype of `decompressedFrame()` is:

```
void decompressedFrame(Xil_unsigned8** bfptr, int type,
                       void* user_ptr);
```

The `XilCisBufferManager` object expects the first parameter, `bfptr`, to be set to one byte past the end of the frame. Several optional parameters exist. The type parameter may specify a positive integer to store as the frame type. The `user_ptr` parameter is assumed to be a pointer to data that the codec has allocated and wishes to associate with the frame. If the value of the `user_ptr` parameter is `NULL`, no change to the current value is made. The `update_next` parameter (flag) is `TRUE` if the function was called after the frame data was processed, not just parsed. The setting of this flag is necessary since there are functions that call `decompressedFrame()` that only establish frame boundaries.

An Alternative to Compressing into a CIS

An alternative to compressing into a CIS is to load the CIS with already compressed data from another bitstream or a file. You can use `putBits()` and `putBitsPtr()` to put data into a buffer of the `XilCisBufferManager` object.

```
void putBits(int nbytes, int nframes, void* data);
```

`putBits()` copies `nbytes` from specified data into a newly allocated `XilCisBuffer` object.

```
void putBitsPtr(int nbytes, int nframes, void* data,
                XIL_FUNCPTR_DONE_WITH_DATA);
```

`putBitsPtr()` creates a new `XilCisBuffer` object whose buffer space references the external storage at a given data pointer. This function has an optional parameter `XIL_FUNCPTR_DONE_WITH_DATA` that is a pointer to a function that can be called when the `XilCisBuffer` object with an external storage pointer is destroyed. This `XilCisBuffer` object can be destroyed because the CIS was reset or destroyed, or the frames in the buffer are no longer necessary to the CIS (see the section "Adjust Start Frame within Buffer Lists" on page 206). The callback can be used to reclaim data space. The default value for this parameter is `NULL`, which means no callback is made.

Both functions expect the parameter `nframes` to be specified with one of the following values:

Value	Meaning
-1	Unknown number of frames
0	May contain partial frames
An integer (*n*) greater than 0	*n* frames

Note – It is very important that a buffer be loaded with a correct value for `nframes`.

Return a Pointer to Data

`getBitsPtr()` returns a pointer to data that has been compressed or loaded into the current read buffer of the `XilCisBufferManager` object, starting at the current read frame. The prototype of the function is:

```
void* getBitsPtr(int* nbytes, int* nframes);
```

The pointer returned is to the start of the read frame; the parameters `nbytes` and `nframes` are loaded with the number of bytes and the number of complete frames in the buffer. If there is not a complete frame of data to return, the pointer is NULL, and `nbytes` and `nframes` are loaded with 0.

Return Data and Frame Information about the CIS

The following three functions return either data or frame information about the CIS:

- `hasData()`
- `numberOfFrames()`
- `hasFrame()`

```
int hasData();
```

This function returns the amount of data in the CIS from the current read position to the end of the CIS.

```
int numberOfFrames();
```

This function returns the number of complete frames in the CIS from the current read position to the end of the CIS.

```
Xil_boolean hasFrame();
```

This function returns `TRUE` if a complete frame exists at the read position of the CIS.

Note – When a CIS is loaded and the frame boundaries are not known, `hasFrame()` and `numberOfFrames()` invoke the `findNextFrameBoundary()` function of the `XilDeviceCompression` class, if necessary, to determine the frame boundaries.

Determine if a Complete Frame Exists

The `XilCisBufferManager` object is responsible for determining if a complete frame exists in the current read buffer for certain functions, such as `nextFrame()`. Frame boundaries are easy to locate in a CIS that was just compressed; the compressor established each frame's start and end. However, if the CIS was loaded with data that contained a partial frame, then the CIS must be parsed to establish the next frame boundary. This parsing is done by each `XilDeviceCompression` object.

When the `XilCisBufferManager` object cannot determine if a complete frame exists, the object saves data about the current read buffer and position, and creates temporary pointers for use by the `XilDeviceCompression` object. Then, the `XilCisBufferManager` object calls the `findNextFrameBoundary()` function of the specific `XilDeviceCompression` object.

`findNextFrameBoundary()` must use the `getNextByte()` and `getNextBytes()` functions of the `XilCisBufferManager` object to get data from the CIS. These functions update the temporary pointers that the `XilCisBufferManager` object created for use by the `XilDeviceCompression` object when parsing.

Xil_unsigned8* getNextByte();

getNextByte() returns a pointer to the next available byte in the CIS. This function handles the transition to the next frame buffer if it reaches the end of the current buffer. If no next buffer exists (the end of the CIS is reached), the function returns NULL.

Xil_unsigned8* getNextBytes(int* nbytes);

getNextBytes() returns a pointer to the next available byte in the CIS and a count of the number of bytes to the end of that buffer. When XilDeviceCompression has parsed through all of the bytes in the buffer and needs to parse the next buffer, it must call getNextBytes() again. If there is no next buffer, getNextBytes() returns NULL.

int foundNextFrameBoundary(Xil_unsigned8* frame_ptr);

If findNextFrameBoundary() is successful in finding the end of the frame, it must call the XilCisBufferManager::foundNextFrameBoundary() function and return this function's status. foundNextFrameBoundary() expects a pointer to one byte beyond the end of the frame, which is the same as the first byte of the next frame. This function resolves the previous state saved by XilCisBufferManager (the state saved before findNextFrameBoundary() was called) and the current state (the state after the getNextByte() and getNextBytes() functions were called). For example, the frame end found by findNextFrameBoundary() may be within the next buffer instead of the current read buffer. This would require the XilCisBufferManager to allocate a new buffer that can hold the entire frame and copy the frame pieces into this new buffer.

The foundNextFrameBoundary() function returns the status of either XIL_SUCCESS or XIL_FAILURE to findNextFrameBoundary(), which then should return the status to its calling function in the XilCisBufferManager object, as shown below:

```
// success, within findNextFrameBoundary
return(cbm->foundNextFrameBoundary(frame_end);
```

If findNextFrameBoundary() did not find the end of the frame and has exhausted all the bytes available in the CIS, it should return XIL_FAILURE. The exception is for a compression type that does not use an end-of-frame marker; the end of this frame is determined by the start of the next frame. In this case, `findNextFrameBoundary()` should return `XIL_UNRESOLVED`. The `XilCisBufferManager` object interprets this status according to whether or not partial frames are present in the buffer.

Over-read Bytes

`ungetBytes()` allows the `XilDeviceCompression` object to return any bytes that it may have over-read when determining the frame end (using `findNextFrameBoundary()`). This function is needed because some compression types do not contain an end-of-frame marker. The next start-of-frame marker must be read and identified before the end-of-frame is known. "Reading ahead" may move the frame tracking pointers to a different `XilCisBuffer` object. Since only data within an `XilCisBuffer` is contiguous, the compression device cannot just subtract *n* bytes from the current pointer to find the end-of-frame pointer. Instead, the compression device must request to return the over-read bytes by using the `ungetBytes()` function. This function will detect and handle backing up the pointer by *n* bytes of valid buffer data.

Seek a Specific Frame within the CIS

The following functions are used for seeking a specific frame within a CIS:

- `seek()`
- `setSeekToStartFrameFlag()`

`int seek(int framenumber, int type);`

The `XilCompressionDevice` object uses the `seek()` function to seek to a specific frame within a CIS. The parameter `framenumber` corresponds to the frame ID of a frame in the CIS. The optional `type` parameter specifies the frame type, which corresponds to the type of a frame in the CIS. This parameter defaults to `XIL_CIS_ANY_FRAME_TYPE` (any frame type), which allows a seek based on position only.

The `XilCisBufferManager` object performs a seek in two stages: the first stage is for position and the second stage is for type. First, `XilCisBufferManager` positions the read frame as close as possible to the desired frame. If the desired frame does not have a frame boundary, the read frame is the closest preceding frame number (this could be the first frame in the CIS, the start frame). The delta between the current read frame and the desired frame number is stored, and the second stage begins.

The second stage positions the read frame based on the requested type. If the frame type is XIL_CIS_ANY_FRAME_TYPE or matches the current read frame type, then the second stage delta is zero. Otherwise, XilCisBufferManager searches backward until it finds the requested frame type. Then, it leaves the read frame at this frame and stores the additional number of frames from the position of the read frame in the second stage delta. The return value is the total of the first stage delta (position) and the second stage delta (frame type).

Note that the XilCisBufferManager object treats the next decompressed frame in a special way. During the seek, the next decompress frame is assigned the type XIL_CIS_ANY_FRAME_TYPE. This assignment ensures that a burn forward into a CIS starts from the last decompressed frame. The next decompress frame is tracked via the update_next parameter of the decompressedFrame() function.

The history_update parameter of the XilDeviceCompression::seek() function can be used to determine the frame type specified to the XilCisBufferManager::seek() function. If history_update is TRUE, then the seek must preserve history. The type specified in the XilCisBufferManager::seek() function must be the appropriate key frame type. If history_update is FALSE, then the seek is for position only, and the type specified in the XilCisBufferManager::seek() function should be the XIL_CIS_NO_BURN_TYPE, which is a special flag to the XilCisBufferManager object to skip the second stage delta. Following is a typical code fragment:

```
if (history_update == TRUE)
   frames_to_burn = cbm.seek(framenumber, seekFrameType);
else
   frames_to_burn = cbm.seek(framenumber, XIL_CIS_NO_BURN_TYPE);
```

Figure 6-2 is a diagram that helps to illustrate the actions taken by the XilCisBufferManager object for a seek.

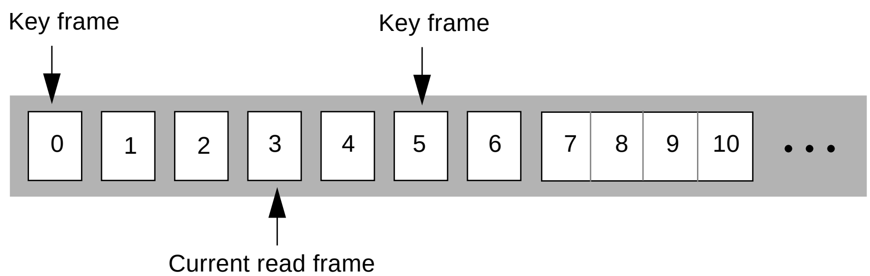

Figure 6-2 Actions Taken by `XilCisBufferManager::seek()`

Figure 6-2 shows key frames at 0 and 5 and a block of four frames starting at frame 7, where boundaries are not known. The current read frame is frame 3 for all examples below.

