Compressed Image Sequences 13

An XIL compressed image sequence (CIS) is a container for compressed images. These compressed images may represent a series of still images, frames from a movie, or pages from a document, and they may have been compressed in one of several formats:

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

For more information about these compression formats—what a particular type of compression is useful for, how a certain type of compressor is implemented, and so on—see chapters 13 through 19.

This chapter has three main sections. The first section discusses basic CIS operations, such as creating a CIS and writing data to it. The second explains what attributes CISs have and how you set and read the values of those attributes. The third talks about recovering from errors in a CIS's datastream.

Basic CIS Management

This section explains several basic CIS-related tasks:

- Creating and destroying a CIS
- Putting compressed data in a CIS
- Reading data from a CIS

13

If you have looked at the example programs presented in Chapter 12, "Compressing and Decompressing Sequences of Images," you already know something about performing these tasks. Don't skip the following sections, however, because they introduce a number of functions that were not used in the examples.

Creating and Destroying a CIS

You create a CIS by calling the function `xil_cis_create()`. This function takes two arguments: a handle to the system-state data structure that was created when you initialized the XIL library and a string that identifies the compressor/decompressor that will be used to write compressed data to, or read compressed data from, the CIS. For example, the call to create a CIS might look like this.

```
XilCis cis;
XilSystemState state;

cis = xil_cis_create(state, "Jpeg");
```

The string you pass to the function—`Jpeg` in the code fragment above—can be any one of the following strings:

- `Cell` (a Cell codec)
- `CellB` (a CellB codec)
- `Jpeg` (a JPEG baseline sequential codec)
- `JpegLL` (a JPEG lossless codec)
- `H261` (an H.261 decompressor)
- `Mpeg1` (an MPEG-1 decompressor)
- `faxG3` (a CCITT Group 3 codec)
- `faxG4` (a CCITT Group 4 codec)

The return value of `xil_cis_create()` is a handle to the newly created CIS. You use this handle as an argument to all subsequent functions that affect the CIS.

A function related to `xil_cis_create()` is `xil_cis_reset()`. The latter function takes an existing CIS, which may contain compressed data, and returns it to its initial state. That is, it clears out all existing compressed data

and frees any memory the CIS has allocated to hold that data. The only argument to this function is a handle to the CIS whose compressed data you want to clear.

To destroy a CIS, you call the function `xil_cis_destroy()`. This function deallocates any buffers allocated to hold compressed data and frees the memory used to hold other information about the CIS. The function's only argument is a handle to the CIS.

Putting Compressed Data into a CIS

There are two primary ways to get compressed data into a CIS. One way is to call the function `xil_compress()` to compress one or more XIL images and write the compressed data to the CIS. The second way is to move an existing compressed data stream from system memory to a CIS. This compressed data may have been compressed originally by an XIL or a non-XIL compressor.

Both of these methods change the value of an internal CIS index so that the next operation that writes compressed data to the CIS will append its data to existing data in the CIS. For more information about this index, see the section "Start Frame, Read Frame, Write Frame" on page 260.

Using `xil_compress()`

The function `xil_compress()` reads an XIL image (or possibly more than one image), compresses it, and writes the compressed data to a CIS. The function takes two arguments. The first argument is a handle to a source image, and the second is a handle to a CIS. You do not have to specify a compressor or codec because each CIS is associated with a single compressor/decompressor when the CIS is created.

If the image handle passed to `xil_compress()` is a handle to a memory image or a display image, the function compresses a single image at a time. If the handle is a handle to a device image such as a video-capture card, `xil_compress()` may compress one or a number of frames per invocation.

Note – This description of what `xil_compress()` does is somewhat oversimplified. Some compressors may prefer to compress groups of frames. Therefore, they read an image each time your application calls `xil_compress()`, but they may not actually compress the images and write

data to the CIS until they have read *n* frames. If you need to make sure that a compressor has compressed all the frames it has read so far, call the function `xil_cis_flush()`.

