Presentation Functions 7

This chapter covers a group of functions you are likely to use in displaying images, or printing them. These functions enable you to:

- Copy a source image to a destination image
- Copy specified image planes to a destination image
- Rescale the values in an image
- Cast an image from one data type to another
- Dither an image
- Convert an image from one color space to another
- Prepare CMY images for printing

These tasks are discussed in detail in the sections below.

Copying an Image to the Display

The `xil_copy()` function lets you copy a source image to a destination image. `xil_copy()` copies each *plane* (bit) in a source-image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel. To control the pixel planes copied from the source to the destination, you can define a *plane mask* to specify the bit planes to copy, then call `xil_copy_with_planemask()` to perform the copy.

The `xil_copy()` and `xil_copy_with_planemask()` functions are discussed separately in the following sections.

7

Copying All Bit Planes

The simplest way to display an image is to use `xil_copy()` to copy your source image to the display. The `xil_copy()` function copies each *plane* (bit) in a source-image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel. For example, an eight-bit-deep frame buffer has eight bit planes; if you use `xil_copy()` to copy an `XIL_BYTE` image to that frame buffer, all eight planes of each source pixel are copied to all eight planes in the frame buffer's destination pixel.

For `xil_copy()` to work, you must have created a display image that will serve as the destination for the copy. (For information about what display images are and how you create them, see the section "Display Images" on page 44.) In addition, the source image—the image to be displayed—and the display image must use the same number of bits to represent a pixel. Table 7-1 lists the combinations of source and display images that will match in this way.

Table 7-1 Matching Source and Display Images

Source Image Type	Required Display Image Depth
Single-band `XIL_BIT` image	One bit
Single-band `XIL_BYTE` image	Eight bits
Three-band `XIL_BYTE` image	Twenty-four bits

If your source and destination images match, you can display the source image using code similar to this.

```
XilImage src, display_image;

xil_copy(src, display_image);
```

Besides performing this kind of explicit copy using `xil_copy()`, you can also request that an image-processing function (like a rotate function) copy its output to the display. You do this by specifying a display image as the

destination for the operation. For example, the following call to `xil_rotate()` rotates the source image and copies the rotated image to the display.

```
XilImage src, display_image;

xil_rotate(src, display_image, "bilinear", -0.7854);
```

Again, before this type of operation will work, the source and display images must use the same number of bits to represent a pixel. If your source and display images do not match in this respect, you must perform one or more of the tasks discussed in this chapter before displaying your image.

If overlapping but not coincident sibling images (children of the same parent) are specified as the source and destination, `xil_copy()` detects the overlap and correctly generates the destination image. All other operations generate a warning message under these conditions and have undefined results, as mentioned on page 54 in the discussion on parent and child images.

Copying Only the Planes Defined in a Plane Mask

The `xil_copy()` function copies each *plane* (bit) in a source-image pixel to the destination image pixel, replacing the value in each corresponding plane in the destination pixel.To control the pixel planes copied from the source to the destination, you can define a *plane mask* to specify the bit planes to copy, then call the `xil_copy_with_planemask()` function to perform the copy. The prototype for `xil_copy_with_planemask()` is shown below.

```
void xil_copy_with_planemask(XilImage src, XilImage dst,
    unsigned int planemask[]);
```

A typical reason to copy with plane mask control is to overlay one image over another. Because you can copy specific planes from each source-image pixel, the underlying image isn't visible where the overlay is drawn; however, all planes from the underlying image are still available and can be refreshed in the destination by rendering it with another plane mask. Overlays can improve performance by reducing the amount of graphic information that has to be redrawn; they can also be used to highlight graphics for selection.

Note – Because read/write color cells can be allocated only in PseudoColor and DirectColor visuals, you need to provide an alternative technique to overlays for other visuals; you may also want to provide an alternative in case the overlay fails. For more information on overlays and their advantages and disadvantages, see the *Xlib Programming Manual*.

When you use `xil_copy_with_planemask()`, each pixel in the destination image is defined by the following operation:

```
dst = (dst & ~mask) | (src & mask)
```

Here, `dst` is the destination image, `mask` is the plane mask, and `src` is the source image. Thus, if the plane-mask bit is "on," the copy overwrites the corresponding bit in the destination image; otherwise, the bit in the destination image is unchanged.

As an example, assume you have two single-band XIL_BYTE images you want to overlay. Table 7-2 shows the plane masks you might use.

Table 7-2 Plane Masks for an Overlay

Plane Mask	Binary Base	Hexadecimal Base	Decimal Base
Mask 1	00000001	0x1	1
Mask 2	11111110	0xfe	254

In Table 7-2, Mask 1 has only its low order bit turned on; thus, it ensures that only the source image's low order bit is copied to the destination. Mask 2, on the other hand, has its seven high-order bits turned on, so it ensures that the source image's seven high-order bits are copied to the destination.