`cbm->seek(3, xxxx)`	position 3, returns 0, regardless of frame type
`cbm->seek(4, Key)`	position 3, returns 1 (burn 1 forward)
`cbm->seek(2, Key)`	position 0, returns 2 (burn 2 forward)
`cbm->seek(6, Key)`	position 5, returns 1 (burn 1 forward)
`cbm->seek(6, Any)`	position 6, returns 0
`cbm->seek(8, Key)`	position 5, returns 3 (burn 3 forward)
`cbm->seek(8, Any)`	position 7, returns 1 (burn 1 forward)

Note that frames are typed by the function `decompressedFrame()` or `compressedFrame()`.

`void setSeekToStartFrameFlag(Xil_boolean value);`

This function determines the behavior of the `XilCisBufferManager::seek()` function during the second stage, seek to frame type. When the `XilCisBufferManager` object has moved back through the CIS to the start frame and still has not matched with the requested type, it checks the value of the `seek_to_start_frame_flag` parameter. If the value of this parameter is `TRUE`, then the start frame of the CIS is granted the type `XIL_CIS_ANY_FRAME_TYPE`. In this case, the worst case burn starts from the first available frame, which for most compressions is a reasonable option. The default value of the `seek_to_start_frame_flag` parameter is `TRUE` for an initialized (reset) `XilCisBufferManager` object.

Adjust Start Frame within Buffer Lists

`adjustStart()` is used to adjust the start frame within the buffer lists. The prototype of this function is:

`int adjustStart(int framenumber, int type);`

Since the new start frame may not be able to be processed without information from prior frames, `adjustStart()` can take as a parameter a frame type. Any frames from the desired frame back to a frame of the given type are kept within the CIS, although the frames may not be accessed directly via `seek()`. These additional frames are used only to process the new start frame. Once the adjustment is made, any buffers prior to the new start buffer are destroyed.

This function returns an `int` that represents status, either `XIL_SUCCESS` or `XIL_FAILURE`.

Device Compression with Out-of-Order Frames

The following functions discussed in this section provide necessary interfaces for device compression with out-of-order frames to take advantage of the `XilCisBufferManager` class:

- `seekBackToFrameType()`
- `addToLastFrame()`

Out-of-order frames means the frame ID and display ID do not match (MPEG is an example of this type of device compression). For more information about MPEG, see the *XIL Programmer's Guide.*

`int seekBackToFrameType(int type);`

`seekBackToFrameType()` is a special case of seek. It begins looking for the specified frame type at the frame previous to the current read frame. This function does not have `framenumber` as a parameter. It takes identical action to the second stage of the `seek()` function. It positions the current read frame at a frame of the specified type, which must be positioned before the original read frame. The number returned is the number of frames from the current read frame to the original read frame (the read frame upon entry to the function).

`int addToLastFrame(Xil_unsigned8* data, int nbytes);`

Device compression with out-of-order frames uses an end-of-sequence marker, which from the library's standpoint is part of the last frame in the CIS. This function allows the bytes for the end-of-sequence marker to be added after the last frame has already been registered via the `compressedFrame()` function. `addToLastFrame()` automatically increases the frame size by nbytes and adjusts the write pointers of the CIS.

The following functions allow device compression with out-of-order frames to implement its own `getBits` function, which can handle out-of-order frames:

- `getRBuffer()`
- `getNumBytesToFrame()`
- `moveEndStartOneBuffer()`

`Xil_unsigned8* getRBuffer();`

`getRBuffer()` returns a pointer to the current read buffer. This pointer allows the `XilDeviceCompression` object to determine when it has "crossed" the buffer boundary to get to the next frame.

Xil_unsigned8* getNumBytesToFrame(int end_id, int* nbytes);

```
Xil_unsigned8* getNumBytesToFrame(int end_id,
                int* nbytes);
```

`getNumBytesToFrame()` returns a pointer to a data block starting at the current read frame in the current buffer. The number of bytes between the current read frame and the specified end frame (`end_id`) is loaded into `nbytes`. This allows the `XilDeviceCompression` object to get less than all of the frames in the current read buffer by allowing you to specify on which frame to stop.

`int moveEndStartOneBuffer();`

`moveEndStartOneBuffer()` moves the last frame of the current buffer and the start frame of the next buffer into a new buffer and inserts it into the buffer list. This function is used to move a predictive frame at the buffer end and bidirectional frame at the next buffer start into the same buffer.

Error Handling and Recovery

The following functions discussed in the section are provided as hooks for error handling and recovery:

- `byteError()`
- `nextSeek()`
- `prevSeek()`
- `nextKnownFrameBoundary()`
- `errorRecoveryDone()`
- `foundFrameDuringRecovery()`

Currently, these functions are not used by the XIL Imaging Library.

`void byteError(Xil_unsigned8* bptr);`

`byteError()` is called when a decompressor finds a bitstream error during `decompress()` or `findNextFrameBoundary()`. The byte pointer input parameter should be set to the location of the bitstream error. Doing this sets up the next byte that `getNextByte()` returns.

`int nextSeek(int framenumber, int type);`

`nextSeek()` determines the closest frame that is greater than or equal to the given `framenumber`. It returns -1 when no such “seekable” frame exists.

`int prevSeek(int framenumber, int type);`

`prevSeek()` determines the closest frame that is less than or equal to the given `framenumber`. It returns -1 if no such “seekable” frame exists.

```
void nextKnownFrameBoundary(Xil_unsigned8* cptr,
              Xil_unsigned8** fptr int* num_frames);
```

`nextKnownFrameBoundary()` returns a pointer to the next established frame boundary in relation to a pointer in the current buffer (`cptr`). It returns the pointer in `fptr` and a number of frames (`num_frames`) between the current position and the known boundary, including the current frame.

```
void errorRecoveryDone(Xil_unsigned8* fptr,
                   int num_frames, Xil_boolean fixed);
```

`errorRecoveryDone()` is called by `xil_cis_attempt_recovery()` just before it completes. It expects the current pointer, the number of frames parsed, and the state of the recovery.

```
void foundFrameDuringRecovery(Xil_unsigned8* fptr);
```

`foundFrameDuringRecovery()` expects a pointer to the start of the next frame, which established previous bytes as part of the previous frame.

Adding a New Compression Method

The complexity of adding a new compression type to the XIL library varies widely, depending on the compression technology. In order to install a compressor like JPEG or CCITT G3, very little work has to be done other than to actually write the compression and decompression functions and the few pure virtual functions. The default implementation for the `XilDeviceCompression` will likely work. For a more complicated compression technique, like MPEG 1, with its multiple frame types and out-of-order transmission, relatively little of the default implementation may be used. In either case, however, the general steps to add a compression technique are the same.

Table 6-1 lists the classes that you must create, the functions that you must implement, and the functions that you optionally can implement to add a new compression method.

Table 6-1 Required and Optional Functions for Adding a New Compression Method

Class/Function	Required	Optional
Derived class from `XilDeviceCompressionType` class	X	
`createDeviceCompression()`	X	
Derived class from `XilDeviceCompression` class	X	
`findNextFrameBoundary()`	X	
`burnFrames()`	X	
`getMaxFrameSize()`	X	
`reset()`	X	

Table 6-1 Required and Optional Functions for Adding a New Compression Method

Class/Function	Required	Optional
`deriveOutputType()`	X	
`decompressHeader()`		X
`seek()`		X
`flush()`		X
`adjustStart()`		X
`getBitsPtr()`		X
`putBits()`		X
`putBitsPtr()`		X
`hasData()`		X
`numberOfFrames()`		X
`hasFrame()`		X
`setInputType()`		X
`attemptRecovery()`		X

1. **Create the derived class from `XilDeviceCompressionType`.**
   For example, the code at the end of this chapter defines `XilDeviceCompressionTypeIdentity`. The global function `XilCreateCompressionType()` must be written to create a single instance of this derived class. If the compression technique contains exposed attributes, these should be registered here by calling `registerAttr()` from within the constructor for `XilDeviceCompressionType`. Finally, the pure virtual member function `createDeviceCompression()` must be written. Usually, this involves calling the constructor for the `XilDeviceCompression` derived class.

2. **Next, a class derived from `XilDeviceCompression` must be created.**
   In the example, the class `XilDeviceCompressionIdentity` is created. Table 6-1 lists the pure virtual functions that must be implemented in the derived class and the ones that are optional. Optional functions need only be implemented if the default implementation inherited from `XilDeviceCompression` is inadequate.

3. **The actual compression and decompression functions are added to a compute handler, as described in Chapter 4, "Compute Devices."**
   The names of these functions should be `compress_`*compression_name* and `decompress_`*compression_name*, where *compression_name* is the name of the new compression type. In the section "Sample Compressor" on page 214 the example derives a compute class called `XilDeviceComputeTypeIdentity`. It contains three members: `describeMembers()`, `compress_Identity()` and `decompress_Identity()`. `describeMembers()` is generated automatically as described in Chapter 4, "Compute Devices."

   The name of the created compute module should look like this:

   `xilcompute`*Compressorname_COMPANYNAME*`memory.so.`*major_ver_no*

   In the case of the example, the compute module is named `xilcomputeIdentity_SUNWmemory.so.1`

4. **Finally, the** `/opt/SUNWits/Graphics-sw/xil/lib/xil.compute` **configuration file must be updated to show the new compression type.**
   The appropriate line to add looks like this:

   `compute`*Compressorname_COMPANYNAME*`memory` *Compressorname*

   This indicates the dependence on the compression handler by the compute handler, which implements the compression and decompression. More information on installing handlers can be found in Chapter 2, "More on Writing Device Handlers."

Adding Compression Hardware

Hardware to support compression usually falls into two categories. In the first, the device operates on memory, doing compression using a fast special purpose processor, and then putting the results back into memory. Adding support into the XIL library for this type of hardware can be a simple as rewriting a single function (or pair of functions, if the device is capable of compression and decompression). The second type of device is usually tied to input or output: a frame grabber with built-in JPEG compression, for example, or a JPEG decompress board with the ability to map windows onto the screen. In this case, it is necessary to write the appropriate I/O handler for the device, and write a molecule to perform the capture/compress or the decompress/display function. If the desired compression format is not one that is currently available, the entire compression handler must be created in the manner described in the previous section.

The simpler case is for devices that are not associated with input or output, and for a compression type that currently exists in the XIL library (say, JPEG). In the simplest fashion, porting this type of device requires subclassing the `XilDeviceComputeType`, just like what is done to add accelerator support for any XIL operator. The new functions would be named `compress_`*Compressorname*`()` and `decompress_`*Compressorname*`()`, where *Compressorname* is the name of the compression type that is being supported. The function `describeMembers()` must be generated using the method described in Chapter 4, “Compute Devices.” The new compute handler containing the compression functions should be called

`xilcompute`*Compressorname_COMPANYNAMEdevicename*`.so`

where *`devicename`* is the name for the accelerator device. A line in the `xil.compute` configuration file should be added as follows:

`compute`*Compressorname_COMPANYNAMEdevicename* *Compressorname*

As described in Chapter 4, “Compute Devices,” adding this compute device will replace the default function called for `compress()`. The implementation of `compress()` and `decompress()` is required to put their compressed data and decompressed images back in the CPU memory when the operation is done. If a device also has other capabilities, such as doing RGB to YCC color conversion, then it would also be advantageous to provide a molecule `compress(color_convert())`, for example. Molecules are added in this case exactly like the noncompression case described in Chapter 4, “Compute Devices.”