Each compressor has certain requirements regarding the type of XIL image you can compress using `xil_compress()`. These requirements are summarized in Table 13-1. An N/A in a table box indicates that there is no requirement in a particular area.

Table 13-1 Types of Images Supported by XIL Compressors

	Width and Height	**Number of Bands**	**Data Type**	**Color Space**
Cell	Multiple of 4	3	`XIL_BYTE`	`rgb709`, `ycc601`, `ycc709`
CellB	Multiple of 4	3	`XIL_BYTE`	`ycc601`
JPEG baseline sequential	N/A	1 to 255	`XIL_BYTE`	N/A
JPEG lossless	N/A	1 to 255	`XIL_BYTE` `XIL_SHORT`	N/A
H.261	176 x 144 (QCIF) or 352 x 288 (CIF)	3	`XIL_BYTE`	`ycc601`
MPEG-1	N/A	3	`XIL_BYTE`	`ycc601`
CCITT Group 3	N/A	1	`XIL_BIT`	N/A
CCITT Group 4	N/A	1	`XIL_BIT`	N/A

You can also obtain information about the type of XIL image that can be written to a particular CIS using the function `xil_cis_get_input_type()`. See “Input and Output Image Type” on page 259 for more information about this function.

Putting a Compressed Datastream into a CIS

If you have an existing stream of compressed data in system memory, you can put that data into a CIS buffer in one of two ways. In both cases, the compressed data must have been compressed with a compressor of the same type as the compressor/decompressor associated with the CIS.

First, you can copy the datastream into the CIS buffer using the function `xil_cis_put_bits()`, whose function prototype is shown below.

```
void xil_cis_put_bits(XilCis cis, int nbytes, int nframes,
    void *data);
```

This routine copies `nbytes` of compressed data representing `nframes` of uncompressed data into the CIS `cis`. The argument `data` is a generic pointer to the data to be copied into the CIS.

The argument `nframes` normally indicates the number of frames represented by `nbytes` of compressed data. If you do not know the exact number of frames in the datastream, but know that there are no partial frames (frames that are not completely represented in the datastream), you should set `nframes` to -1. This value informs the CIS that the data being placed into it contains one or more complete frames and no partial frames. If you know that the first or last frame is not represented completely by the data in the stream, or you're not sure whether the datastream contains a partial frame, you should set `nframes` to 0. This value alerts the compressor that the datastream may contain a partial frame.

Note – Don't set `nframes` to 0 if you know that the datastream contains only complete frames. Doing so will slow down the copy.

You can also put a compressed datastream into a CIS using the function `xil_cis_put_bits_ptr()`. Its function prototype is shown below:

```
void xil_cis_put_bits_ptr(XilCis cis, int nbytes, int nframes,
    void *data, XIL_FUNCPTR_DONE_WITH_DATA done_with_data);
```

This function does not copy data into the CIS buffer, but gives the CIS a pointer to the datastream. If you use this function, your application is responsible for ensuring that the datastream remains valid.

You must also free the memory where the datastream is stored. One way to do this is to pass `xil_cis_put_bits_ptr()` a pointer to a function (*done_with_data*) that frees the memory. The `xil_cis_put_bits_ptr()` routine will call this function if the CIS is destroyed, is reset, or no longer needs the data. You should define this *done_with_data* routine to return `void` and to take as its only argument a generic pointer to `data`. Thus, the prototype for the function should look like this:

```
void free_memory(void *data)
```

When you call `xil_cis_put_bits_ptr()`, put the function's name in the spot where *done_with_data* is shown in the function prototype.

```
xil_cis_put_bits_ptr(cis, nbytes, nframes, data, free_memory);
```

If you don't pass `xil_cis_put_bits_ptr()` a pointer to a function, pass `NULL` as the final argument. In this case, your program must determine when the memory holding the datastream is no longer needed.

The value of `nframes` has the same meanings when you're using `xil_cis_put_bits_ptr()` as it does when you're using `xil_cis_put_bits()`.

