XIL Images 3

Because all XIL programs work with XIL images, you should not only know how to create an XIL image, but understand exactly what an XIL image is and what its chief characteristics are. When you first think about a computer image, you probably think of a picture or scene that has been digitized and is represented as a two-dimensional array of numbers or vectors. For example, the upper-left corner of an 8-bit, single-band image might look like this:

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

Figure 3-1 Digitized Image

3

One component of an XIL image *is* a set of pixel values like those shown above. However, an XIL image is more than that. It is a data structure containing many members, one of which is a pointer to a set of pixel values.

The upcoming sections explain some of the key things to understand about XIL images, such as:

- The most basic attributes of an XIL image
- Exporting XIL images so that an application can process pixel values using non-XIL functions
- The memory formats used for XIL images
- Special types of images, such as display images

You should understand the concepts presented in these sections before you proceed too far with your XIL programming.

The last section in this chapter completes the discussion of XIL images by talking about the image attributes that haven't already been discussed.

Basic XIL Image Attributes

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

Width, Height, and Number of Bands

These three attributes define the dimensions of an image. An image's width and height are given in pixels. Its number of bands is the number of values needed to describe a single pixel; for example, in a grayscale image, one value describes the gray level of each pixel, so the image is said to have one band. In an RGB image, three values (red, green, and blue) are required to describe a single pixel, so such an image has three bands. Each band in a multiband image must have the same width and height.

Each of these image attributes is stored as an unsigned 32-bit integer. In this release of the XIL library, the possible values for each attribute are 0 to 65,535.

One other note. These three attributes are set when you create an XIL image and cannot be changed afterwards. The functions you use to read the values of these attributes are shown in Table 3-1.

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

Function Name	What the Function Does
`xil_get_width`	Gets the width of an image in pixels
`xil_get_height`	Gets the height of an image in pixels
`xil_get_size`	Gets the width and height of an image in pixels
`xil_get_nbands`	Gets the number of bands in an image

Data Type

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

Note – IEEE single-precision floating-point images are defined and may be created, destroyed, imported, and exported, but image-processing operations on them are not implemented. To create a floating-point image, you specify a data type of `XIL_FLOAT`.

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

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

3

Exporting XIL Images

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

However, you do have the option of *exporting* the image, using the function `xil_export()`. Once an image is exported, you can obtain a complete description of how its pixel values are stored in memory using the function `xil_get_memory_storage()`. This information enables you to process the image in ways that the XIL library does not support. You can also continue to process the image using XIL functions while it is exported; however, you may lose some acceleration.

While an image is exported, you can also *set* its memory storage using the function `xil_set_memory_storage()`. This involves setting a pointer to the image's pixel values and providing other information such as the distance between pixels and the distances between scanlines. (The exact values you set depend on the data type of the image. See the section "Memory Formats for XIL Images" on page 37 for more information on this subject.) Defining a image's memory storage in this way would be appropriate, for example, if you had exported an empty image and wanted to fill it with pixel values by setting a pointer to some pixel values residing in system memory.

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

Caution – The description of memory storage that you read using `xil_get_memory_storage()` or set using `xil_set_memory_storage()` is valid only as long as your image remains exported. Once you import the image, both the address at which the image's pixel values are located and their layout in memory may change. Trying to access pixel values using an invalid pointer or invalid information about their layout can cause serious problems in your application.

The XIL functions related to importing and exporting XIL images are shown in Table 3-2.

Table 3-2 Functions for Importing and Exporting Images

Function Name	What the Function Does
`xil_import`	Imports an image so that it must be processed using XIL functions
`xil_export`	Exports an image so that you can get or set the image's pointer to its pixel values
`xil_get_exported`	Determines whether an image is currently exported
`xil_get_memory_storage`	Reads information about how an image is stored in memory (if the image is exported)
`xil_set_memory_storage`	Defines how an image's pixel values will be stored in memory (if the image is exported)

Memory Formats for XIL Images

The memory format for an XIL image depends primarily on its data type. Images containing 8-bit or 16-bit data elements are stored in a pixel-sequential format, and images containing 1-bit data elements are stored in a band-sequential format.