The code below shows how you might define and use the plane masks shown in Table 7-2:

```
XilImage src1, src2, dst;
unsigned int planemask1[1], planemask2[1];

planemask1[0] = 0x1;
planemask2[0] = 0xfe;

xil_copy_with_planemask(src1, dst, planemask1);
xil_copy_with_planemask(src2, dst, planemask2);
```

The plane mask must be an array of unsigned integers. The number of array elements must match the number of image bands; each array element specifies the plane mask for the corresponding band in the destination image. Both the source and destination images must have the same type and number of bands, and in-place operations are supported.

When using a plane mask for copying an image to the display, the depth of the window is the upper limit on the number of meaningful bits you can set in the plane mask, and you must manipulate the colormap to get a reasonable display. For more information on allocating color cells for overlays, see the *Xlib Programming Manual*.

In addition to overlays, you can use a plane mask for double buffering on hardware that doesn't have separate memory buffers. Double buffering is useful for animation because you can render an image in a hidden memory buffer while you display another image in a second buffer. Once rendering is complete in the hidden buffer, you can display its contents, then hide the second buffer and render a new image into it. Rendering into the hidden buffer and quickly switching between buffers smooths the transition between images. As with overlays, double buffering requires you to toggle between two colormaps as you change plane masks between images. For more information on using double-buffering for animation, consult Foley, et al. *Computer Graphics: Principles and Practice* (see Appendix E, "Bibliography").

7

Rescaling an Image

The XIL function `xil_rescale()` maps the values in each band of an image from one range (for example, 0 to 2047) to another range (for example, 0 to 255). The function performs this mapping by multiplying each value in a band by one constant and then adding a constant to the result of the multiplication.

The prototype for `xil_rescale()` is shown below.

```
void xil_rescale(XilImage src, XilImage dst, float *scale,
    float *offset);
```

The parameters `src` and `dst` are handles to the source and destination images. The parameter `scale` is an array of floating-point numbers that will serve as the constants by which values in the bands of the source image are multiplied. The values in band 0 are multiplied by `scale[0]`, the values in band 1 are multiplied by `scale[1]`, and so on. Therefore, the number of elements in the array must match the number of bands in the image. The final parameter, `offset`, is an array of floating-point numbers, one of which is added to the scaled values in each band of the source image.

There are many cases in which this function will be useful as you prepare an image for display. A few of these are mentioned below.

One obvious case is that in which you want to display an image containing `XIL_SHORT` values. For example, say that you want to display a single-band `XIL_SHORT` image in an X window created using an 8-bit `GrayScale` visual. The values in your image may range from -32768 to 32767, but the values you write to the display image must fall in the range 0 to 255 (or some subset of that range). To handle this situation, you could perform these steps.

1. **Determine the minimum and maximum values in your source image.** You determine the minimum and maximum values using the function `xil_extrema()`. Your image will look better when displayed if you map the minimum value to 0 and the maximum value to 255, as opposed to mapping -32768 to 0 and 32767 to 255. In the latter case, you might wind up with very few gray levels in the image to be displayed.

2. **Rescale the values in your `XIL_SHORT` image so that they fall in the range 0 to 255.**
   The code you might use to accomplish this step is shown below.

```
XilImage src, dst;
float maximum[1], minimum[1];
float multiplier[1], offset[1];

multiplier[0] = 255 / (maximum[0] - minimum[0]);
offset[0] = -((minimum * 255) / (maximum[0] - minimum[0]));
xil_rescale(src, dst, multiplier, offset);
```

3. **Cast the values in the rescaled image to be of type `XIL_BYTE`.**
   Although you have rescaled your image so that its values fall in the range 0 to 255, those values are still 16-bit signed values. Before copying your image to the display, you must use the function `xil_cast()` to create an `XIL_BYTE` version of the image. For more information about `xil_cast()`, see the next section, "Casting an Image from One Data Type to Another."

Here's another case in which you may need to rescale an image before displaying it. Suppose you want to display an 8-bit grayscale image in an 8-bit `GrayScale` window, but that you do not want to use the currently installed X colormap in displaying it. You want to create a new virtual X colormap, store a grayscale ramp in that colormap, and have it installed when your application is active. If you write values to all 256 colorcells in the virtual colormap, you're almost certainly going to see colormap flashing when the colormap is installed, so you may decide not to write values to the first 16 colorcells—to write all the values needed to display your image in colorcells 17 through 255. This strategy requires that you rescale the values in your image so that they fall in the range 17 to 255.