If the device contains integrated input or output, the situation is slightly different. First, in order for the XIL library to expose the device as a device image to the application, an I/O handler must be written. This procedure is described in Chapter 3, “I/O Devices.” In most cases, it is advantageous to provide such a handler even for the cases where compression or decompression are not performed (raw frame grab or displaying uncompressed data), if the hardware supports such capabilities. In order to provide the decompression capabilities of the device, a molecule must be written that supports `display(decompress())` or `compress(capture())`. This is also discussed in Chapter 3, “I/O Devices,” where the interaction of compute and I/O handlers is discussed. Again, it may be advantageous to provide other molecules to support whatever functionality the hardware has: color conversion or zoom, for example.

After the I/O handler is written, the compute handler must be written. The situation is the same as described above. The compute module should be called

`xilcompute`*Compressorname_COMPANYNAMEdevicename*`.so`

just as before. However, this time, there is an added notation in the configuration file that indicates the dependence on the I/O handler for the device:

`compute`*Compressorname_COMPANYNAMEdevicename*`\`
    Compressorname   *ioCOMPANYNAMEdevicename*

These two dependency entries in `xil.compute` would reference the following modules:

`xil`*Compressorname*`.so`

and

`xilio`*devicename*`.so`

The first module contains the generic compression information for the compression type, and the second contains the generic I/O handler for the accelerator device.

Finally, I/O compression devices with associated image storage may also be defined. Chapter 5, “Storage Devices,” describes storage devices in detail.

6

Sample Compressor

The code in this sample implements an example *identity* compression. In this lossless compression, raw image data is simply put into the CIS in a predefined manner.

The example contains four files:

- `XilDeviceCompressionTypeIdentity.h` and `XilDeviceCompressionTypeIdentity.cc`, which defines the device compression type
- `XilDeviceCompressionIdentity.h` and `XilDeviceCompressionIdentity.cc`, which defines the identity device compression itself
- `compress_Identity.cc`, which encodes the images into the CIS
- `decompress_Identity.cc`, which decodes the images from the CIS

XilDeviceCompressionTypeIdentity.h

Code Example 6-5 XilDeviceCompressionTypeIdentity.h

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:XilDeviceCompressionTypeIdentity.h
//  Project:XIL
//  Created:93/04/14
//  Revision:1.1
//  Last Mod:12:05:08, 07 Mar 1994
//
//  Description:
//  This is the class that maintains the Identity compression
//  type information.  It is derived from the more generic
//  XilDeviceCompressionType class and is responsible for
//  registering the attribute setting/getting functions for
//      Identity compression and decompression.
//
//      The class is also used to maintain information which is not
//      specific to any single instantiation of the Identity
//      compressor.  There will be only one instantiation of this
//      class for the Identity compression irregardless of how many
//      XilCis objects are created.
//
//-----------------------------------------------------------------------
#pragma ident“@(#)XilDeviceCompressionTypeIdentity.h1.1\t94/03/07  “

#ifndef XilDeviceCompressionTypeIdentity_H
#define XilDeviceCompressionTypeIdentity_H

#include <xil/XilError.h>
#include <xil/XilCis.h>
#include <xil/XilDeviceCompressionType.h>

classXilDeviceCompressionTypeIdentity : public XilDeviceCompressionType
{
public:
    virtual XilDeviceCompression*  createDeviceCompression(XilCis*  xcis);

    //
    //  The constructor is moved into the public space here because
    //  this derived class can be created, but the parent class is not
    //  permitted to be created without being derived upon.
```

Code Example 6-5 `XilDeviceCompressionTypeIdentity.h` *(Continued)*

```
    //
    XilDeviceCompressionTypeIdentity(void);
    ~XilDeviceCompressionTypeIdentity(void);
};

#endif  XilDeviceCompressionTypeIdentity_H
```

XilDeviceCompressionTypeIdentity.cc

Code Example 6-6 XilDeviceCompressionTypeIdentity.cc *(1 of 3)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      XilDeviceCompressionTypeIdentity.cc
//  Project:   XIL
//  Created:   93/04/14
//  Revision:  1.2
//  Last Mod:  09:31:22, 22 Mar 1994
//
//  Description:
//-----------------------------------------------------------------------
#pragma ident“@(#)XilDeviceCompressionTypeIdentity.cc1.2\t94/03/22  “


#include “XilDeviceCompressionTypeIdentity.h”
#include “XilDeviceCompressionIdentity.h”

//-----------------------------------------------------------------------
//
//  Function:  XilCreateCompressionType
//  Created:   93/04/14
//
//  Description:
//  The XilCreateCompressionType() is called when the XIL core
//  opens the xilIdentity.so library.  XilCreateCompressiontype()
//  is responsible for creating the Identity compression type class.
//
//-----------------------------------------------------------------------

XilDeviceCompressionType*
XilCreateCompressionType()
{
    XilDeviceCompressionTypeIdentity* device;

    device = new XilDeviceCompressionTypeIdentity();
    if(device==NULL) {
    // out of memory
    XIL_ERROR(NULL, XIL_ERROR_RESOURCE,”di-1”,TRUE);
    }
```

Code Example 6-6 `XilDeviceCompressionTypeIdentity.cc` *(2 of 3)*

```
    return device;
}

//------------------------------------------------------------------------
//
//  Function:  XilDeviceCompressionTypeIdentity::createDeviceCompression()
//  Created:   93/04/14
//
//  Description:
// createDeviceCompression() is used to create new instances of
//  the Identity device compression when new CISs are created by the
//  user.
//
//------------------------------------------------------------------------
XilDeviceCompression*
XilDeviceCompressionTypeIdentity::createDeviceCompression(XilCis* xcis)
{
    XilDeviceCompressionIdentity* device;

    //
    //  Create a new XilDeviceCompressionIdentity
    //
    device = new XilDeviceCompressionIdentity(this, xcis);
    if(device == NULL) {
        // out of memory
        XIL_ERROR(xcis->getSystemState(), XIL_ERROR_RESOURCE,"di-1",TRUE);
    }

    //
    //  Check to see if the device construction was completed
    //  successfully.
    //
    device = device->ok();
    if(device == NULL) {
        //  Couldn't create internal base XilDeviceCompression object
        XIL_ERROR(xcis->getSystemState(), XIL_ERROR_SYSTEM,"di-278", FALSE);
    }

    return device;
}
```

Code Example 6-6 `XilDeviceCompressionTypeIdentity.cc` *(3 of 3)*

```
//------------------------------------------------------------------------
//
//  Function:  XilDeviceCompressionTypeIdentity()
//  Created:   93/04/14
//
//  Description:
//  The device compression type constructor initializes any
//  Identity compression type specific data and registers all of the
//  Identity attributes with the XIL core.
//
//------------------------------------------------------------------------

XilDeviceCompressionTypeIdentity::XilDeviceCompressionTypeIdentity()
: XilDeviceCompressionType("Identity","IDENTITY")
{

   // any attributes which the codec would like to provide access
   // to via the xil_cis_set_attribute() and/or xil_cis_get_attribute()
   // bindings must be registered here.
   // NOTE: These attribute functions are registered here as an
   // example ONLY...the Identity codec does not make use of
   // "quality" ...it is just an example of the registerAttr mechanism.

        registerAttr("COMPRESSION_QUALITY",

(setAttrFunc)XilDeviceCompressionIdentity::setCompressionQuality,

(getAttrFunc)XilDeviceCompressionIdentity::getCompressionQuality);

        registerAttr("DECOMPRESSION_QUALITY",

(setAttrFunc)XilDeviceCompressionIdentity::setDecompressionQuality,

(getAttrFunc)XilDeviceCompressionIdentity::getDecompressionQuality);
}

XilDeviceCompressionTypeIdentity::~XilDeviceCompressionTypeIdentity(void) { }
```

XilDeviceCompressionIdentity.h

Code Example 6-7 XilDeviceCompressionIdentity.h *(1 of 3)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//------------------------------------------------------------------------
//
//  File:      XilDeviceCompressionIdentity.h
//  Project:   XIL
//  Created:   93/04/14
//  Revision:  1.2
//  Last Mod:  09:29:16, 22 Mar 1994
//
//  Description:
//  The file contains the definitions for Identity compression and
//  decompression.  Each Identity cis has its own instantiation of
//  this class.
//
//  The Identity bit stream has the following format:
//
//          [ 32-bit INTEGER ]    width
//          [ 32-bit INTEGER ]    height
//          [ 32-bit INTEGER ]    nbands
//          [ IMAGE DATA ]
//
//  NOTE:  The code included here to implement this bitstream
//  creates a compressed stream which is not portable between
//  different endian machines (i.e. x86 <--> SPARC).
//
//------------------------------------------------------------------------
#pragma ident“@(#)XilDeviceCompressionIdentity.h1.2\t94/03/22  “

#ifndef XilDeviceCompressionIdentity_H
#define XilDeviceCompressionIdentity_H

#include <xil/XilError.h>
#include <xil/XilCis.h>
#include <xil/XilImage.h>
#include <xil/XilDeviceCompression.h>

#define FRAMES_PER_BUFFER  3
#define IDENTITY_FRAME_TYPE 1
```

Code Example 6-7 `XilDeviceCompressionIdentity.h` *(2 of 3)*

```
class XilDeviceCompressionIdentity : public XilDeviceCompression
{
public:
    XilDeviceCompressionIdentity(XilDeviceCompressionType* xdct,
                                 XilCis*                   cis);
    ~XilDeviceCompressionIdentity(void);

    int comp_quality;           //compression quality attribute
    int decomp_quality;         // decompression quality attribute
    Xil_boolean derivedType;    // flag for derived type from bitstream

    //
    //  Pure virtual member functions of XilDeviceCompression which I
    //  must implement.
    //
    int             getMaxFrameSize(void);
    void            burnFrames(int nframes);
    int             findNextFrameBoundary(void);


    //  Allocation/Creation verification member function
    //
    XilDeviceCompressionIdentity* ok(Xil_boolean destroy = TRUE);

    //
    //  Function to read header and fill in the header information --
    //  specifically width and height
    //
    int   deriveOutputType(void);

    //
    //  Function to reset the codec state, destroy old inputType
    //
    voidreset();

    //
    //  Virtual member functions of XilDeviceCompression which I've
    //  chosen to implement because the default functions do not work
    //  for the Identity codec.
    //  the Identity codec marks even frames with its own
    //  frame type; this is done in order to illustrate how a codec
    //  with typed frames would interface with the cbm
```

Code Example 6-7 `XilDeviceCompressionIdentity.h` *(3 of 3)*