`XIL_BYTE` and `XIL_SHORT` Images

Figure 3-2 illustrates how a 3-band image with 8-bit data elements might be stored in memory.

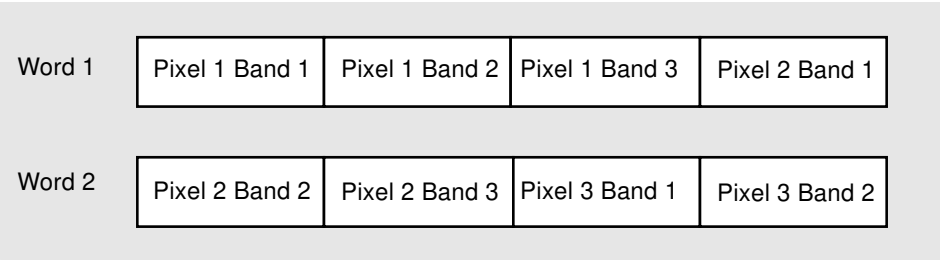

Figure 3-2 Memory Format for a 3-Band Image Containing 8-Bit Data Elements

Similarly, Figure 3-3 shows how a 3-band image with 16-bit data elements might be stored.

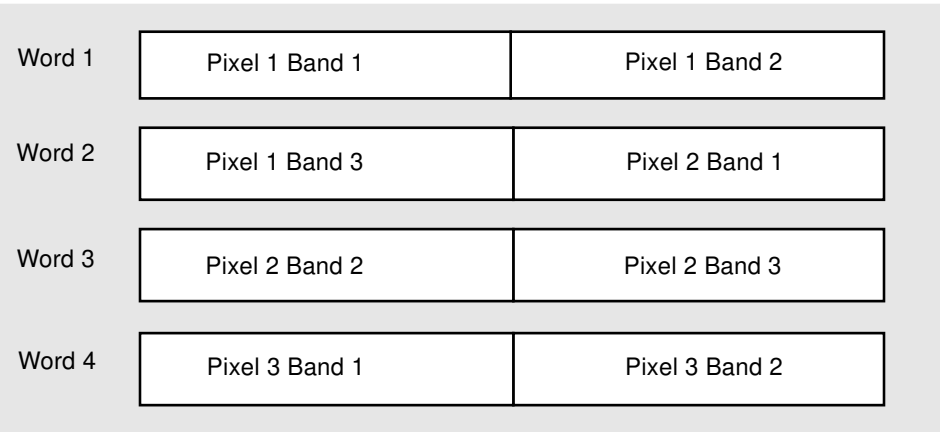

Figure 3-3 Memory Format for a 3-Band Image Containing 16-Bit Data Elements

It is important to realize, however, that there may be unused bytes between pixels and between scanlines.

As long as you're processing an image using XIL functions, you don't need to know the details about how the image is stored in memory. However, there may be occasions when you want to process an image directly. To do this, you

must export the image and then call `xil_get_memory_storage()` to obtain the details about how the pixels values are stored. This function returns a union of structures of the type defined below.

```
typedef struct {
    Xil_unsigned8 *data;
    unsigned int scanline_stride;
    unsigned long band_stride;
    unsigned char offset;
} XilMemoryStorageBit;

typedef struct {
    Xil_unsigned8 *data;
    unsigned long scanline_stride;
    unsigned int pixel_stride;
} XilMemoryStorageByte;

typedef struct {
    Xil_signed16 *data;
    unsigned long scanline_stride;
    unsigned int pixel_stride;
} XilMemoryStorageShort;

typedef union {
    XilMemoryStorageBit bit;
    XilMemoryStorageByte byte;
    XilMemoryStorageShort shrt;
} XilMemoryStorage;
```

The structures that contain information about `XIL_BYTE` and `XIL_SHORT` images both contain an element called `data`. This is a pointer to the beginning of the pixel data. The structures also contain elements called `pixel_stride` and `scanline_stride`. For an `XIL_BYTE` image, these elements indicate the distance in bytes between pixels and between scanlines, and for an `XIL_SHORT` image, they indicate the distance in 16-bit units between pixels and between scanlines.