Casting an Image from One Data Type to Another

The XIL function `xil_cast()` casts an XIL image from one data type to another. The function's prototype is shown below.

```
void xil_cast(XilImage src, XilImage dst);
```

The parameter `src` is a handle to your source image, and `dst` is a handle to a destination image. This destination must have the same width, height, and number of bands as the source image and must have the data type to which you want to cast the source image.

There are a number of instances in which you may need to use this function as you prepare an image for display. For example, you may have a single-band `XIL_SHORT` image that you want to display in an 8-bit window. To do this you need to follow these steps.

1. **Rescale the image.**
   Unless the values in your `XIL_SHORT` image already fall in the range 0 to 255, you must use the function `xil_rescale()` to map them to that range, or a subset of that range. For more information about rescaling images, see "Rescaling an Image" on page 102.

2. **Cast the `XIL_SHORT` image to an `XIL_BYTE` image.**
   Before you can display your image in an 8-bit window, you must cast the 16-bit values in the source image to 8-bit values. See the code fragment below.

```
XilImage short_image, byte_image;
unsigned int width, height, nbands; /* Dimensions of images */

byte_image = xil_create(state, width, height, nbands,
    XIL_BYTE);
xil_cast(short_image, byte_image);
```

Similarly, if you have an `XIL_BIT` image that you want to display in an 8-bit window, you must cast the source image to an `XIL_BYTE` image before displaying it. In this cast, the values 0 and 1 in the `XIL_BIT` image are cast to indices 0 and 1 in the `XIL_BYTE` image. If you want different indices, convert the image by passing it through a lookup table rather than by casting its data type. For information about lookup tables, see "Passing an Image Through a Lookup Table" on page 199.

Note – When casting the image so you can display it, you may want to use a display image as the destination image for the cast. However, if you intend to use the converted image again, you can cast the source image to an interim destination image, then use `xil_copy()` to copy the interim image to the display.

Dithering an Image

The XIL library provides several functions you can use to prepare images for display by dithering them. Before looking at these individual functions, though, the section explains what it means to dither an image in an XIL application.

What Is Dithering?

First, you need to know that the XIL library gives you the ability to pass a single-band image of any data type through a lookup table to produce a single-band or multiband image of the same or another data type. The lookup table used for this operation is a data structure of type `XilLookup` and has (among others) the following attributes:

- An input data type
- An output data type
- A number of bands on the output side

Figure 7-1 shows a single-band `XIL_BIT` image being passed through a lookup table to produce a three-band `XIL_BYTE` image.

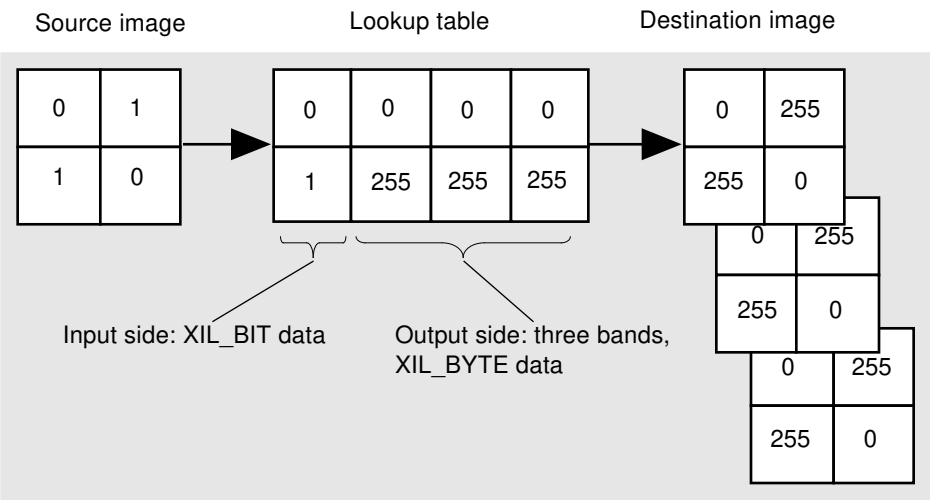

Figure 7-1 XIL Lookup Operation

A dither operation is an *inverse* lookup operation. As Figure 7-2 indicates, the dither operation matches a value (or values) in the source image—the image to be dithered—with a value from the *output* side of the lookup table being used, and then writes the corresponding value from the *input* side of the lookup table to a single-band destination image.

Note – Since the values in the source image don't actually match values in the output side of the table, each source-image value is paired with the value in the table closest to it.