Reading Data from a CIS

As with putting data into a CIS, there are two basic ways to get data out of a CIS. First, you can use the function `xil_decompress()` to read one frame's worth of compressed data from the CIS, decompress the data, and write the decompressed data to an XIL image. Second, you can get a pointer to the compressed data in the CIS and write the data to a file using a function like `fwrite()`.

Both of these methods change the value of an internal CIS index so that the next operation that reads compressed data from the CIS will begin reading with the first frame in the CIS following the last frame read during the just-completed operation. For more information about this index, see the section "Start Frame, Read Frame, Write Frame" on page 260.

Using `xil_decompress()`

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

```
void xil_decompress(XilCis cis, XilImage dst);
```

The first argument to the function is a handle to your CIS, and the second is a handle to an XIL image into which a frame of video can be decompressed. The decompressor used for the operation will be the one you specified when you created your CIS.

The XIL image into which you decompress a frame of data must have the same width, height, number of bands, and data type as the images stored in the CIS. If you do not know these attributes of the images in the CIS, you can determine them using the function `xil_cis_get_output_type()`. See the section "Input and Output Image Type" on page 259 for more information on this function.

Copying Compressed Data from a CIS

If you want to copy data from a CIS without decompressing it, you should follow this procedure:

1. Use the function `xil_cis_get_bits_ptr()` to get a generic pointer to the compressed data in the CIS.
2. Use a function like `fwrite()` to copy the data from the CIS buffer to a file.

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

```
void *xil_cis_get_bits_ptr(XilCis cis, int *nbytes,
    int *nframes);
```

The first argument to this function is a handle to the CIS. The second argument is the address of a variable in which the function stores the number of bytes to which its return value (a pointer to `void`) points. The third argument is the address of a variable in which the function stores the number of frames represented by `nbtyes`.

Note that `nbytes` is frequently not the total number of bytes of compressed data stored in the CIS. This is true because the CIS's "buffer" may actually be a list of buffers. As a result, you may have to call `xil_cis_get_bits_ptr()` multiple times to read all the compressed data in a CIS. See the next section for information about controlling loops in which you're reading data from a CIS.

Reading Data from a CIS in a Loop

If you want to decompress all the frames in a CIS or copy all the compressed data in a CIS to a file, you need to decompress data or copy data repeatedly, until all the data in the CIS has been read. A good way to control the loop that decompresses or copies data is to use the function `xil_cis_has_frame()`, as shown below.

```
XilCis cis;
XilImage dst;

while (xil_cis_has_frame(cis) == TRUE)
    xil_decompress(cis, dst);
```

The function `xil_cis_has_frame()` returns `TRUE` as long as one complete frame of compressed data remains in the CIS. Its single argument is a handle to your CIS.

Note – A loop like the one shown above works because a call to `xil_decompress()` or `xil_cis_get_bits_ptr()` changes an attribute of the CIS called its *read frame*. (For complete information about this attribute, see the section "Start Frame, Read Frame, Write Frame" on page 260.) If you decompress a frame, the read frame is incremented by 1, and if you copy 50 frames, the read frame is increased by 50.

You can also control this type of loop using the functions `xil_cis_has_data()` and `xil_cis_number_of_frames()`. The first of these functions returns the number of unread bytes in the CIS, and the second returns the number of unread frames. Neither function, however, is as effective as `xil_cis_has_frame()` for normal loop control: `xil_cis_has_data()` can return a nonzero value even after the last complete frame has been read (if the CIS contains a partial frame), and `xil_cis_number_of_frames()` may take longer to execute than `xil_cis_has_frame()`.

Nonsequential Reads

Normally, when you read data from a CIS, you begin reading at the last frame read plus 1. However, you can also begin reading at a different point. To do this, you call the routine `xil_cis_seek()` to indicate the frame you would like to read next. The function prototype for this routine is shown below.

```
void xil_cis_seek(XilCis cis, int framenumber, int relative_to);
```

The first argument to this function is a handle to your CIS. The second and third arguments, taken together, determine which frame will become the CIS's read frame (the frame that will be read first the next time you read data from the CIS). The second argument, `framenumber`, is a number of frames and is to be construed as an offset from one of three locations. This location is specified by the value of `relative_to`. A value of 0 indicates that `framenumber` is an offset from the beginning of the CIS (frame 0), a value of 1 that `framenumber` is an offset from the current read frame, and a value of 2 that `framenumber` is an offset from the last frame in the CIS.

