Handling Input and Output 4

This chapter takes a systematic look at how you handle the reading and writing of images in an XIL program. Basically, input can come from, and output can go to, three places:

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

The following sections explain the different possibilities.

Note – This chapter deals exclusively with reading and writing single images. As we'll discuss in Chapter 12, "Compressing and Decompressing Sequences of Images," XIL programs can also read and write data streams that represent compressed video or multipage documents. These data streams are stored in an XIL data structure called a compressed image sequence. Moving data into and out of these structures is discussed in Chapter 12.

Moving Image Data from a File to an XIL Image

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

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

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

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

   The parameter `state` is the system-state data structure that was returned when you called `xil_open()` to initialize the library. The parameters `width`, `height`, and `nbands` are integers (`unsigned int`) representing the image's width, height, and number of bands. The final parameter `datatype` is an enumeration constant of type `XilDataType`, whose definition is shown below.

```
enum XilDataType { XIL_BIT, XIL_BYTE, XIL_SHORT };
```

   The constant `XIL_BIT` indicates that the image contains 1-bit data; the constant `XIL_BYTE` indicates that the image contains unsigned 8-bit data; and the constant `XIL_SHORT` indicates that the image contains signed 16-bit data. These are the only data types supported by the XIL library.

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

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

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

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

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

4. **Get information about how your XIL image's pixel values are stored in memory.**
   To obtain this information, you call the function `xil_get_memory_storage()`, whose function prototype is shown below.

```
Xil_boolean xil_get_memory_storage(XilImage image,
    XilMemoryStorage *storage);
```