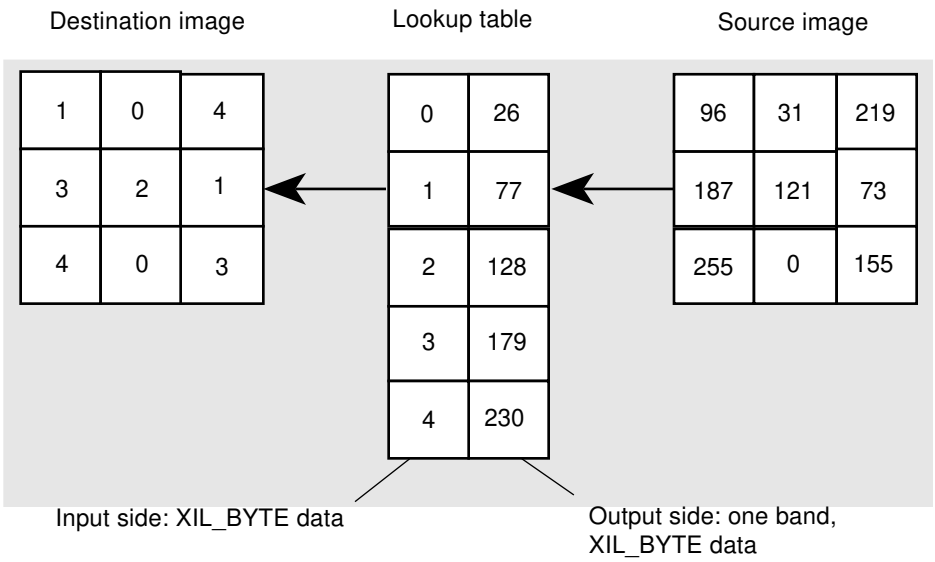

Figure 7-2 Dithering an Image

The purpose of this dithering operation is to produce an image that when mapped forward through the lookup table will produce an image as similar as possible to the original source image. Another way to state this is that the dithering operation produces an image that when displayed using the lookup table as its colormap will look as much like the original as possible.

Such dithering operations have many applications, but a couple of them are by far the most common. One common use of dithering is to convert a true-color (3-band `XIL_BYTE`) image to a pseudocolor (1-band `XIL_BYTE`) image. Another common use is to convert a grayscale (1-band `XIL_BYTE`) image to a monochrome (1-band `XIL_BIT`) image.

Methods of Dithering

This section contains information about the three functions the XIL library provides for performing dithering operations: `xil_nearest_color()`, `xil_error_diffusion()`, and `xil_ordered_dither()`.

xil_nearest_color()

The function `xil_nearest_color()` is the simplest of the dithering functions in that its algorithm for performing the inverse lookup described above is the most straightforward. This algorithm includes no provision for eliminating unwanted contours in the dithered image.

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

```
void xil_nearest_color(XilImage src, XilImage dst,
    XilLookup lookup);
```

The parameters `src` and `dst` are handles to the source image—the image to be dithered—and the destination image. The parameter `lookup` is a handle to the lookup table through which the source image will be passed to produce the destination. There are actually two types of lookup tables that can be referred to here. One is the generic lookup table that is used for lookup operations, and the other is a special type of lookup table called a *colorcube*. The basic difference between the two types of lookup tables is this: when you create a generic lookup table, you specify the dimensions of the table and the data to be stored in the table; when you create a colorcube, you specify the dimensions of the colorcube, and the function you use to create the colorcube fills in the data in the table for you. The sections below provide additional information about these two types of lookup tables.

Lookup Tables

You can create a generic lookup table in one of two ways: using the function `xil_lookup_create()` or using the function `xil_choose_colormap()`. In either case, `xil_nearest_color()` pairs the values in your source image with values on the output side of the lookup table by searching for nearest matches. This means that for each pixel in your source image, `xil_nearest_color()` must examine each entry in the lookup table. This is a time-consuming process. However, generic lookup tables do give you the best control over the contents of your lookup table and, therefore, over the quality of the dithered image.

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

```
XilLookup xil_lookup_create(XilSystemState state,
    XilDataType input_datatype, XilDataType output_datatype,
    unsigned int output_nbands, unsigned int num_entries,
    short first_entry_offset, void *table_data);
```

Here's an example of a time when you might want to use this function to create your lookup table. You have a true-color image that you want to dither to a single-band `XIL_BYTE` image, and you want to be able to display the dithered image using the currently installed X colormap. The code you use to create this lookup table might look like this:

```
/* Read the color values stored in the X colormap and write them
   to an array of Xil_unsigned8. The values should be written
   in the following order: b0, g0, r0, b1, g1, r1, and so on.
   Let's say the array is called table_data */

xil_lookup_create(state, XIL_BYTE, XIL_BYTE, 3, 256, 0,
    table_data);
```

Using this table for your dither operation enables you to produce a dithered image that contains the maximum number of unique values (256) and to display the image without having to write anything to the X colormap. You won't have to worry about colormap flashing. The disadvantage of this method is that there may be few or no exact matches between the RGB values in your source image and the RGB values on the output side of the lookup table. Clearly, this can affect how accurately the dithered image, when displayed, will reflect the original source image.