This information about an image's memory storage remains valid only as long as your image is exported. Once you reimport the image, the location of your image's pixel values in memory and the layout of those values may change (except that the pixels will always remain in a pixel-sequential order).

XIL_BIT Images

Unlike images containing 8- and 16-bit data elements, multiband images containing 1-bit data elements are stored in a band-sequential format. In addition, scanlines are padded to the nearest byte boundary if necessary. For instance, Figure 3-4 shows how a 3-band, 3-by-3 image containing 1-bit data elements might look in memory.

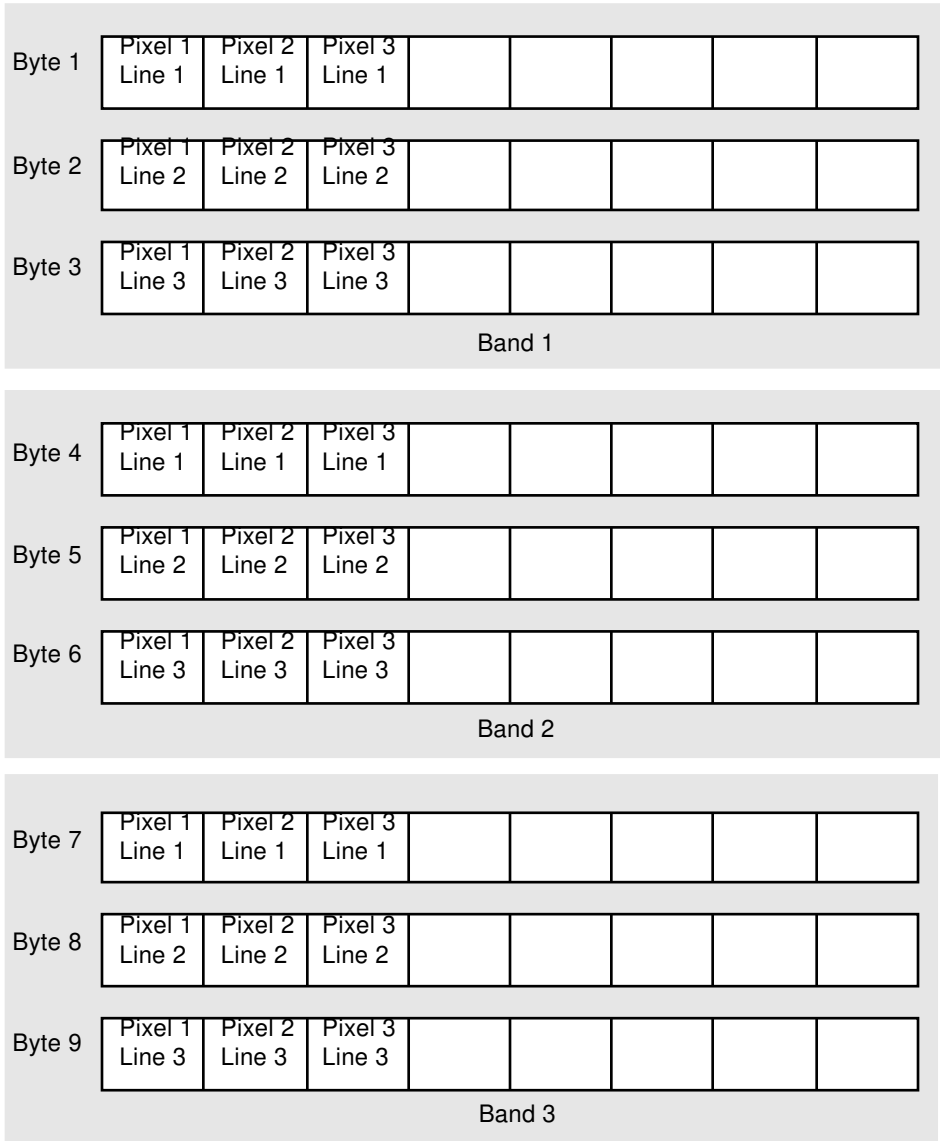

Figure 3-4 Memory Format for a 3-Band Image Containing 1-Bit Data Elements

To obtain a complete description of how an `XIL_BIT` image is stored in memory, however, you must export the image and call `xil_get_memory_storage()`. The function will give you access to the structure members shown below.

```
typedef struct {
    Xil_unsigned8 *data;
    unsigned int scanline_stride;
    unsigned long band_stride;
    unsigned char offset;
} XilMemoryStorageBit;
.
.
.
typedef union {
    XilMemoryStorageBit bit;
    XilMemoryStorageByte byte;
    XilMemoryStorageShort shrt;
} XilMemoryStorage;
```

The member `data` is a pointer to the first byte of pixel data. The member `scanline_stride` indicates the number of bytes from the first byte of one scanline to the first byte of the next, and `band_stride` indicates the number of bytes from the beginning of one band to the beginning of the next. The member `offset` specifies the number of *bits* from the beginning of a scanline to the first bit that contains a pixel value.

Remember that the values of all these structure members remain valid only as long as the image is exported.