It is an error to seek for a frame prior to the first frame currently in the CIS or for a frame beyond the end of the CIS. See the section "Start Frame, Read Frame, Write Frame" on page 260 for information about determining the frame numbers of the first and last frames in the CIS. It is also an error to seek a frame prior to the current read frame if the CIS's random-access attribute is set to 0. See the section "Random Access Flag" on page 260 for a discussion of this attribute.

General CIS Attributes

Up to this point, this chapter has presented the CIS as a buffer or database in which you store compressed data. However, you can also think of a CIS as a structure with members that contain information about the compressor/decompressor associated with the CIS, the type of the images in a CIS, and so on. These members are referred to as *attributes* of the CIS. The following list shows you the general CIS attributes:

Note – The term *general attribute* is used here because CISs also have codec-specific attributes. These are covered in chapters 13 to 19, where the different XIL compressors/decompressors are discussed.

- Compressor (or decompressor) associated with the CIS
- Type of the compressor
- Type of image that can be compressed and stored in the CIS
- Type of image that will result when compressed data in the CIS is decompressed
- Flag indicating whether the data in the CIS can be accessed randomly
- Index to the first image currently in the CIS
- Index to the image that will be read next
- Index to the image that will be written next
- Maximum number of images the CIS can contain
- Number of already-read images the CIS should try to keep in its buffer
- Flag indicating whether the compressor should automatically recover from an error it knows how to handle
- Name of the CIS

These attributes are discussed in detail in the following sections.

Compressor and Compression Type

A CIS's compressor attribute is a string that identifies the compressor/decompressor that will be used to compress or decompress data for the CIS. This attribute is set when you create the CIS. For example, if you create your CIS using the statement

```
cis = xil_cis_create(state, "Cell");
```

the attribute will be set to `Cell`. You cannot change the value of this attribute after creating the CIS; however, you can read its value using the function `xil_cis_get_compressor()`.

The compression-type attribute is a string that identifies a compressor's class. In this release of the XIL library, each class contains only one compressor/decompressor. However, as Sun and third parties write new XIL compressors/decompressors, this situation will change. For instance, if a third party writes a JPEG baseline sequential compressor, that compressor will have a unique compressor name, but a compression type of `JPEG`, which is the compression type of the XIL JPEG baseline sequential codec. Like the

compressor attribute, this attribute is set when you create your CIS and cannot be changed afterwards. You can retrieve the value, however, using the function `xil_cis_get_compression_type()`.

Table 13-2 shows the currently available compressors/decompressors and their types.

Table 13-2 Compressors and Compressor Types

Compressor	Compression Type
`Jpeg`	`JPEG`
`JpegLL`	`JPEGLL`
`Cell`	`CELL`
`CellB`	`CELLB`
`H261`	`H261`
`Mpeg1`	`MPEG1`
`faxG3`	`FAXG3`
`faxG4`	`FAXG4`

Input and Output Image Type

A CIS's input-image-type attribute is a data structure of type `XilImageType` that defines the type of XIL image that may be compressed and written to the CIS. This structure contains information about an image's width, height, number of bands, and data type.

This information is available as soon as the CIS is created and may be retrieved with the function `xil_cis_get_input_type()`. For example, assume you use the following statement to create a CIS:

```
cis = xil_cis_create(state, "Cell");
```

This statement creates a CIS that will be written to by the Cell compressor. If you then call `xil_cis_get_input_type()` to determine the type of image that can be compressed and written to this CIS, you will find that the image must have three bands and contain `XIL_BYTE` data. The Cell compressor can only work with images that have these characteristics. At this point, the values of the image type's width and height members will be 0, which indicates that there are currently no specific requirements for these characteristics.