```
    void        seek(int framenumber, Xil_boolean history_update=TRUE);
    int         adjustStart(int framenumber);

    // functions for registered attribute set/get
    void       setCompressionQuality(int value);
    int        getCompressionQuality();
    void       setDecompressionQuality(int value);
    int        getDecompressionQuality();

private:
    Xil_boolean     isok;

    //
    //  Function used by reset and the constructor to set values
    //
    int    initValues();
};

#endif
```

XilDeviceCompressionIdentity.cc

Code Example 6-8 XilDeviceCompressionIdentity.cc *(1 of 9)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      XilDeviceCompressionIdentity.cc
//  Project:   XIL
//  Created:   93/04/14
//  Revision:  1.3
//  Last Mod:  08:37:54, 28 Mar 1994
//
//  Description:
//  Contains the member functions of XilDeviceCompressionIdentity.
//
//
//
//
//
//
//
//-----------------------------------------------------------------------
#pragma ident  "@(#)XilDeviceCompressionIdentity.cc1.3\t94/03/28  "

#include "XilDeviceCompressionIdentity.h"

XilDeviceCompressionIdentity*
XilDeviceCompressionIdentity::ok(Xil_boolean destroy) {
    if(this == NULL) {
        return NULL;
    } else {
        if(XilDeviceCompression::ok(FALSE) == this && isok == TRUE) {
            return this;
        } else {
            if(destroy == TRUE) delete this;
            return NULL;
        }
    }
}

int             XilDeviceCompressionIdentity::getMaxFrameSize(void) {
    return ((int)inputType->getWidth()*(int)inputType->getHeight()
              *inputType->getBands() + 12);
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(2 of 9)*

```
}

int
XilDeviceCompressionIdentity::initValues()
{
    XilImageType* t =
        getCis()->getSystemState()->createImageType(0,0,0,XIL_BYTE);

    XIL_SIMULATE_FAILURE(992, t=NULL);
    if(t == NULL) {
        // out of memory
       XIL_ERROR(getCis()->getSystemState(), XIL_ERROR_RESOURCE,"di-1",TRUE);
        return XIL_FAILURE;
    }

    inputType = outputType = t;

    // output type has not yet been derived from bitstream
    derivedType = FALSE;

    // reset any attributes to default state
    comp_quality = 0;
    decomp_quality = 0;

    return XIL_SUCCESS;
}

void
XilDeviceCompressionIdentity::reset()
{
    if (inputType != outputType)
    outputType->destroy();
    inputType->destroy();

    initValues();
    XilDeviceCompression::reset();
}

//
//  FRAMES_PER_BUFFER is a recommendation on the size of each buffer inside the
//  CBM.
//
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(3 of 9)*

```
XilDeviceCompressionIdentity::XilDeviceCompressionIdentity
      (XilDeviceCompressionType* xdct,XilCis* cis)
: XilDeviceCompression(xdct, cis, 0, FRAMES_PER_BUFFER)
{
    isok = FALSE;

    if(XilDeviceCompression::ok(FALSE) == NULL) {
        //  Couldn't create internal base XilDeviceCompression object
        XIL_ERROR(getCis()->getSystemState(), XIL_ERROR_SYSTEM,"di-278",
FALSE);
        return;
    }

    if(initValues() == XIL_FAILURE) {
        //  Couldn't create internal Identity compressor object
        XIL_ERROR(getCis()->getSystemState(), XIL_ERROR_SYSTEM,"di-275",
FALSE);
        return;
    }

    isok = TRUE;
}

XilDeviceCompressionIdentity::~XilDeviceCompressionIdentity(void) { }

//
//  Function to read header and fill in the ImageType information
//
int
XilDeviceCompressionIdentity::deriveOutputType(void)
{
   // derivedType flags if the type has been derived from
   // the bitstream--prevents an infinite loop when neither the
   // boundary nor type of the first frame in the CIS have been
   // established
    if (derivedType == FALSE) {
       //
       //  This call will ensure that there is an entire frame for me to
       //  look through.  If necessary, the cbm will call this class's
       //  findNextFrameBoundary to parse the bitstream and
       //  locate the end of the frame.
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(4 of 9)*

```
        //
        Xil_unsigned32* bp32 =
          (Xil_unsigned32*)cbm.nextFrame();
        if(bp32 == NULL) {
           return XIL_FAILURE;
        }

        //
        //  NOTE:  This doesn't produce an endian-portable bitstream.
        //
        unsigned int image_width  = *bp32++;
        unsigned int image_height = *bp32++;
        unsigned int image_bands  = *bp32++;

        if(image_width && image_height && image_bands) {
           XilImageType*  newtype =
         cis->getSystemState()->createImageType(image_width, image_height,
                                              image_bands, XIL_BYTE);

           XIL_SIMULATE_FAILURE(993, newtype=NULL);
           if(newtype == NULL) {
              // out of memory
              XIL_ERROR(getCis()->getSystemState(), XIL_ERROR_RESOURCE,"di-
1",TRUE);
              return XIL_FAILURE;
           }

           //
           //  This will also set the outputType as a side-effect
           //
           setInputType(newtype);
           newtype->destroy();// destroy copy
           derivedType=TRUE;
        }
    }
    return XIL_SUCCESS;
}


void
XilDeviceCompressionIdentity::burnFrames(int nframes)
{
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(5 of 9)*

```
    int frame_type;

    // In order to illustrate “key” frames,
    // this codec marks even frames with its own frame type
    // This illustrates the use of frame type with the
    // compressedFrame/decompressFrame/seek/adjustStart functions
    // (Of course, codecs generally have a much better reason
    // to mark a frame as a “key” frame!)

    //
    //  Get the information about the CIS image type.
    //
    XilImageType*  cis_outtype    = getOutputType();
    unsigned int   cis_width      = cis_outtype->getWidth();
    unsigned int   cis_height     = cis_outtype->getHeight();
    unsigned int   cis_bands      = cis_outtype->getBands();

    //
    //  Compute how far the next frame should be...
    //
    unsigned long  frame_size =
        cis_width*cis_height*cis_bands + 3*sizeof(Xil_unsigned32);

    for(int i=0; i<nframes; i++) {

       Xil_unsigned8* bp = (Xil_unsigned8*)cbm.nextFrame();

       // Get the frame number of the burn frame
       if (cbm.getRFrameId() & 0x1)
         // odd frame, no special frame type
         frame_type = XIL_CIS_DEFAULT_FRAME_TYPE;
       else
         // even frame, mark it as our key frame
         frame_type = IDENTITY_FRAME_TYPE;

       bp += frame_size;
       cbm.decompressedFrame(bp,frame_type);
    }
}
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(6 of 9)*

```
//
//  Function to find the next frame boundary
//
int
XilDeviceCompressionIdentity::findNextFrameBoundary(void)
{
    Xil_unsigned8* bp;
    unsigned long  frame_size;

    if (derivedType==FALSE) {
       unsigned int   image_dimensions[3] = {0,0,0};
       unsigned int   i,j;

       // not yet derived input/output type
       // cannot call getOutputType because we will recurse on this function!
       // parse bitstream bytes to get width/height/bands
       for (i=0;i<3;i++) {
          for (j=0;j<sizeof(Xil_unsigned32);j++) {
             if ((bp=cbm.getNextByte())==NULL)
               // here if no more bytes in buffer--failed!
               return XIL_FAILURE;
             // accumulate bytes into current dimension
             image_dimensions[i] = (image_dimensions[i]*256) + *bp;
          }
       }
       //
       //  Compute how far we have to advance the pointer.
       //
       frame_size =
         image_dimensions[0]*image_dimensions[1]*image_dimensions[2];
    }
    else {
       unsigned int   image_width;
       unsigned int   image_height;
       unsigned int   image_bands;
       //
       //  Get the information about the CIS image type.
       //  will cause deriveOutputType() to be called if
       //  outputType not yet established.
       //
       XilImageType*  cis_outtype    = getOutputType();
       image_width      = cis_outtype->getWidth();
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(7 of 9)*

```
        image_height     = cis_outtype->getHeight();
        image_bands      = cis_outtype->getBands();

        //
        //  Compute how far we have to advance the pointer.
        //
        frame_size =
          image_width*image_height*image_bands + 3*sizeof(Xil_unsigned32);
    }

    //
    //  Run through the frame one byte at a time up to the second to
    //  last byte in the frame.  The final byte will be set to the
    //  return value of getNextByte() -- as opposed to updating it
    //  on every cycle of the loop.
    //

    for(int i=0; i<frame_size - 1; i++) {
        if(cbm.getNextByte() == NULL)     return XIL_FAILURE;
    }
    if((bp = cbm.getNextByte()) == NULL)  return XIL_FAILURE;

    //
    //  Tell the CisBufferManager where the frame boundary is...
    //
    return cbm.foundNextFrameBoundary(bp + 1);
}

void
XilDeviceCompressionIdentity::seek(int framenumber, Xil_boolean
history_update)
{
    int frames_to_burn;

    if (history_update == TRUE)
      // when history_update is true, if we have key frames
      // then we must seek with respect to the key frame.
      // The “frames_to_burn” returned by the cbm
      // will start from a key frame, which means our history remains
      // correct
      frames_to_burn = cbm.seek(framenumber, IDENTITY_FRAME_TYPE);
    else
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(8 of 9)*

```
      // when history_update is false, then we are interested
      // in position only for this seek.  Flag the cbm that
      // there should be no burn frames for frame type.
      frames_to_burn = cbm.seek(framenumber, XIL_CIS_NO_BURN_TYPE);

    if(frames_to_burn > 0) {
    burnFrames(frames_to_burn);
    }
}


int
XilDeviceCompressionIdentity::adjustStart(int new_start_frame)
{
   //Called by the compression core to indicate that existing
   //frames prior to the frame number given are not to be retained
   //any longer due to KEEPFRAMES or MAXFRAMES requirements.
   //We'll just simply call the XilCisBufferManager and tell it
   //which type of frame MUST be kept and let it do any actual
   //deleting of data.

    return cbm.adjustStart(new_start_frame, IDENTITY_FRAME_TYPE);
}

// NOTE: the Identity codec does not make use of
// "quality" ...these functions are here to illustrate
// the XilDeviceCompressionIdentityType registerAttr mechanism.

void
XilDeviceCompressionIdentity::setCompressionQuality(int value)
{
   comp_quality = value;
}

int
XilDeviceCompressionIdentity::getCompressionQuality()
{
   return comp_quality;
}

void
XilDeviceCompressionIdentity::setDecompressionQuality(int value)
```

Code Example 6-8 `XilDeviceCompressionIdentity.cc` *(9 of 9)*