Memory Formats for Images of Different Color Spaces

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

Types of XIL Images

There are two basic types of XIL images:

- Memory images
- Device images

As you will see in the upcoming section "Device Images," device images can be further subdivided into images that represent displays and images that represent other devices. Images that represent displays are referred to as *display* images. The main difference between these two subtypes of device image is the way in which they are created.

Memory Images

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

Device Images

XIL device images represent devices such as displays, cameras, frame grabbers, scanners, and printers. For each device image, the structure describing the image is stored in system memory, but the image's pixel values are read from or written to the device. To create a device image, you generally pass the name of a device to the function `xil_create_from_device()`. The exception to this rule is that to create a display image, you call the function `xil_create_from_window()`.

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

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

Display Images

Display images are a special subclass of device images. They are very valuable because they eliminate the need for the scenario shown in Figure 3-5.

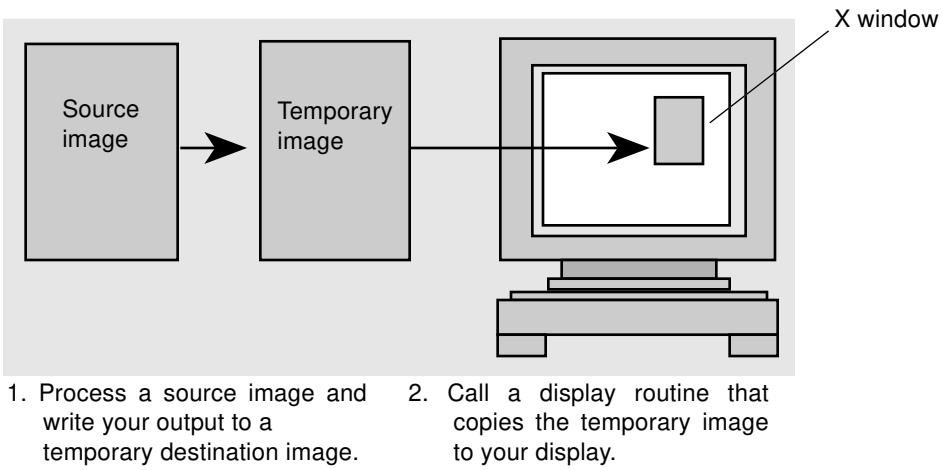

Figure 3-5 Copying Data from a Temporary Image to the Display (Not Necessary)

Display images enable the XIL library to process a source image and write the results of the processing directly to an X window by using a display image as the destination image for the operation. In other words, functions can write directly to frame-buffer memory, so the need for the temporary image is eliminated (see Figure 3-6).