After you have used `xil_compress()` to write compressed data to your CIS, additional information about the input image type will become available. For instance, a call to `xil_cis_get_input_type()` will return nonzero values for the image type's width and height. These will correspond to the width and height of the images that have already been compressed.

A CIS's output-data-type attribute is also a structure of type `XilImageType`, but indicates the type of image that will be produced when data stored in the CIS is decompressed. You get this image-type structure by calling `xil_cis_get_output_type()`.

It's an error to call this function before writing data to your CIS. However, once the CIS contains data, `xil_cis_get_output_type()` will return information about the width, height, number of bands, and data type of the images stored in the CIS. In general, programs that decompress data must call this function because `xil_decompress()` expects its second argument to be an XIL image to which it can write its output. Before you can create this XIL image, you must know the output image type.

Random Access Flag

You can always seek forward in a CIS using the function `xil_cis_seek()`. However, only certain compressors/decompressors allow you to seek backwards. To determine whether your compressor/decompressor allows this type of seek, call the function `xil_cis_get_random_access()` to determine the value of the CIS's random-access attribute. A return value of 1 indicates that backward seeks are supported, and a return value of 0 indicates that such seeks are not allowed.

Start Frame, Read Frame, Write Frame

These attributes are integer indexes to certain important images or frames stored in a CIS.

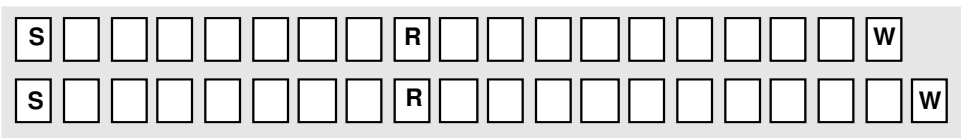
The start-frame attribute is an index to the first frame in the CIS that is currently accessible. For now, just think of the start frame as the first frame written to the CIS (frame 0). To obtain the number of the current start frame, you call the function `xil_cis_get_start_frame()`.

The read-frame attribute is an index to the frame that will be read the next time you read data from the CIS. If you use `xil_decompress()` to decompress a frame, the function decompresses the current read frame and then increments the read frame by 1. If you get a pointer to data you want to read using the function `xil_cis_get_bits_ptr()`, the read frame is incremented by the number of frames that you read from the CIS. You can determine the current read frame using the function `xil_cis_get_read_frame()`.

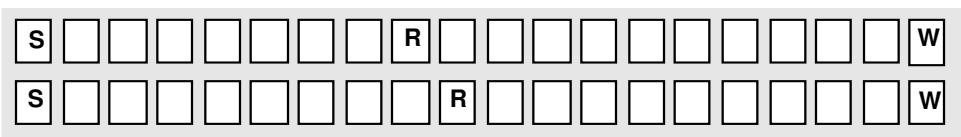
The write-frame attribute is an index to the frame that will be written to the next time you put data into the CIS. (Thus, the index to the last frame currently in the CIS is the write-frame index minus 1.) When you add data to the CIS using `xil_compress()`, `xil_cis_put_bits()`, or `xil_cis_put_bits_ptr()`, this write-frame index is incremented by the number of frames added to the CIS. To read the current value of the write-frame attribute, use the function `xil_cis_get_write_frame()`.

The diagrams below may shed further light on how these indexes relate to one another. In each diagram, each row of rectangles represents a CIS buffer, and each rectangle in the row represents a frame of compressed data. Start frames, read frames, and write frames are labeled S, R, and W respectively.

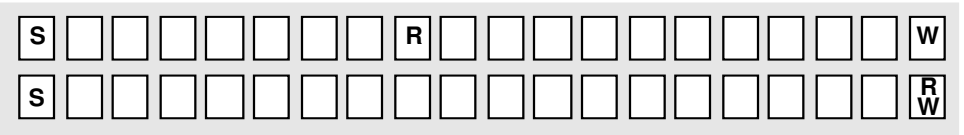
The first diagram below shows a CIS buffer before and after a call to `xil_compress()`.