```
{
   decomp_quality = value;
}

int
XilDeviceCompressionIdentity::getDecompressionQuality()
{
   return decomp_quality;
}
```

XilDeviceComputeTypeIdentityMemory.h

Code Example 6-9 XilDeviceComputeTypeIdentityMemory.h

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      XilDeviceComputeTypeIdentityMemory.h
//  Project:   XIL
//  Created:   93/04/15
//  Revision:  1.2
//  Last Mod:  09:32:01, 22 Mar 1994
//
//  Description:
//  Contains the definitions of the derived XilDeviceComputeType
//  for Identity compression and decompression.
//
//-----------------------------------------------------------------------
#pragma ident"@(#)XilDeviceComputeTypeIdentityMemory.h1.2\t94/03/22  "

#include <xil/XilDeviceComputeType.h>
#include "XilDeviceCompressionIdentity.h"

class XilDeviceComputeTypeIdentityMemory : public XilDeviceComputeType {
public:
    XilDeviceComputeTypeIdentityMemory()
    : XilDeviceComputeType("XilDeviceCompIdentityMemory") {};

    int describeMembers();

    //
    //  Compress
    //
    int  compress_Identity(XilOp* op, int op_count);

    //
    //  Decompress
    //
    int  decompress_Identity(XilOp* op, int op_count);


    ~XilDeviceComputeTypeIdentityMemory();
};
```

XilDeviceComputeTypeIdentityMemory.cc

Code Example 6-10 XilDeviceComputeTypeIdentityMemory.cc

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      XilDeviceComputeTypeIdentityMemory.cc
//  Project:   XIL
//  Created:   93/04/15
//  Revision:  1.2
//  Last Mod:  09:32:03, 22 Mar 1994
//
//  Description:
//
//
//
//
//
//
//
//
//-----------------------------------------------------------------------
#pragma ident"@(#)XilDeviceComputeTypeIdentityMemory.cc1.2\t94/03/22  "

#include <xil/xili.h>
#include "XilDeviceComputeTypeIdentityMemory.h"

XilDeviceComputeType* XilCreateComputeType()
{
    XilDeviceComputeTypeIdentityMemory* device;

    device= new XilDeviceComputeTypeIdentityMemory();

    XIL_SIMULATE_FAILURE(942, delete device;device=NULL);
    if(device==NULL) {
    // out of memory error
    XIL_ERROR( NULL, XIL_ERROR_RESOURCE,"di-1",TRUE);
    return NULL;
    }

    device->describeMembers();

    return(device);
```

Code Example 6-10 `XilDeviceComputeTypeIdentityMemory.cc` *(Continued)*

```
}

XilDeviceComputeTypeIdentityMemory::~XilDeviceComputeTypeIdentityMemory()
{
}
```

compress_Identity.cc

Code Example 6-11 `compress_Identity.cc` *(1 of 4)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-----------------------------------------------------------------------
//
//  File:      compress_Identity.cc
//  Project:   XIL
//  Created:   93/04/15
//  Revision:  1.2
//  Last Mod:  09:32:32, 22 Mar 1994
//
//  Description:
//
//
//
//
//
//
//
//
//-----------------------------------------------------------------------
#pragma ident"@(#)compress_Identity.cc1.2\t94/03/22  "

#include <xil/XilRoi.h>
#include <xil/XilRoiList.h>
#include <xil/XilOp.h>
#include "XilDeviceComputeTypeIdentityMemory.h"
#include "XilDeviceCompressionIdentity.h"

/* XILCONFIG: compress_Identity= compress_Identity()  */
int
XilDeviceComputeTypeIdentityMemory::compress_Identity(XilOp* op, int)
{
   int frame_type;

    //
    //  Get the source image off of the DAG.
    //
    XilImage* src = op->getSrc1();
```

Code Example 6-11 `compress_Identity.cc` *(2 of 4)*

```
    //
    //  Get the XilDeviceCompression associated with this CIS
    //
    XilDeviceCompressionIdentity* dc = (XilDeviceCompressionIdentity*)
    (op->getDstCis())->getDeviceCompression();

    //
    //  Get the system state.
    //
    XilSystemState* systemState = src->getSystemState();

    // In order to illustrate “key” frames,
    // this codec marks even frames with its own frame type
    // This illustrates the use of frame type with the
    // compressedFrame/decompressFrame/seek/adjustStart functions
    // (Of course, codecs generally have a much better reason
    // to mark a frame as a “key” frame!)

    // Get the frame number of the compress
    if (op->getLongParam(1) & 0x1)
      // odd frame, no special frame type
      frame_type = XIL_CIS_DEFAULT_FRAME_TYPE;
    else
      // even frame, mark it as our key frame
      frame_type = IDENTITY_FRAME_TYPE;

    //
    //  Local copies of image type information.
    //
    XilImageType*  cis_intype = dc->getOutputType();
    unsigned int   cis_width  = cis_intype->getWidth();
    unsigned int   cis_height = cis_intype->getHeight();
    unsigned int   cis_bands  = cis_intype->getBands();

    //
    //  No ROI clipping or non-zero origins are allowed for compressions.
    //  Also, the image width and  image height must match the size of the CIS.
    //  Both of these conditions are checked in XilCis::compress().  So, no
    //  check is required here.
    //
```

Code Example 6-11 `compress_Identity.cc` *(3 of 4)*

```
//
//  Get the next buffer to compress into.
//
XilCisBuffer* cisbuf = dc->getCisBufferManager()->nextBuffer();

//
//  Write the image parameters into the byte-stream
//
cisbuf->addBytes((Xil_unsigned8*)&cis_width, sizeof(cis_width));
cisbuf->addBytes((Xil_unsigned8*)&cis_height, sizeof(cis_height));
cisbuf->addBytes((Xil_unsigned8*)&cis_bands, sizeof(cis_bands));

//
//  Get the source image’s memory storage.
//
long x_origin, y_origin;
src->getOrigin(&x_origin,&y_origin);

//
//  Actually perform the compression into the CisBuffer
//
XilMemoryStorageByte* src_mem =
    (XilMemoryStorageByte*)src->getMemoryStorage();

Xil_unsigned8* src_data        = src_mem->data;

Xil_unsigned8* src_scanline =
    src_mem->data +
    (y_origin * src_mem->scanline_stride) +
    (x_origin * src_mem->pixel_stride);

Xil_unsigned8* src_pixel;
Xil_unsigned8* src_band;

for(int i=0; i<cis_height; i++) {
    src_pixel = src_scanline;
    for(int j=0; j<cis_width; j++) {
        src_band = src_pixel;
        for(int k=0; k<cis_bands; k++) {
            cisbuf->addByte(*src_band);
            src_band++;
        }
```

Code Example 6-11 `compress_Identity.cc` *(4 of 4)*

```
                src_pixel += src_mem->pixel_stride;
            }
            src_scanline += src_mem->scanline_stride;
        }

        dc->getCisBufferManager()->compressedFrame(frame_type);

        return XIL_SUCCESS;
    }
```

decompress_Identity.cc

Code Example 6-12 decompress_Identity.cc *(1 of 4)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//-------------------------------------------------------------------------
//
//   File:      decompress_Identity.cc
//   Project:   XIL
//   Created:   93/04/15
//   Revision:  1.2
//   Last Mod:  09:33:15, 22 Mar 1994
//
//   Description:
//
//
//
//
//
//
//
//
//-------------------------------------------------------------------------
#pragma ident"@(#)decompress_Identity.cc1.2\t94/03/22  "

#include <xil/XilOp.h>
#include <xil/XilDefines.h>
#include "XilDeviceComputeTypeIdentityMemory.h"

/* XILCONFIG: decompress_Identity= decompress_Identity()  */

#define IDENTITY_BYTESTREAM_ERROR(bp,ftype) \
{ \
   dc->getCisBufferManager()->decompressedFrame((Xil_unsigned8*)bp,ftype); \
   XIL_CIS_ERROR(XIL_ERROR_SYSTEM, "di-285", TRUE, dc, FALSE, FALSE); \
   return XIL_FAILURE; \
}

int
XilDeviceComputeTypeIdentityMemory::decompress_Identity(XilOp* op, int)
{
    int frame_type;
```

Code Example 6-12 `decompress_Identity.cc` *(2 of 4)*

```
//
//  Get the destination image off of the DAG
//
XilImage*                 dst = op->getDst();

//
//  Get the XilDeviceCompression associated with this CIS
//
XilDeviceCompressionIdentity* dc  = (XilDeviceCompressionIdentity*)
(op->getSrcCis())->getDeviceCompression();

//
//  The frame number which we’re supposed to decompress is
//  specified by the first parameter on the Op.  So, we’ll seek to
//  that frame.
//
dc->seek((int)op->getLongParam(1));

// In order to illustrate “key” frames,
// this codec marks even frames with its own frame type
// This illustrates the use of frame type with the
// compressedFrame/decompressFrame/seek/adjustStart functions
// (Of course, codecs generally have a much better reason
// to mark a frame as a “key” frame!)

// Test odd/even frame for decompress
if (op->getLongParam(1) & 0x1)
  // odd frame, no special frame type
  frame_type = XIL_CIS_DEFAULT_FRAME_TYPE;
else
  // even frame, mark it as our key frame
  frame_type = IDENTITY_FRAME_TYPE;

//
//  Get the information about the CIS image type.
//
XilImageType*  cis_outtype = dc->getOutputType();
unsigned int   cis_width      = cis_outtype->getWidth();
unsigned int   cis_height     = cis_outtype->getHeight();
unsigned int   cis_bands      = cis_outtype->getBands();
```

Code Example 6-12 `decompress_Identity.cc` *(3 of 4)*

```
//
//  Get the pointer to the data to decompress...
//
Xil_unsigned32* bp32 =
    (Xil_unsigned32*) dc->getCisBufferManager()->nextFrame();

if(bp32 == NULL) {
// XilCis: No data to decompress
    XIL_CIS_ERROR(XIL_ERROR_SYSTEM, “di-100”, TRUE, dc, FALSE, FALSE);
return XIL_FAILURE;
}

//
//  Just in case we’ve had an error before, we don’t want to
//  SEGV trying to access a word when non-word aligned
//
if((int)bp32 % sizeof(unsigned int)) {
    IDENTITY_BYTESTREAM_ERROR(bp32,frame_type);
}
if(*bp32++ != cis_width) {
    IDENTITY_BYTESTREAM_ERROR(bp32,frame_type);
}
if(*bp32++ != cis_height) {
    IDENTITY_BYTESTREAM_ERROR(bp32,frame_type);
}
if(*bp32++ != cis_bands) {
    IDENTITY_BYTESTREAM_ERROR(bp32,frame_type);
}

Xil_unsigned8* bp = (Xil_unsigned8*) bp32;

//
//  Get the destination image’s origin
//
long x_origin, y_origin;
dst->getOrigin(&x_origin,&y_origin);