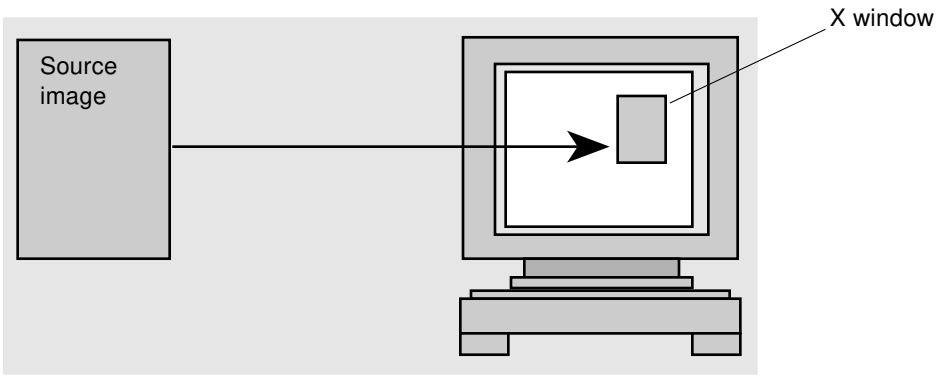

Figure 3-6 Writing the Output of an Operation to a Display Image

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

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

A display image can also serve as a source image in most cases (if the display is readable). Any pixels that are obscured by another window when a read takes place are set to 0.

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

- 1-band images containing 1-bit data elements
- 1-band images containing 8-bit data elements
- 3-band images containing 8-bit data elements

Additional XIL Image Attributes

The remainder of this chapter discusses some additional image attributes:

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

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

Origin

An image's *origin* is a pair of floating-point numbers that define a point in the image's coordinate plane. By default, an image's origin is in its upper-left corner (0.0, 0.0), but you can change the origin using the function `xil_set_origin()`. When an operation is performed, the origins of the source image or images and the destination image are aligned. (The floating-point origin values are rounded to integers for this purpose.) Then, for all nongeometric operations, the intersection of the source and destination images determines which pixels in the destination image will be modified. See Figure 3-7.

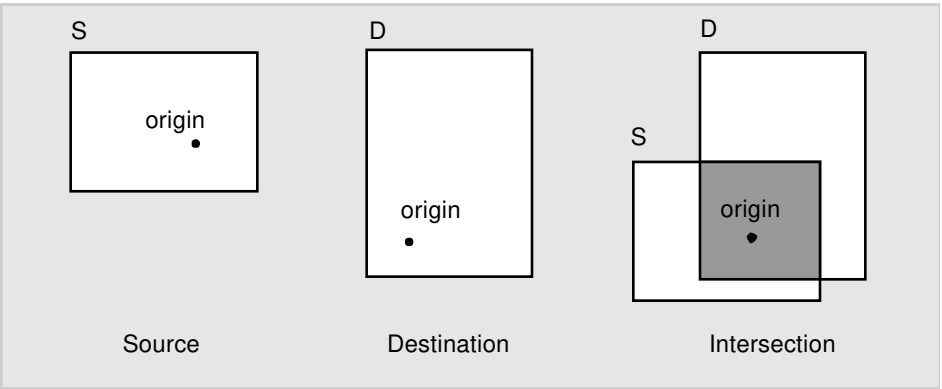

Figure 3-7 Image Origins

Only the shaded area shown in Figure 3-7 is modified in the destination image, and only the shaded area in the source is used by the operator.