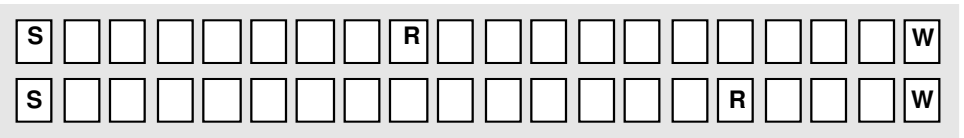
The diagram below illustrates the buffer before and after a call to `xil_decompress()`.



This illustration shows the buffer before and after a call to `xil_cis_get_bits_ptr()` that returns a pointer to the data from the read frame to the write frame minus 1 inclusive. After obtaining this pointer, you can use a function like `fwrite()` to write the data to a file.



Finally, this illustration shows the buffer before and after a seek operation. Only the read-frame index changes.



Maximum Frames and Keep Frames

The maximum-number-of-frames attribute specifies the maximum number of frames a CIS should buffer at any one time. You set this attribute using the function `xil_cis_set_max_frames()`, and you can read the value of the attribute using `xil_cis_get_max_frames()`.

Each CIS has a maximum-frames attribute whether you set it or not. If you don't set the value of the attribute, the value is a default value that depends on the compressor/decompressor that is associated with the CIS. You can set the value of the attribute by passing an integer greater than 0 to `xil_cis_set_max_frames()`. You can also pass a -1 to this function: this value indicates that there should be no limit on the number of frames that the CIS can buffer.

Note – In the case where you set the maximum-frames attribute to an integer greater than 0, this setting is actually a suggestion rather than a requirement. Some compressors/decompressors cannot function properly if a CIS's buffer is too small.

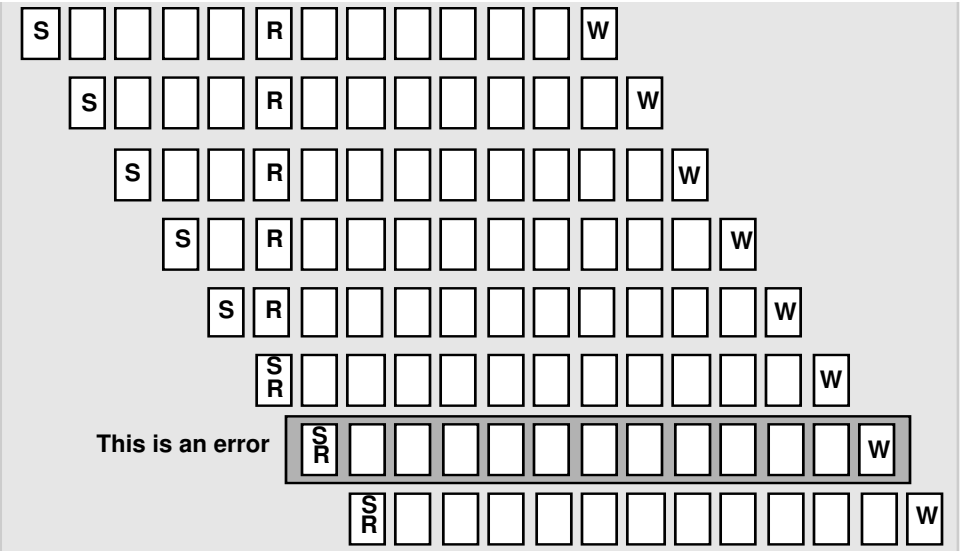
The keep-frames attribute specifies the number of frames *prior to the current read frame* that a CIS should try to keep in its buffer. As with the maximum-frames attribute, the keep-frames attribute will have a compressor-specific value if you do not set it. It you do set the attribute, you can pass to `xil_cis_set_keep_frames()` an integer representing the number of frames to buffer or a -1, which means that there is no limit on the number of keep frames. You read the value of the keep-frames attribute using the function `xil_cis_get_keep_frames()`.

Note – Like the maximum-frames attribute, the keep-frames attribute is only a hint to the compressor/decompressor. Some decompression algorithms will not work if certain already-decompressed frames, such as key frames, aren't available.