//
//  Get the destination image’s memory storage.
//
XilMemoryStorageByte* dst_mem =
    (XilMemoryStorageByte*)dst->getMemoryStorage();
```

Code Example 6-12 `decompress_Identity.cc` *(4 of 4)*

```
    Xil_unsigned8* dst_data        = dst_mem->data;

    Xil_unsigned8* dst_scanline =
        dst_mem->data +
        (y_origin * dst_mem->scanline_stride) +
        (x_origin * dst_mem->pixel_stride);

    Xil_unsigned8* dst_pixel;
    Xil_unsigned8* dst_band;

    for(int i=0; i<cis_height; i++) {
        dst_pixel = dst_scanline;
        for(int j=0; j<cis_width; j++) {
            dst_band = dst_pixel;
            for(int k=0; k<cis_bands; k++) {
                *dst_band++ = *bp++;
            }
            dst_pixel += dst_mem->pixel_stride;
        }
        dst_scanline += dst_mem->scanline_stride;
    }

    dc->getCisBufferManager()->decompressedFrame(bp,frame_type);

    return XIL_SUCCESS;
}
```

Sample Molecule A

This example illustrates a molecule for performing 16-to-8 bit remapping of memory images. It implements the combined atomics `convert16_8(rescale16())`. The source image must be a 1-banded, `XIL_SHORT` image. The destination must be a 1-banded, `XIL_BYTE` image. This example contains a single file, `Rescale16Convert16to8.cc`, which implements the molecule.

Code Example A-1 `Rescale16Convert16yo8.cc` *(1 of 6)*

```
//This line lets emacs recognize this as -*- C++ -*- Code
//----------------------------------------------------------------------
//
//  Description:
//     Contains the convert(rescale()) molecule for 16 bit to 8 bit conversion
//  (Memory to memory 16-to-8 bit remapping)
//
//  Parameters:
//  Source must be a 1-banded, XIL_SHORT image.
//  Destination must be a 1-banded, XIL_BYTE image.
//
//  Returns:
//
//  XIL_SUCCESS or XIL_FAILURE
//
//  Side Effects:
//
//  Notes:
```

Code Example A-1 `Rescale16Convert16yo8.cc` *(2 of 6)*

```
//
//  Deficiencies/ToDo:
//  Should be able to handle multiple bands.
//
//
//---------------------------------------------------------------------
#pragma ident “@(#)Rescale16Convert16to8.cc1.2\t94/03/23 “

#include <xil/XilDefines.h>
#include <xil/XilError.h>
#include <xil/XilImage.h>
#include <xil/XilOp.h>
#include <xil/XilRoi.h>
#include <xil/XilRoiList.h>


//
// Class definition for this molecule
//
class XilDeviceComputeTypeMemory : public XilDeviceComputeType {
 public:
    int Rescale16Convert16to8(XilOp* op, int count);
    ~XilDeviceComputeTypeMemory();

};


//
// Declaration of molecule name and the atomic functions it
// implements for describeMembers routine
//
/* XILCONFIG: Rescale16Convert16to8 = convert16to8(rescale16()) */


//
// define for 16-bit rounding
//
#define ROUND_16(_round16_input_,_round16_output_)              \
{                                       \
    float _round16_tmp_;                                \
                                        \
    if ((_round16_input_) >=0) {                        \
```

Code Example A-1 `Rescale16Convert16yo8.cc` *(3 of 6)*

```
        _round16_tmp_ = (_round16_input_) + 0.5;                \
    }                                   \
    else {                              \
        _round16_tmp_ = (_round16_input_) + -0.5;               \
    }                                   \
    if (_round16_tmp_ >= (float)MAXSHORT) {     \
        (_round16_output_) = (MAXSHORT);                \
    }                                   \
    else if (_round16_tmp_ <= (float)MINSHORT) {        \
        (_round16_output_) = (MINSHORT);                \
    }                                   \
    else {                              \
        (_round16_output_) = ((Xil_signed16) _round16_tmp_);\
    }                                   \
}


//
// the molecule
//
int
XilDeviceComputeTypeMemory::Rescale16Convert16to8(
    XilOp*     op,           // a pointer into the DAG
    int        )             // unused--the number of combined ops to be done
{
    //
    // Get the destination image from the convert16to8 op
    //
    XilImage*  dst = op->getDst();

    //
    // Go to the next operation on the DAG (the rescale16 op)
    // and get the source image and the rescale values
    //
    op = op->getOp1();
    XilImage*  src = op->getSrc1();
    float *scale_value = (float *)(op->getPtrParam(1));
    float *offset_value = (float *)(op->getPtrParam(2));

    //
    // ensure that molecule requirements are met (1 banded images, 16 to 8)
    //
```

Code Example A-1 `Rescale16Convert16yo8.cc` *(4 of 6)*

```
    if((src->getBands() != 1) ||
       (dst->getBands() != 1) ||
       (src->getDataType() != XIL_SHORT) ||
       (dst->getDataType() != XIL_BYTE))
        return XIL_FAILURE;

    //
    // get information about the source
    //
    long src_x_origin, src_y_origin;
    src->getOrigin(&src_x_origin, &src_y_origin);

    //
    // get source’s memory storage
    XilMemoryStorageShort *short_storage;
    short_storage = (XilMemoryStorageShort *)src->getMemoryStorage();
    if (short_storage==NULL) {
    // we could flag an error here, but the core will re-try with
    // atomic operators
        return XIL_FAILURE;
    }
    Xil_signed16 *src_base_addr = (Xil_signed16 *)short_storage->data;
    unsigned long src_next_pixel = short_storage->pixel_stride;
    unsigned long src_next_scan  = short_storage->scanline_stride;

    //
    // get information about the destination
    //
    long dst_x_origin, dst_y_origin;
    dst->getOrigin(&dst_x_origin, &dst_y_origin);

    //
    // get destination’s memory storage
    XilMemoryStorageByte *byte_storage;
    byte_storage = (XilMemoryStorageByte *)dst->getMemoryStorage();
    if (byte_storage==NULL) {
    // we could flag an error here, but the core will re-try with
    // atomic operators
        return XIL_FAILURE;
    }
    Xil_unsigned8 *dst_base_addr = (Xil_unsigned8 *)byte_storage->data;
    unsigned long dst_next_pixel = byte_storage->pixel_stride;
```

Code Example A-1 `Rescale16Convert16yo8.cc` *(5 of 6)*

```
    unsigned long dst_next_scan  = byte_storage->scanline_stride;

    //
    // get the list of intersected ROIs between source and destination
    //
    XilRoi* roi;
    XilRoiList* roi_list= XiliGetRoiList(&roi,dst,src);
    if (roi_list==NULL) {
       // we could flag an error here, but the core will re-try with
       // atomic operators
       return XIL_FAILURE;
    }

    //
    //  Now that we've intersected to determine the pixels that will
    //  be touched in the destination, set the pixelsTouchedRoi on
    //  the image.
    //
    dst->setPixelsTouchedRoi(roi);
    dst->setPixelsTouchedRoi_flag(TRUE);

    //
    // operate on each ROI, all ROI's are guaranteed not to go outside images
    //
    long       x, y;
    unsigned int x_size, y_size;
    float scale = scale_value[0];
    float offset = offset_value[0];
    while (roi_list->next(&x,&y,&x_size,&y_size)) {
        Xil_signed16    *src_scanline = src_base_addr +
                                ((y + src_y_origin) * src_next_scan) +
                                ((x + src_x_origin) * src_next_pixel);
        Xil_signed16    *src_pixel;
        Xil_unsigned8   *dst_scanline = dst_base_addr +
                                ((y + dst_y_origin) * dst_next_scan) +
                                ((x + dst_x_origin) * dst_next_pixel);
        Xil_unsigned8   *dst_pixel;

    //
    // loop over each scanline in the ROI
    //
        do {
```

Code Example A-1 `Rescale16Convert16yo8.cc` *(6 of 6)*

```
            // point to the first pixel of the scanline
            src_pixel = src_scanline;
            dst_pixel = dst_scanline;

        // do the rescale-cast for each pixel in the scanline
            int pixel_count = x_size;
        Xil_signed16 result;
            do {
                float tmp = ((float)(*src_pixel) * scale) + offset;
    ROUND_16(tmp, result);
                *dst_pixel = (Xil_unsigned8) result;
                src_pixel += src_next_pixel;
                dst_pixel += dst_next_pixel;
            } while (--pixel_count);

            // move to next scanline
            src_scanline += src_next_scan;
            dst_scanline += dst_next_scan;

        } while (--y_size);
    }

    // delete the intersected roilist
    // (the roi stored in dest “pixelsTouchedRoi”
    // will be destroyed by the xil core)
    roi_list->destroy();