The function's return value indicates whether the routine succeeded (`TRUE`) or failed (`FALSE`). The parameter `image` is a handle to an XIL image. The argument `&storage` is a pointer to a union of structures defined like this:

```
typedef struct {
    Xil_unsigned8 *data;  /* pointer to first byte of image */
    unsigned int scanline_stride;  /* number of bytes between scanlines */
    unsigned long band_stride;  /* number of bytes between bands */
    unsigned char offset;  /* number of bits to the first pixel */
} XilMemoryStorageBit;

typedef struct {
    Xil_unsigned8 *data;  /* pointer to first byte of image */
    unsigned long scanline_stride;  /* number of bytes between scanlines */
    unsigned int pixel_stride; /* number of bytes between pixels */
} XilMemoryStorageByte;

typedef struct {
    Xil_signed16 *data;  /* pointer to first 16-bit word of image */
   unsigned long scanline_stride;  /* no of 16-bit words between scanlines */
    unsigned int pixel_stride;  /* number of 16-bit words between pixels */
} XilMemoryStorageShort;

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

The structure you should look at for information depends on the data type of your image. If it is an `XIL_BIT` image, look at the structure `bit`; if it is an `XIL_BYTE` image, look at the structure `byte`; and if it is an `XIL_SHORT` image, look at the structure `shrt`. The elements in the appropriate structure will provide you with the information you need to write your image to memory in the proper location and in the proper format.

5. **Read the pixel values stored in your image file and write them to memory using the information obtained in the preceding step.**
How you perform this step depends on several factors:

   - The type of data in your image: 1-bit, 8-bit, or 16-bit.
   - The organization of the data elements in your image file. For multiband images, the elements may be arranged in a pixel-sequential or band-sequential format.
   - The organization of the memory allocated to hold your XIL image. The information you obtained in step 4 describes this organization.

   One of the examples supplied with the XIL library contains a function called `load_file()` that illustrates how to handle this step for the cases listed in Table 4-1.

Table 4-1 Cases Handled by the Function `load_file()`

Data Type	File Format	Memory Layout for XIL Image
`XIL_BIT`	Band sequential	Any
`XIL_BYTE`	Pixel sequential	Any
`XIL_SHORT`	Pixel sequential	Any

You can find this routine in the directory `$XILHOME/examples/display`.

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

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

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

Caution – After you import an image you give up control of it, and the pointers you received for it from a previous `xil_export()` are invalid. If you need to use an image's data after the image has been imported, you must call `xil_export()` again to export the image, and then call `xil_get_memory_storage()` again to get new pointers to the image data. After accessing the data, be sure to get the pixel stride and scanline stride when accessing pixel values.

For example, although an RGB image will always contain blue values, followed by green values, followed by red values, the data layout in XIL memory is arbitrary; there may be any number of pixels between the next set of blue, green, and red values. Access the pixel address of the first blue value, or the pixel stride to the next blue value, or the scanline stride to the next scanline, through the values returned by `xil_get_memory_storage()`. For more information, see "Exporting XIL Images" on page 36 and "Memory Formats for XIL Images" on page 37.

Moving Image Data from an XIL Image to a File

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

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

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

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

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

   You may want to read other attributes of your XIL image depending on what type of information about the image you must write to your image file. See the section "Additional XIL Image Attributes" on page 46 for a list of image attributes you may be interested in.

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

3. **Get information about how your image's pixel values are stored in memory.**
   You get this information by calling the function `xil_get_memory_storage()`. If you haven't read the section "Moving Image Data from a File to an XIL Image" on page 57, see that section for an explanation of what information this function provides.

4. **Write the appropriate header information and pixel values to your file.**
   At this point, you should have all the information that you plan to write to your file's header, a pointer to your XIL image's pixel values, and information about the layout of those values in memory. So your only remaining XIL-related concern should be "Do I want to write the pixel

values to my file in the format in which they are stored in memory?" To answer this question, you need to know how pixel values for XIL images are stored in memory (see the section "Memory Formats for XIL Images" on page 37) and what kind of organization of data your file format demands.

Sending Output to (and Reading Input from) the Display

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

The steps listed below discuss this procedure in a bit more detail:

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

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

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

---

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

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

   The parameter `state` is the system-state data structure that was returned when you called `xil_open()` to initialize the XIL library. The parameters `display` and `window` are of data types defined in Xlib. The `display` parameter is a pointer to a structure that is returned when you initially connect to the X server. It contains information about the server and the screens the server controls. The parameter `window` is the ID of the X window you created in the previous step. The return value of `xil_create_from_window()` is a handle to your newly created display image.

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

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

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

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

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

Possible Complications

One question we ignored in the preceding discussion is "What if the depths of your source image and your X window are different?" What if you have a source image that has three bands and contains 8-bit data (24 bits per pixel) and an X window that is 8 bits deep? The answer to this particular question might be that you call an XIL function to dither the source image from 24 bits to 8 bits before writing it to the window. But other types of mismatches can occur. Chapter 6, "Preparing Images for Display," discusses an example program that handles many of these mismatches.

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

- `window.c`, which is located in the subdirectory `example1`
- `window.c` (a modified version of the routine mentioned above), which is located in the subdirectory `display`
- `xilcis_color.c`, which is located in the subdirectory `movie_player_example`.

Reading a Display Image

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

1. **If you don't already have an XIL image that you can read the image data in the frame buffer into, you must create one.**
   You create this image using the function `xil_create()`. The image you create must have the proper width, height, number of bands, and data type.
2. **Perform an XIL operation using the display image as your source image.**
   For example, you could use the function `xil_copy()` to copy an image from your screen to an XIL memory image. Any pixels in the on-screen image that are obscured by another window when the read occurs are cleared to 0.

Reading and Writing Devices Other than Displays

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

Each device type has attributes associated with it. For example, the Photo CD reader that comes with the XIL library (see Chapter 5, "Reading Kodak Photo CD Images") has a FILEPATH attribute that indicates the Photo CD image you want to read, and a RESOLUTION attribute that specifies a display resolution.

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

- The device needs one or more attributes defined at the time the device is created.
- Multiple attributes are interdependent and need to be set simultaneously for the device.

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

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

Initializing a Device's Attributes

As mentioned in the previous section, some device types require you to initialize one or more device attributes before creating an image for the device. To initialize a device's attributes in the XIL library, you create an XIL data structure of type `XilDevice`. Once this device object exists, you use it to store the device-initialization values. You then pass the device object as an argument on the `xil_create_from_device()` function that creates the device image, as discussed in "Creating a Device Image" on page 71; this applies to the device image all the attributes stored by the device object.

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

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

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

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

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

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

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

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

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

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

   The device object's only use is to initialize device attributes when you call the `xil_create_from_device()` function that creates the device image. It cannot be used to adjust a device image's attributes after the device image is created; `xil_set_device_attribute()` does that, as discussed on page 72. However, after using the device object to create one device image, you can use the same object to store different initialization attributes, then use the modified device object when you create another device image of the same type.

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

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

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

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

   To set multiple attributes, make a separate `xil_device_set_value()` function call for each; although the attributes are set individually on the device object, they are applied simultaneously to the device image when that device image is actually created (see "Creating a Device Image" on page 71). You can set as many attributes as you need to derive all required initialization attributes for the device.

   For example, if you created `deviceObj` for a video card as shown in Step 1 on page 69, you might initialize its volume and speed attributes as shown in the following code fragment:

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

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

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

4. **Destroy the device object.**
   You can do this immediately after creating the device image. However, if you plan to create other devices of the same type, you may want to keep the device object around to initialize those other devices. If desired, you can set different initialization values on the object. When you're done using it, destroy the object; "Destroying a Device Object" on page 73 tells you how.

Creating a Device Image

As explained in "Initializing a Device's Attributes" on page 68, some device types require that one or more device attributes be initialized before the device image is created; for those devices, initialize the required attributes before following the steps in this section. For devices that don't support initialized attributes, the first step for using the device is to create a device image for it. Once the device image is created, you can set its required attributes, then use the image in an XIL operation.

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

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

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

   The parameter `state` is the system-state data structure that was returned to you when you called `xil_open()` to initialize the XIL library. The `devicename` parameter is the name of the device to be associated with the device image; the name of the device must be provided by the group that writes the device handler. If you initialized attributes for this device type, the device name should be the same name specified on the `xil_device_create()` function shown in Step 1 on page 69. The final parameter, `deviceObj`, is the device object you created to initialize the device's attributes. If you didn't create a device object for this device type, pass NULL for this parameter.

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

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

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

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

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

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

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

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

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

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

operation's destination image is a device image, the operation will write an image to the associated device. For instance, the operation might print a color image on a color printer.

Destroying a Device Object

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

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

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

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

4