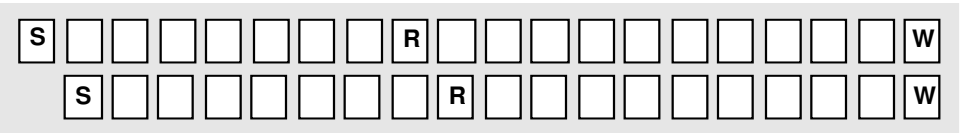
The diagrams below illustrate how the maximum-frames and keep-frames attributes affect the state of the CIS buffer.

The first diagram shows how compressing frames affects a buffer that already contains the maximum number of frames. The first line in the illustration shows the initial state of the buffer. The second line shows how the buffer looks after a call to `xil_compress()`, the third line how the buffer looks after a second call to `xil_compress()`, and so on. Note that an error occurs if, in adding new data to the buffer, you lose the current read frame. An error also occurs if R minus S becomes less than the value of the keep-frames attribute. This error is reported only once, after the operation that first causes this condition.

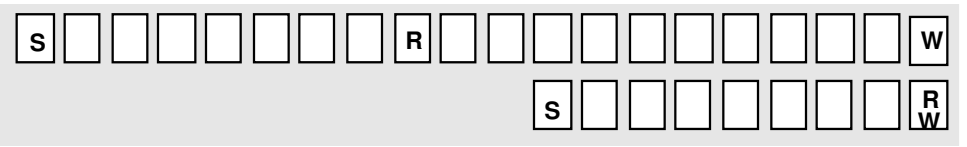
Note – An error does not occur if you cause R minus S to become less than the value of the keep-frames attribute by seeking backwards in the buffer.



In the next two illustrations, R minus S equals the value of the keep-frames attribute in the initial view of the buffer. The first diagram below shows the effect of decompressing a frame from the CIS. Note that S is incremented to prevent R minus S from exceeding the value of the keep-frames attribute. Frame S - 1 is no longer accessible.



This picture shows the CIS buffer before and after an operation that reads all the compressed data between the read frame and the write frame minus 1 inclusive. Note that R minus S equals the value of the keep-frames attributes in both depictions of the buffer.



Error-Recovery Flag

There are two types of XIL datastream errors: those that a codec or decompressor knows how to recover from and those that it can't recover from without help from the programmer. Here's an example of the former type of error. Some datastreams contain end-of-line markers. If a decompressor, while it is decoding data, finds a line that is longer or shorter than expected, it can (but doesn't have to) recover from this error by truncating or zero-filling the line. The value of a CIS's error-recovery flag determines whether a codec or decompressor automatically recovers from errors that it knows how to handle or doesn't. For information on recovering from errors that the module doesn't know how to handle, see "CIS Error Recovery" on page 267".

By default, a CIS's error-recovery flag is set to `FALSE`. This setting indicates that a codec or decompressor should not automatically recover from the type of error mentioned above.

Note – The library will generate an error structure when any type of datastream error occurs and pass that structure to the currently installed error handler. The error handler can then deal with the error as it sees fit.

To set this flag to `TRUE` in order to enable automatic error recovery, you call the function `xil_cis_set_autorecover()`.

```
void xil_cis_set_autorecover(XilCis cis, Xil_boolean on_off);
```

The parameter `cis` is a handle to the CIS whose datastream is being accessed, and `on_off` should be the enumeration constant `TRUE`. (You would use the constant `FALSE` to turn automatic recovery back off.)

The XIL library also includes a function that determines whether the error-recovery flag is currently set. This function is called `xil_cis_get_autorecover()`.

```
Xil_boolean xil_cis_get_autorecover(XilCis cis);
```

Note – If you set the error-recovery flag for a codec that cannot automatically recover from any datastream errors—like the Cell compressor—the flag has no effect.

Name

The library enables you to assign a name (`char *`) to a CIS. This type of naming is useful because it enables you to get a handle to a CIS later in your program by specifying the name of the CIS. The functions that allow for the naming of CISs are shown in Table 13-3.