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

Table 3-3 Functions for Reading or Setting an Image's Origin

Function Name	What the Function Does
`xil_get_origin`	Gets both coordinates of an image's origin
`xil_get_origin_x`	Gets the *x* coordinate of an image's origin
`xil_get_origin_y`	Gets the *y* coordinate of an image's origin
`xil_set_origin`	Sets an image's origin

3

Region of Interest

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

Since an ROI is an attribute of an image, as opposed to a parameter for an operation, two or three ROIs may be involved in a single operation; for example, the operation's source image may have one ROI, and the operation's destination image another. In this case, the ROI used for the operation is the intersection of the ROIs associated with the source and destination images, as shown in Figure 3-8.

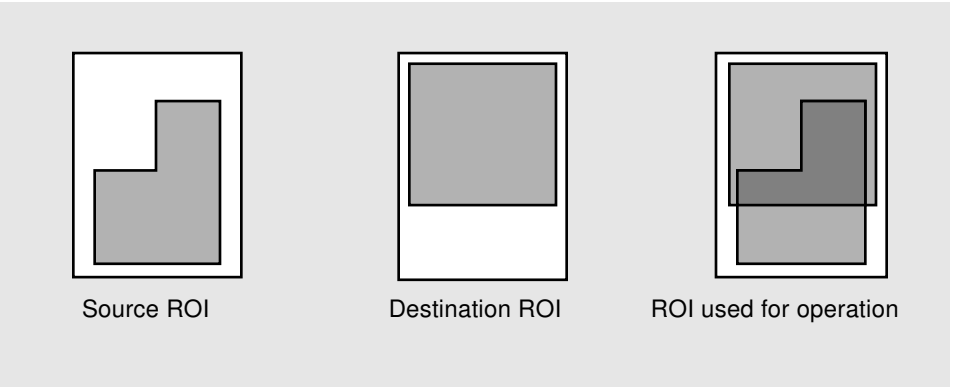

Figure 3-8 Regions of Interest

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

If the origin of the source or destination has been set to something other than 0.0, the origins are first aligned; then, the ROI is determined. See Figure 3-9.

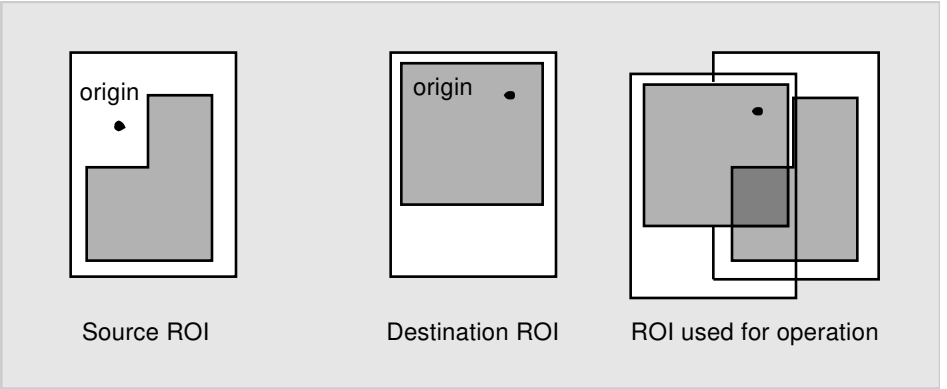

Figure 3-9 Regions of Interest and Origins

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

Creating and Destroying an ROI

There are several ways to create an ROI.

- You can create a new ROI data structure using the function `xil_roi_create()`, which returns a handle to the ROI. When first created, the ROI is empty. For information on how to set values in the ROI, see the next section "Building an ROI."
- You can create a copy of an existing ROI using the function `xil_roi_create_copy()`.
- You can get a copy of the ROI associated with a image using the function `xil_get_roi()`.
- You can create an ROI by taking the union or intersection of two existing ROIs. The functions you use for these operations are `xil_roi_unite()` and `xil_roi_intersect()`.