The other function you might use to create a lookup table for dithering the true-color image mentioned above to a pseudocolor image is `xil_choose_colormap()`. This function returns the best lookup table of a particular size to use in dithering the image. The best lookup table is defined to be the one that contains as many as possible of the most frequently used colors in the source image.

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

```
XilLookup xil_choose_colormap(XilImage src, unsigned int size);
```

When you use this function to create your lookup table, there are two basic approaches you might take to dithering your true-color image. One approach is to use `xil_choose_colormap()` to create a lookup table that contains 256 entries. This strategy will produce a dithered image that looks very much like the original source image when displayed because the 256 most frequently used colors in the original will be reproduced exactly. However, you also have to get 256 specific colors into a hardware colormap to display your image. This will lead to colormap flashing on most displays.

A second approach is to use `xil_choose_colormap()` to create a lookup table with fewer than 256 entries, say 240 entries. You will still have most of the colors you need to display your image, and you can write those colors to your hardware colormap (if you only have one) without overwriting the first 16 entries in that colormap. This should prevent your application from contending with the OpenWindows™ tools for colors.

Note – `xil_choose_colormap()` accepts only 3-banded `XIL_BYTE` source images.

Colorcubes

You create a colorcube using the function `xil_colorcube_create()`. The prototype for this function is shown below.

```
XilLookup xil_colorcube_create(XilSystemState state,
    XilDataType input_type, XilDataType output_type,
    unsigned int nbands, short offset, int multipliers[],
    unsigned int dimensions[]);
```

Note that the parameters to this function are similar to those for `xil_lookup_create()`. However, you don't specify a number of entries in the lookup table or the data to be loaded into the table. Instead you provide arrays called `dimensions` and `multipliers`. The number of elements in both of these arrays must equal the number of bands in the image being dithered.

To understand the roles of `dimensions` and `multipliers`, consider the situation where you want to dither an `XIL_BYTE` RGB image to a 1-band `XIL_BYTE` image. To handle this case, you might declare and initialize `dimensions` and `multipliers` as follows:

```
unsigned int dimensions[] = {4, 9, 6};
int multipliers[] = {1, 4, 36};
```

These values would lead to the creation of a colorcube that would dither blue values in the source image to one of 4 blue levels, green values to one of 9 green levels, and red values to one of 6 red levels. You could picture this colorcube as a cube with dimensions of 4, 9, and 6, but it's probably more helpful to think of it as a lookup table with three bands on the output side. Figure 7-3 shows what the first 22 elements of the 216-element table would look like.

Values to be written to indexed-color image.

	Blue values	Green values	Red values
0	0	0	0
1	85	0	0
2	170	0	0
3	255	0	0
4	0	32	0
5	85	32	0
6	170	32	0
7	255	32	0
8	0	64	0
9	85	64	0
10	170	64	0
11	255	64	0
12	0	96	0
13	85	96	0
14	170	96	0
15	255	96	0
16	0	128	0
17	85	128	0
18	170	128	0
19	255	128	0
20	0	159	0
21	85	159	0

Figure 7-3 Colorcube for Dithering a True-Color Image to a Pseudocolor Image

This illustration should help clarify the significance of the three elements of `multipliers`. The element `multipliers[0]` indicates the distance between changes in blue levels on the output side of the table, `multipliers[1]` indicates the distance between changes in green levels, and `multipliers[2]` indicates the distance between changes in red levels. That is, the blue level changes with each entry, the green level changes every fourth entry, and the red level changes every thirty-sixth entry.

Note – The elements of `multipliers` can be negative numbers. Negative numbers indicate that the values in a color ramp should appear in decreasing as opposed to increasing order.

The values in the table are calculated by `xil_colorcube_create()` based on the dimensions and multipliers you supply. This means that when you dither an image using a colorcube, the dither function does not need to search the output side of a lookup table to find a set of values. Instead, it can calculate the value to be written to the dithered image using the values stored in the `dimensions` and `multipliers` arrays. As a result, dithering operations that use a colorcube are much faster than those that use a generic lookup table. What you give up is control over the exact contents of the colorcube.