    // molecule successfully completed
    return XIL_SUCCESS;
}
```

XIL Atomic Functions B

Table B-1 lists the XIL atomic functions. The first column gives the name of the function that must be supplied in the XILCONFIG header comment in order to associate an implemented function with an API call. The second column gives the name of the API binding call associated with the atomic name. Further description of these API functions can be found in the *XIL Reference Manual*.pf

Table B-1 XIL Atomic Functions *(1 of 12)*

Atomic Function	**What It Does**
absolute16	xil_absolute for 16-bit images
add1	xil_add for 1-bit images
add16	xil_add for 16-bit images
add8	xil_add for 8-bit images
addconst1	xil_add_const for 1-bit images
addconst16	xil_add_const for 16-bit images
addconst8	xil_add_const for 8-bit images
affine16bicubic	xil_affine for 16-bit images, bicubic interpolation
affine16bilinear	xil_affine for 16-bit images, bilinear interpolation
affine16general	xil_affine for 16-bit images, general interpolation
affine16nearest	xil_affine for 16-bit images, nearest neighbor interpolation
affine1bicubic	xil_affine for 1-bit images, bicubic interpolation

Table B-1 XIL Atomic Functions *(2 of 12)*

Atomic Function	What It Does
affine1bilinear	xil_affine for 1-bit images, bilinear interpolation
affine1general	xil_affine for 1-bit images, general interpolation
affine1nearest	xil_affine for 1-bit images, nearest neighbor interpolation
affine8bicubic	xil_affine for 8-bit images, bicubic interpolation
affine8bilinear	xil_affine for 8-bit images, bilinear interpolation
affine8general	xil_affine for 8-bit images, general interpolation
affine8nearest	xil_affine for 8-bit images, nearest neighbor interpolation
and1	xil_and for 1-bit images
and16	xil_and for 16-bit images
and8	xil_and for 8-bit images
andconst1	xil_and_const for 1-bit images
andconst16	xil_and_const for 16-bit images
andconst8	xil_and_const for 8-bit images
bandCombine1	xil_band_combine for 1-bit images
bandCombine16	xil_band_combine for 16-bit images
bandCombine8	xil_band_combine for 8-bit images
blackgeneration16	xil_black_generation for 16-bit images
blackgeneration8	xil_black_generation for 8-bit images
blend16a1	xil_blend for 16-bit images with 1-bit alpha
blend16a16	xil_blend for 16-bit images with 16-bit alpha
blend16a8	xil_blend for 16-bit images with 8-bit alpha
blend1a1	xil_blend for 1-bit images with 1-bit alpha
blend1a16	xil_blend for 1-bit images with 16-bit alpha
blend1a8	xil_blend for 1-bit images with 8-bit alpha
blend8a1	xil_blend for 8-bit images with 1-bit alpha
blend8a16	xil_blend for 8-bit images with 16-bit alpha

Table B-1 XIL Atomic Functions *(3 of 12)*

Atomic Function	What It Does
`blend8a8`	`xil_blend` for 8-bit images with 8-bit alpha
`chooseColormap8`	`xil_choose_colormap` for 8-bit images
`colorconvert`	`xil_color_convert`
`convert16_1`	`xil_cast` for 16-bit source, 1-bit destination
`convert16_8`	`xil_cast` for 16-bit source, 8-bit destination
`convert1_16`	`xil_cast` for 1-bit source, 16-bit destination
`convert1_8`	`xil_cast` for 1-bit source, 8-bit destination
`convert8_1`	`xil_cast` for 8-bit source, 1-bit destination
`convert8_16`	`xil_cast` for 8-bit source, 16-bit destination
`convolve1`	`xil_convolve` for 1-bit images
`convolve16`	`xil_convolve` for 16-bit images
`convolve8`	`xil_convolve` for 8-bit images
`copy1`	`xil_copy` for 1-bit images
`copy16`	`xil_copy` for 16-bit images
`copy8`	`xil_copy` for 8-bit images
`copy_with_planemask1`	`xil_copy_with_planemask` for 1-bit images
`copy_with_planemask16`	`xil_copy_with_planemask` for 16-bit images
`copy_with_planemask8`	`xil_copy_with_planemask` for 8-bit images
`copypattern1`	`xil_copy_pattern` for 1-bit images
`copypattern16`	`xil_copy_pattern` for 16-bit images
`copypattern8`	`xil_copy_pattern` for 8-bit images
`dilate1`	`xil_dilate` for 1-bit images
`dilate16`	`xil_dilate` for 16-bit images
`dilate8`	`xil_dilate` for 8-bit images
`divide1`	`xil_divide` for 1-bit images
`divide16`	`xil_divide` for 16-bit images
`divide8`	`xil_divide` for 8-bit images

Table B-1 XIL Atomic Functions *(4 of 12)*

Atomic Function	What It Does
divideintoconst1	xil_divide_into_const for 1-bit images
divideintoconst16	xil_divide_into_const for 16-bit images
divideintoconst8	xil_divide_into_const for 8-bit images
edge_detect_sobel1	xil_edge_detection for 1-bit images
edge_detect_sobel16	xil_edge_detection for 16-bit images
edge_detect_sobel8	xil_edge_detection for 8-bit images
erode1	xil_erode for 1-bit images
erode16	xil_erode for 16-bit images
erode8	xil_erode for 8-bit images
errordiffusion16_1	xil_error_diffusion for 16-bit source, 1-bit destination
errordiffusion16_16	xil_error_diffusion for 16-bit source, 16-bit destination
errordiffusion16_8	xil_error_diffusion for 16-bit source, 8-bit destination
errordiffusion1_1	xil_error_diffusion for 1-bit source, 1-bit destination
errordiffusion1_16	xil_error_diffusion for 1-bit source, 16-bit destination
errordiffusion1_8	xil_error_diffusion for 1-bit source, 8-bit destination
errordiffusion8_1	xil_error_diffusion for 8-bit source, 1-bit destination
errordiffusion8_16	xil_error_diffusion for 8-bit source, 16-bit destination
errordiffusion8_8	xil_error_diffusion for 8-bit source, 8-bit destination
extrema1	xil_extrema for 1-bit images
extrema16	xil_extrema for 16-bit images
extrema8	xil_extrema for 8-bit images

Table B-1 XIL Atomic Functions *(5 of 12)*

Atomic Function	What It Does
fill1	xil_fill for 1-bit images
fill16	xil_fill for 16-bit images
fill8	xil_fill for 8-bit images
histogram1	xil_histogram for 1-bit images
histogram16	xil_histogram for 16-bit images
histogram8	xil_histogram for 8-bit images
lookup16_1	xil_lookup for 16-bit source, 1-bit destination
lookup16_16	xil_lookup for 16-bit source, 16-bit destination
lookup16_8	xil_lookup for 16-bit source, 8-bit destination
lookup1_1	xil_lookup for 1-bit source, 1-bit destination
lookup1_16	xil_lookup for 1-bit source, 16-bit destination
lookup1_8	xil_lookup for 1-bit source, 8-bit destination
lookup8_1	xil_lookup for 8-bit source, 1-bit destination
lookup8_16	xil_lookup for 8-bit source, 16-bit destination
lookup8_8	xil_lookup for 8-bit source, 8-bit destination
max16	xil_max for 16-bit images
max8	xil_max for 8-bit images
min16	xil_min for 16-bit images
min8	xil_min for 8-bit images
multiply1	xil_multiply for 1-bit images
multiply16	xil_multiply for 16-bit images
multiply8	xil_multiply for 8-bit images
multiplyconst1	xil_multiply_const for 1-bit images
multiplyconst16	xil_multiply_const for 16-bit images
multiplyconst8	xil_multiply_const for 8-bit images
nearestcolor16_1	xil_nearest_color for 16-bit source, 1-bit destination

Table B-1 XIL Atomic Functions *(6 of 12)*

Atomic Function	What It Does
nearestcolor16_16	xil_nearest_color for 16-bit source, 16-bit destination
nearestcolor16_8	xil_nearest_color for 16-bit source, 8-bit destination
nearestcolor1_1	xil_nearest_color for 1-bit source, 1-bit destination
nearestcolor1_16	xil_nearest_color for 1-bit source, 16-bit destination
nearestcolor1_8	xil_nearest_color for 1-bit source, 8-bit destination
nearestcolor8_1	xil_nearest_color for 8-bit source, 1-bit destination
nearestcolor8_16	xil_nearest_color for 8-bit source, 16-bit destination
nearestcolor8_8	xil_nearest_color for 8-bit source, 8-bit destination
not1	xil_not for 1-bit images
not16	xil_not for 16-bit images
not8	xil_not for 8-bit images
or1	xil_or for 1-bit images
or16	xil_or for 16-bit images
or8	xil_or for 8-bit images
orconst1	xil_or_const for 1-bit images
orconst16	xil_or_const for 16-bit images
orconst8	xil_or_const for 8-bit images
ordereddither16_1	xil_ordered_dither for 16-bit source, 1-bit destination
ordereddither16_16	xil_ordered_dither for 16-bit source, 16-bit destination
ordereddither16_8	xil_ordered_dither for 16-bit source, 8-bit destination

Table B-1 XIL Atomic Functions *(7 of 12)*

Atomic Function	**What It Does**
`ordereddither1_1`	`xil_ordered_dither` for 1-bit source, 1-bit destination
`ordereddither1_16`	`xil_ordered_dither` for 1-bit source, 16-bit destination
`ordereddither1_8`	`xil_ordered_dither` for 1-bit source, 8-bit destination
`ordereddither8_1`	`xil_ordered_dither` for 8-bit source, 1-bit destination
`ordereddither8_16`	`xil_ordered_dither` for 8-bit source, 16-bit destination
`ordereddither8_8`	`xil_ordered_dither` for 8-bit source, 8-bit destination
`paint1`	`xil_paint` for 1-bit images
`paint16`	`xil_paint` for 16-bit images
`paint8`	`xil_paint` for 8-bit images
`rescale1`	`xil_rescale` for 1-bit images
`rescale16`	`xil_rescale` for 16-bit images
`rescale8`	`xil_rescale` for 8-bit images
`rotate16bicubic`	`xil_rotate` for 16-bit images, bicubic interpolation
`rotate16bilinear`	`xil_rotate` for 16-bit images, bilinear interpolation
`rotate16general`	`xil_rotate` for 16-bit images, general interpolation
`rotate16nearest`	`xil_rotate` for 16-bit images, nearest neighbor interpolation
`rotate1bicubic`	`xil_rotate` for 1-bit images, bicubic interpolation
`rotate1bilinear`	`xil_rotate` for 1-bit images, bilinear interpolation
`rotate1general`	`xil_rotate` for 1-bit images, general interpolation
`rotate1nearest`	`xil_rotate` for 1-bit images, nearest neighbor interpolation
`rotate8bicubic`	`xil_rotate` for 8-bit images, bicubic interpolation
`rotate8bilinear`	`xil_rotate` for 8-bit images, bilinear interpolation

Table B-1 XIL Atomic Functions *(8 of 12)*

Atomic Function	What It Does
rotate8general	xil_rotate for 8-bit images, general interpolation
rotate8nearest	xil_rotate for 8-bit images, nearest neighbor interpolation
scale16bicubic	xil_scale for 16-bit images, bicubic interpolation
scale16bilinear	xil_scale for 16-bit images, bilinear interpolation
scale16general	xil_scale for 16-bit images, general interpolation
scale16nearest	xil_scale for 16-bit images, nearest neighbor interpolation
scale1bicubic	xil_scale for 1-bit images, bicubic interpolation
scale1bilinear	xil_scale for 1-bit images, bilinear interpolation
scale1general	xil_scale for 1-bit images, general interpolation