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

Building an ROI

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

Table 3-4 Functions Used to Build an ROI

Function Name	What the Function Does
`xil_roi_add_rect`	Adds a rectangle to the ROI. You specify the width and height of the rectangle and its coordinates.
`xil_roi_subtract_rect`	Subtracts a rectangle from an ROI.
`xil_roi_add_image`	Adds an `XIL_BIT` image to the ROI. Bits that are set in the image are added to the ROI.
`xil_roi_add_region`	Adds an X region to an ROI.

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

Setting an Image's ROI Attribute

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

Translating an ROI

The library includes a function `xil_roi_translate()` that moves all the pixels in an ROI that have been set to 1 left or right and/or up or down. The prototype for this function is shown below.

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

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

The function returns a handle to the translated ROI.

Naming an ROI

The XIL library enables you to specify a string that will serve as the name for an ROI. This naming is useful because it enables you to later get a handle to an ROI by using its name. The functions that allow you to perform these tasks are listed in Table 3-5.

Table 3-5 ROI Naming Functions

Function Name	What the Function Does
`xil_roi_set_name`	Sets the name of an ROI
`xil_roi_get_name`	Returns a copy of an ROI's name
`xil_roi_get_by_name`	Returns a handle to the ROI that has the name you specify

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

Color Space

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

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

Images are usually assigned color spaces so that an image can be converted from one color space to another (using the `xil_color_convert()` function). For example, if you had an RGB image and wanted to convert it to the YC_bC_r color space, you would follow this procedure:

1. Set the color-space attribute of your RGB image to one of the RGB color spaces listed above.
2. Create an image whose width, height, number of bands, and data type match those of the RGB image.
3. Set the new image's color-space attribute to one of the YC_bC_r color spaces listed above.
4. Use the function `xil_color_convert()` to convert the RGB data from the first image to its YC_bC_r counterpart and write the converted data to the newly created image.

For more information about color-space conversion, see the section "Color-Space Conversion" on page 122.

Parent

The XIL library enables you to create a child image (subimage) from an image (the child's parent). Such a child image has an attribute that identifies its parent image. To retrieve this attribute, a handle to the parent image, use the function `xil_get_parent()`.

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

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

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

Table 3-6 Parameters to `xil_create_child()`

Parameter	How It Is Used
`src`	A handle to the parent image
`xstart`	The horizontal offset in pixels from the upper-left corner of the source image to the upper-left corner of the child
`ystart`	The vertical offset in pixels from the upper-left corner of the source image to the upper-left corner of the child
`width`	The width of the child image in pixels
`height`	The height of the child image in pixels
`startband`	The offset in bands from the first band in the parent to the first band in the child
`numbands`	The number of bands in the child image. These bands must match contiguous bands in the parent.

Note – A child image does not get a copy of part of the parent's pixel data, but a pointer to that data.

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

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

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

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

Image Type

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

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

Table 3-7 Image Type Utility Functions

Function Name	What the Function Does
`xil_imagetype_get_width`	Gets the width of an image type in pixels
`xil_imagetype_get_height`	Gets the height of an image type in pixels
`xil_imagetype_get_size`	Gets the width and height of an image type in pixels
`xil_imagetype_get_nbands`	Gets the number of bands in an image type
`xil_imagetype_get_datatype`	Gets the data type of an image type
`xil_imagetype_get_info`	Gets the width, height, number of bands, and data type of an image type
`xil_imagetype_set_name`	Sets the name of an image type
`xil_imagetype_get_name`	Returns a copy of an image type's name
`xil_imagetype_get_by_name`	Returns a handle to an image type that has the name you specify

Synchronization Flag

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

Readable and Writable Flags

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

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

This function returns `TRUE` if the device image is readable.

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

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

This function returns `TRUE` if the device image is writable.

Name

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

Table 3-8 Image Naming Functions

Function Name	What the Function Does
`xil_set_name`	Sets the name of an image
`xil_get_name`	Returns a copy of an image's name
`xil_get_by_name`	Returns a handle to the image that has the name you specify