One other note about colorcubes. Although they are called color*cubes*, colorcubes need not have three dimensions. As mentioned earlier, a colorcube has a number of dimensions equal to the number of bands in the image to be dithered. Therefore, to create a colorcube suitable for dithering a grayscale image to a monochrome image, you might define dimensions and multipliers as follows.

```
unsigned int dimensions[] = {2};
int multipliers[] = {-1};
```

Note – The XIL library creates two colorcubes when you initialize the library. One of these has the dimensions 4:9:6 and is useful for dithering RGB images to 216 colors. The other has the dimensions 8:5:5 and is useful for dithering YC_bC_r images to 200 colors. To get a handle to one of these colorcubes, use the function `xil_lookup_get_by_name()`.

In addition to `xil_colorcube_create()`, the XIL library contains the colorcube-related functions shown in Table 7-3.

Table 7-3 Functions for Managing Colorcubes

Function Name	What the Function Does
`xil_lookup_get_colorcube`	Determines whether a lookup table is a colorcube or a generic lookup table
`xil_lookup_get_colorcube_info`	Determines whether a lookup table is a colorcube and, if it is, returns the dimensions and multipliers used to create the colorcube and the colorcube's origin

xil_error_diffusion()

The function `xil_error_diffusion()` is similar to `xil_nearest_color()`, but in addition to performing an inverse lookup, it uses a process called *error diffusion* to deal with any difference between a source-image pixel value and a value on the output side of a lookup table with which that pixel value is matched. The difference (or error) is distributed to the source-image pixels immediately to the right of and below the last pixel processed.

The function prototype for `xil_error_diffusion()` is shown below:

```
void xil_error_diffusion(XilImage src, XilImage dst,
    XilLookup lookup, XilKernel distribution);
```

The first three parameters are the same as the parameters to `xil_nearest_color()`. The parameter `src` is a handle to your source image, and `dst` is a handle to your destination image. The parameter `lookup` can be either a generic lookup table or a special type of lookup table called a colorcube. For information about these lookup tables, see the sections "Lookup Tables" on page 108 and "Colorcubes" on page 110. The final parameter, `distribution`, is an error-distribution kernel.

Note – XIL kernels (data structures of type `XilKernel`) are used primarily for convolution operations, which are discussed in the section "Filtering an Image" on page 187. The utility functions that affect kernels are also discussed in that section.

You create the error-distribution kernel using the function `xil_kernel_create()`, whose function prototype is shown below.

```
XilKernel xil_kernel_create(XilSystemState system_state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, float *data);
```

The parameters `width` and `height` determine the size of the kernel, `keyx` and `keyy` determine the kernel's *origin*, and `data` is a pointer to the floating-point values to be stored in the kernel. For example, the code below creates a 3-by-3 kernel with an origin of 1,1.

```
XilKernel kernel;
float kernel_data[] = {
    0/16.0, 0/16.0, 0/16.0,
    0/16.0, 0/16.0, 7/16.0,
    3/16.0, 5/16.0, 1/16.0
};

kernel = xil_kernel_create(state, 3, 3, 1, 1, kernel_data);
```

Note – The library provides a special shorthand method of creating the kernel shown above because it is used so commonly in error-diffusion operations. This method is to call the function `xil_kernel_get_by_name()` using a name of "`floyd-steinberg`".

Figure 7-4 shows what this kernel looks like.

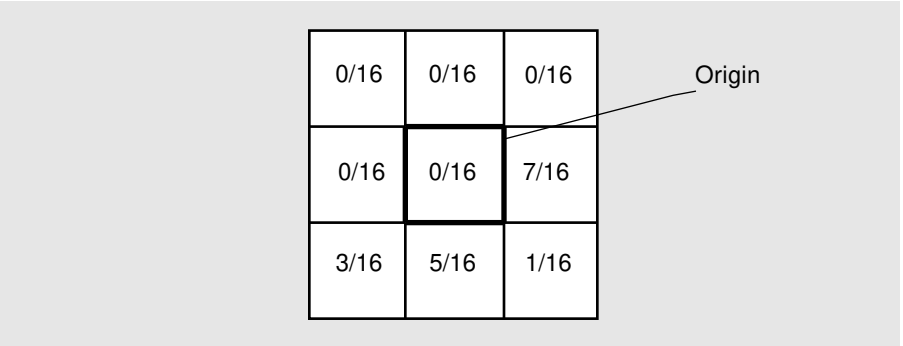

Figure 7-4 Error-Distribution Kernel

Note – The value of the kernel's origin and of the elements above and to the left of the origin must be 0 when you're calling `xil_error_diffusion()`.

Here's how `xil_error_diffusion()` uses the kernel. Say that you're dithering a grayscale image with values in the range 0 to 255 to a grayscale image with values in the range 0 to 4. This situation is shown in Figure 7-5.