scale1nearest	xil_scale for 1-bit images, nearest neighbor interpolation
scale8bicubic	xil_scale for 8-bit images, bicubic interpolation
scale8bilinear	xil_scale for 8-bit images, bilinear interpolation
scale8general	xil_scale for 8-bit images, general interpolation
scale8nearest	xil_scale for 8-bit images, nearest neighbor interpolation
setvalue1	xil_set_value for 1-bit images
setvalue16	xil_set_value for 16-bit images
setvalue8	xil_set_value for 8-bit images
softfill1	xil_soft_fill for 1-bit images
softfill16	xil_soft_fill for 16-bit images
softfill8	xil_soft_fill for 8-bit images
squeezerange1	xil_squeeze_range for 1-bit images
squeezerange16	xil_squeeze_range for 16-bit images
squeezerange8	xil_squeeze_range for 8-bit images
subsample1_8	xil_subsample_binary_to_gray 8-bit destination
subsampleAdaptive1	xil_subsample_adaptive for 1-bit images

Table B-1 XIL Atomic Functions *(9 of 12)*

Atomic Function	What It Does
subsampleAdaptive16	xil_subsample_adaptive for 16-bit images
subsampleAdaptive8	xil_subsample_adaptive for 8-bit images
subtract1	xil_subtract for 1-bit images
subtract16	xil_subtract for 16-bit images
subtract8	xil_subtract for 8-bit images
subtractfromconst1	xil_subtract_from_const for 1-bit images
subtractfromconst16	xil_subtract_from_const for 16-bit images
subtractfromconst8	xil_subtract_from_const for 8-bit images
tablewarp16bicubic	xil_tablewarp for 16-bit images, bicubic interpolation
tablewarp16bilinear	xil_tablewarp for 16-bit images, bilinear interpolation
tablewarp16general	xil_tablewarp for 16-bit images, general interpolation
tablewarp16nearest	xil_tablewarp for 16-bit images, nearest neighbor interpolation
tablewarp1bicubic	xil_tablewarp for 1-bit images, bicubic interpolation
tablewarp1bilinear	xil_tablewarp for 1-bit images, bilinear interpolation
tablewarp1general	xil_tablewarp for 1-bit images, general interpolation
tablewarp1nearest	xil_tablewarp for 1-bit images, nearest neighbor interpolation
tablewarp8bicubic	xil_tablewarp for 8-bit images, bicubic interpolation
tablewarp8bilinear	xil_tablewarp for 8-bit images, bilinear interpolation
tablewarp8general	xil_tablewarp for 8-bit images, general interpolation

Table B-1 XIL Atomic Functions *(10 of 12)*

Atomic Function	What It Does
tablewarp8nearest	xil_tablewarp for 8-bit images, nearest neighbor interpolation
tablewarph16bicubic	xil_tablewarp_horizontal for 16-bit images, bicubic interpolation
tablewarph16bilinear	xil_tablewarp_horizontal for 16-bit images, bilinear interpolation
tablewarph16general	xil_tablewarp_horizontal for 16-bit images, general interpolation
tablewarph16nearest	xil_tablewarp_horizontal for 16-bit images, nearest neighbor interpolation
tablewarph1bicubic	xil_tablewarp_horizontal for 1-bit images, bicubic interpolation
tablewarph1bilinear	xil_tablewarp_horizontal for 1-bit images, bilinear interpolation
tablewarph1general	xil_tablewarp_horizontal for 1-bit images, general interpolation
tablewarph1nearest	xil_tablewarp_horizontal for 1-bit images, nearest neighbor interpolation
tablewarph8bicubic	xil_tablewarp_horizontal for 8-bit images, bicubic interpolation
tablewarph8bilinear	xil_tablewarp_horizontal for 8-bit images, bilinear interpolation
tablewarph8general	xil_tablewarp_horizontal for 8-bit images, general interpolation
tablewarph8nearest	xil_tablewarp_horizontal for 8-bit images, nearest neighbor interpolation
tablewarpv16bicubic	xil_tablewarp_vertical for 16-bit images, bicubic interpolation
tablewarpv16bilinear	xil_tablewarp_vertical for 16-bit images, bilinear interpolation
tablewarpv16general	xil_tablewarp_vertical for 16-bit images, general interpolation

Table B-1 XIL Atomic Functions *(11 of 12)*

Atomic Function	What It Does
tablewarpv16nearest	xil_tablewarp_vertical for 16-bit images, nearest neighbor interpolation
tablewarpv1bicubic	xil_tablewarp_vertical for 1-bit images, bicubic interpolation
tablewarpv1bilinear	xil_tablewarp_vertical for 1-bit images, bilinear interpolation
tablewarpv1general	xil_tablewarp_vertical for 1-bit images, general interpolation
tablewarpv1nearest	xil_tablewarp_vertical for 1-bit images, nearest neighbor interpolation
tablewarpv8bicubic	xil_tablewarp_vertical for 8-bit images, bicubic interpolation
tablewarpv8bilinear	xil_tablewarp_vertical for 8-bit images, bilinear interpolation
tablewarpv8general	xil_tablewarp_vertical for 8-bit images, general interpolation
tablewarpv8nearest	xil_tablewarp_vertical for 8-bit images, nearest neighbor interpolation
threshold1	xil_threshold for 1-bit images
threshold16	xil_threshold for 16-bit images
threshold8	xil_threshold for 8-bit images
translate16bicubic	xil_translate for 16-bit images, bicubic interpolation
translate16bilinear	xil_translate for 16-bit images, bilinear interpolation
translate16general	xil_translate for 16-bit images, general interpolation
translate16nearest	xil_translate for 16-bit images, nearest neighbor interpolation
translate1bicubic	xil_translate for 1-bit images, bicubic interpolation
translate1bilinear	xil_translate for 1-bit images, bilinear interpolation

Table B-1 XIL Atomic Functions *(12 of 12)*

Atomic Function	What It Does
translate1general	xil_translate for 1-bit images, general interpolation
translate1nearest	xil_translate for 1-bit images, nearest neighbor interpolation
translate8bicubic	xil_translate for 8-bit images, bicubic interpolation
translate8bilinear	xil_translate for 8-bit images, bilinear interpolation
translate8general	xil_translate for 8-bit images, general interpolation
translate8nearest	xil_translate for 8-bit images, nearest neighbor interpolation
transpose1	xil_transpose for 1-bit images
transpose16	xil_transpose for 16-bit images
transpose8	xil_transpose for 8-bit images
xor1	xil_xor for 1-bit images
xor16	xil_xor for 16-bit images
xor8	xil_xor for 8-bit images
xorconst1	xil_xor_const for 1-bit images
xorconst16	xil_xor_const for 16-bit images
xorconst8	xil_xor_const for 8-bit images

`XilOp` *Object* C

This appendix lists the number of image sources supported by an XIL function and the `XilOp` member functions that you must use to extract the image sources and to extract an XIL function's parameters from the `XilOp` object. You must know this information anytime you implement XIL atomic functions, such as when you write a compute device handler. For more information about the `XilOp` class, see Chapter 1, "Overview." For more information about compute devices, see Chapter 4, "Compute Devices."

XIL Function	`XilOp` Member Function	Parameter
`absolute`	`getSrc1`	src1
`add`	`getSrc1` `getSrc2`	src1 src2
`addconst`	`getSrc1` `getPtrParam(1)`	src1 const_array

`affine`	`getSrc1`
	`getPtrParam(1)`
	`getObjParam(3)`
	`getObjParam(4)`
		src1
		matrix
		horiz_inter_tbl
		vert_inter_tbl

`and`	`getSrc1` `getSrc2`	src1 src2
`andconst`	`getSrc1` `getPtrParam(1)`	src1 const_array

XIL Function	XilOp Member Function	Parameter
blackgeneration	getSrc1 getFloatParam(1) getFloatParam(2)	src1 black undercolor
bandCombine	getSrc1 getObjParam(1)	src1 matrix
blend	getSrc1 getSrc2 getSrc3	src1 src2 scr3
chooseColormap	getPtrParam(1) getLongParam(2)	lut_p size
colorconvert	getSrc1 getObjParam(1) getObjParam(2)	src1 src_colorspace dst_colorspace
compress	getSrc1 getLongParam(1)	src1 write_frame
convert	getSrc1	src1
convolve	getSrc1 getObjParam(1) getLongParam(2)	src1 kernel edge_condition
copy	getSrc1	src1
copypattern	getSrc1	src1
copy_with_planemask	getSrc1 getPtrParam(1)	src1 const_array
decompress	getLongParam(1)	read_frame
dilate	getSrc1 getObjParam(1)	src1 sel
divide	getSrc1 getSrc2	src1 src2
divideintoconst	getSrc1 getPtrParam(1)	src1 const_array
edge_detection_sobel	getSrc1	src1
erode	getSrc1 getObjParam(1)	src1 sel

XIL Function	XilOp Member Function	Parameter
errorDiffusion	getSrc1 getObjParam(1) getObjParam(2)	src1 colormap distribution
extrema	getSrc1 getPtrParam(1) getPtrParam(2)	src1 max min
fill	getSrc1 getFloatParam(1) getFloatParam(2) getPtrParam(3) getPtrParam(4)	src1 xseed yseed boundary fill
histogram	getSrc1 getObjParam(1) getLongParam(2) getLongParam(3)	src1 histogram skip_x skip_y
lookup	getSrc1 getObjParam(1)	src1 lut
max	getSrc1 getSrc2	src1 src2
min	getSrc1 getSrc2	src1 src2
multiply	getSrc1 getSrc2	src1 src2
multiplyconst	getSrc1 getPtrParam(1)	src1 const_array
nearestcolor	getSrc1 getObjParam(1)	src1 cmap
not	getSrc1	src1
or	getSrc1 getSrc2	src1 src2
orconst	getSrc1 getPtrParam(1)	src1 const_array
ordereddither	getSrc1 getObjParam(1) getObjParam(2)	src1 colormap dithermask

XIL Function	XilOp **Member Function**	Parameter
paint	getSrc1 getPtrParam(1) getObjParam(2) getLongParam(3) getPtrParam(4)	src1 color brush count points
rescale	getSrc1 getPtrParam(1) getPtrParam(2)	src1 scale_array offset_array
rotate	getSrc1 getFloatParam(1) getObjParam(3) getObjParam(4)	src1 angle horiz vertical
scale	getSrc1 getFloatParam(1) getFloatParam(2) getObjParam(3) getObjParam(4)	src1 xfactor yfactor horiz vertical
setvalue	getPtrParam(1)	const_array
softfill	getSrc1 getFloatParam(1) getFloatParam(2) getPtrParam(3) getLongParam(4) getPtrParam(5) getPtrParam(6)	src1 xseed yseed fg num_bgcolor bg fill
squeezerange	getPtrParam(1)	lut_p
subsample1_8	getSrc1 getFloatParam(1) getFloatParam(2)	src1 xfactor yfactor
subsampleAdaptive	getSrc1 getFloatParam(1) getFloatParam(2)	src1 xfactor yfactor
subtract	getSrc1 getSrc2	src1 src2
subtractfromconst	getSrc1 getPtrParam(1)	src1 const_array

XIL Function	XilOp Member Function	Parameter
tablewarp	getSrc1 getSrc2 getObjParam(3) getObjParam(4)	src1 warp_table image horiz table vert table
tablewarph	getSrc1 getSrc2 getObjParam(3) getObjParam(4)	src1 warp_table image horiz table vert table
tablewarpv	getSrc1 getSrc2 getObjParam(3) getObjParam(4)	src1 warp_table image horiz table vert table
threshold	getSrc1 getPtrParam(1) getPtrParam(2) getPtrParam(3)	src1 low high map
translate	getSrc1 getFloatParam(1) getFloatParam(2) getObjParam(3) getObjParam(4)	src1 xoff yoff horiz vertical
transpose	getSrc1 getLongParam(1)	src1 fliptype
xor	getSrc1 getSrc2	src1 src2
xorconst	getSrc1 getPtrParam(1)	src1 const_array

Index

D

H

I

K

L

M

N