Table 13-3 CIS Naming Functions

Function Name	What the Function Does
`xil_cis_set_name`	Sets the name of a CIS
`xil_cis_get_name`	Returns a copy of a CIS's name
`xil_cis_get_by_name`	Returns a handle to the CIS that has the name you specify

CIS Error Recovery

This section discusses how you recover from datastream errors that a codec or decompressor does not know how to handle. For information on handling errors that such a module can deal with, see the section "Error-Recovery Flag" on page 265.

When a datastream error occurs that a codec or decompressor cannot recover from, two things happen:

- The CIS is marked invalid. If the datastream error occurs while data is being decompressed, the CIS is marked invalid for further reading. If the error occurs while data is being compressed, the CIS is marked invalid for further writing. To determine whether a CIS is invalid, you can call either `xil_cis_get_read_invalid()` or `xil_cis_get_write_invalid()`.
- The XIL library generates an error structure and passes it to the currently installed error handler. If this is a user-installed error handler, it may take some action to deal with the datastream error.

Once a CIS has been marked invalid, there are three ways to make it valid again.

- Reset the CIS by calling `xil_cis_reset()`. This function clears all existing compressed data from the CIS.
- Seek a valid frame using `xil_cis_seek()`. If this seek fails, the library will generate a seek error.
- Ask the library to recover from the error by calling `xil_cis_attempt_recovery()`. The remainder of this section explains how to use this function.

You use the function `xil_cis_attempt_recovery()` to try to recover from a datastream error that (1) occurred while you were decompressing a CIS and (2) made the CIS invalid for further reading. The prototype for this function is shown below.

```
void xil_cis_attempt_recovery(XilCis cis, unsigned int nframes,
    unsigned int nbytes);
```

The parameter `cis` is a handle to the CIS you're working with. The parameter `nframes` indicates the maximum number of frames the function should scan in its attempt to recover from the error. The parameter `nbytes` is the maximum number of bytes the function should scan. If both `nframes` and `nbytes` are set to 0, the function can search as far forward as necessary in order to recover.

If one of the last two parameters is zero and the other is nonzero, the function behaves as follows. If `nframes` is nonzero and `nbytes` is zero, the error recovery mechanism attempts to search `nframes` frames ahead, using its best guess as to exactly how many bytes those frames would contain. If `nframes` is zero, and `nbytes` is nonzero, the recovery routine will scan `nbytes` bytes, regardless of how many frames those bytes represent.

When `xil_cis_attempt_recovery()` returns, you can check to see whether it was successful by calling the routine `xil_cis_get_read_invalid()`. If you asked the recovery routine to scan a relatively few frames or bytes, it may be necessary to call the routine several times to recover from the error.

The example below shows `xil_cis_attempt_recovery()` being used inside a user-defined error handler. (For a complete discussion of writing error handlers, see Chapter 8, "Error Handling.")

```
Xil_boolean cis_error_handler(XilError error)
{
    XilCis cis;
    XilObject object;

    if ((xil_error_get_category(error) == XIL_ERROR_CIS_DATA) &&
        ((object = xil_error_get_object(error)) != NULL) &&
        (xil_object_get_type(object) == XIL_CIS)) {

        cis = (XilCis)object;
        if (xil_cis_get_read_invalid(cis)) {
            xil_cis_attempt_recovery(cis, 0, 0);
            if (!xil_cis_get_read_invalid(cis))
                return TRUE;
        }
    }
    return xil_call_next_error_handler(error);
}
```

If the error affects a CIS and the CIS has been marked read invalid, the error handler uses `xil_cis_attempt_recovery()` to try to recover from the error. The recovery routine has permission to scan the entire datastream if necessary. The error handler returns `TRUE` if the recovery is successful. Otherwise, it calls the next error handler.

Note – When `xil_cis_attempt_recovery()` is called from within an error handler, the call to `xil_decompress()` that produced the error will fail even if the recovery is successful. The next call to `xil_decompress()` will succeed.

13