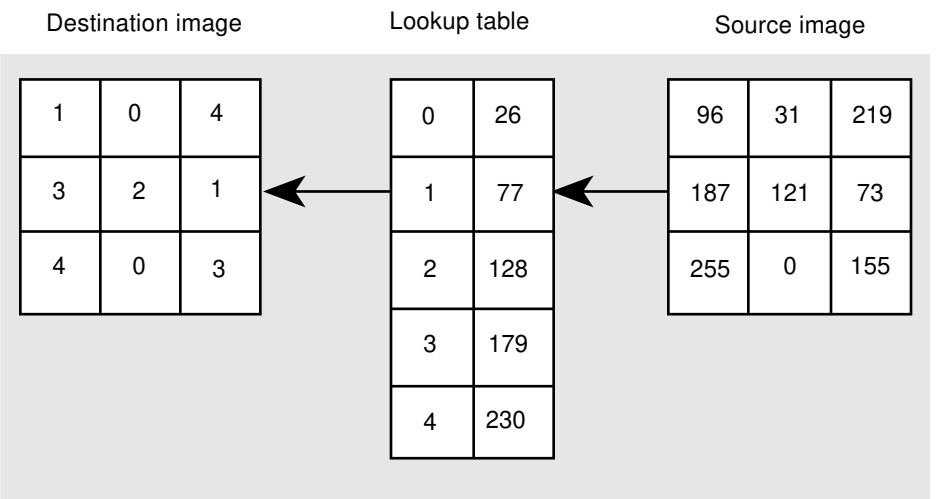

Figure 7-5 Using `xil_error_diffusion()` to Dither an Image

When the pixel in the middle of the top scanline of the source image (31) is passed through the lookup table, it is matched with a 26, so a 0 is written to the destination image. When the destination is displayed using the lookup table as its colormap, this pixel will have the value 26. Thus, in the dither-and-reconstruction process, there will be an error of 5 (31 - 26). To counteract the loss of data inherent in this process, `xil_error_diffusion()` distributes this error before passing the next pixel (219) through the lookup table. The way in which the error is distributed depends on the kernel created earlier. See Figure 7-6 for an example.

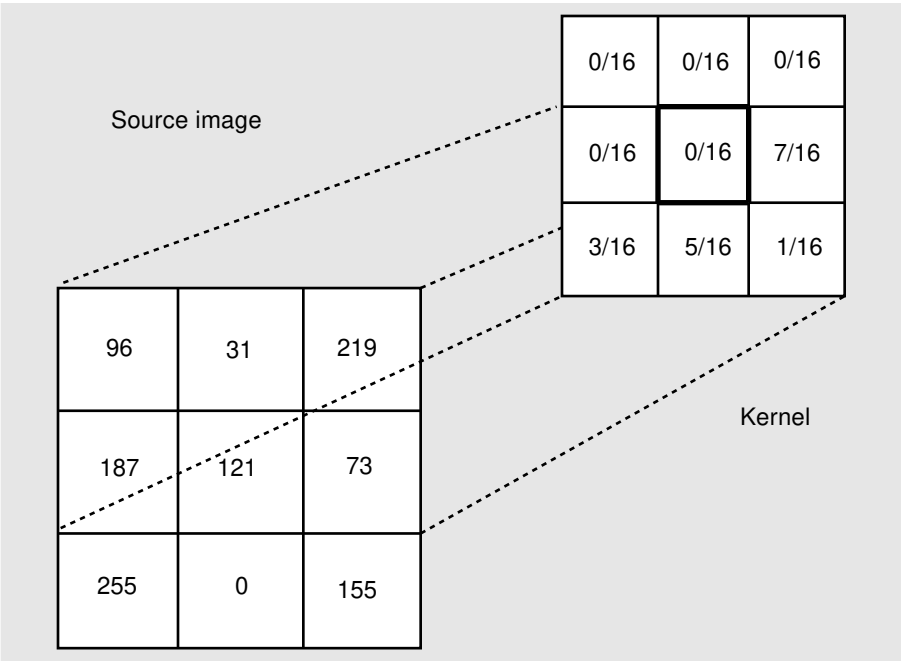

Figure 7-6 Error Diffusion

The kernel is laid on top of the source image so that its origin aligns with the last pixel to be passed through the lookup table. Before the next pixel is passed through, the following processing will take place:

- The pixel at 0,2 will be set to 219 + (5 * (7 / 16.0))
- The pixel at 1,0 will be set to 187 + (5 * (3 / 16.0))
- The pixel at 1,1 will be set to 121 + (5 * (5 / 16.0))
- The pixel at 1,2 will be set to 73 + (5 * (1 / 16.0))

This processing is fairly time-consuming, but can greatly reduce contouring in the dithered image.

xil_ordered_dither()

The XIL library's third dithering function is `xil_ordered_dither()`. Like `xil_nearest_color()` and `xil_error_diffusion()`, `xil_ordered_dither()` processes an image by performing an inverse lookup. However, two characteristics distinguish `xil_ordered_dither()` from the other dithering functions. One is that the lookup table the function uses to do its job must be a colorcube; it cannot be a generic lookup table. (For information about the differences between colorcubes and generic lookup tables, see the sections "Lookup Tables" on page 108 and "Colorcubes" on page 110.) Second, `xil_ordered_dither()` makes use of an XIL data structure called a dither mask to help eliminate contouring in the dithered image.

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

```
void xil_ordered_dither(XilImage src, XilImage dst,
    XilLookup lookup, XilDitherMask mask);
```

The parameters `src` and `dst` are handles to the source and destination images, and `lookup` is the lookup table to be used for the dither operation. As mentioned above, this lookup table must be a colorcube; that is, it must have been created with the function `xil_colorcube_create()`—or `xil_lookup_get_by_name()`. The final parameter, `mask`, is the dither mask, a data structure of type `XilDitherMask`.

You create the dither mask using the function `xil_dithermask_create()`, whose prototype is shown below.

```
XilDitherMask xil_dithermask_create(XilSystemState state,
        unsigned int width, unsigned int height,
        unsigned int nbands, float *data);
```

The parameters `width` and `height` define the width and height of the dither mask in pixels. These values are user defined. The next parameter, `nbands`, determines the number of bands in the dither mask. This number must match

the number of bands in the image being dithered. Finally, `data` is a pointer to the data to be stored in the dither mask. All the values in the mask must fall in the range 0.0 to 1.0. For example, the code below creates a 1-band, 4-by-4 dither mask.

```
XilDitherMask mask;
float mask_data[] = {
     0/16.0, 8/16.0, 2/16.0,10/16.0,
    12/16.0, 4/16.0,14/16.0, 6/16.0,
     3/16.0,11/16.0, 1/16.0, 9/16.0,
    15/16.0, 7/16.0,13/16.0, 5/16.0
}

mask = xil_dithermask_create(state, 4, 4, 1, mask_data);
```

Note – When you initialize the XIL library, four standard dither masks are created: 1- and 3-band 4-by-4 masks and 1- and 3-band 8-by-8 masks. To get a handle to one of these masks, you use the function `xil_dithermask_get_by_name()`.

Here's how the mask is used. Assume that you're dithering a grayscale image with values in the range 0 to 255 and that the destination is to contain values in the range 0 to 15. First, think of the 4-by-4 dither mask as having been replicated over the entire source image, as depicted in Figure 7-7.

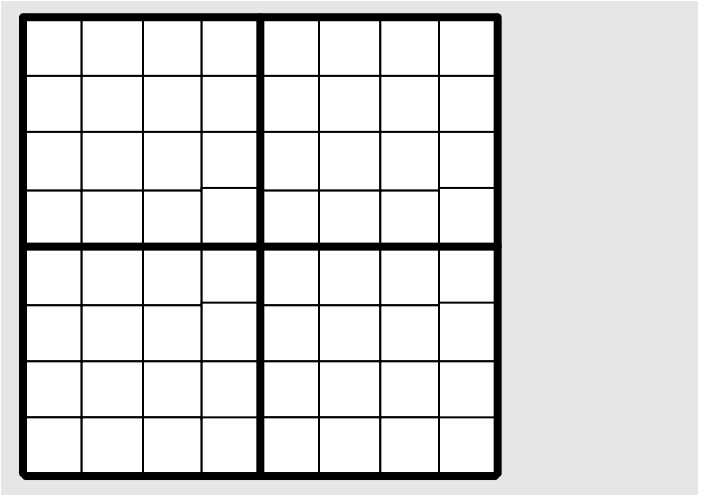

Figure 7-7 Dither Mask Replicated over a Source Image

At this point, each pixel in each 4-by-4 block of the image has been associated with (in this case) a unique dither-mask value. The `xil_ordered_dither()` function then performs the following steps for each pixel in the image:

1. **Passes the value through the lookup table.**
   In this step, the function is actually dividing values in the source image by 17 (the number of values that can be represented in 8 bits divided by the length of the colorcube minus one). That is, if no further processing were to take place, a 200 in the source image would map to an 11 since 200 divided by 17 equals 11.76.

2. **Considers the dither-mask value associated with the source-image pixel value.**
   The dither function now compares the fractional part of the quotient from the division operation with the source pixel's dither-mask value. If the fractional part is greater than the dither-mask value, the dividend shown above (11) is incremented by one, so a 12 is written to the destination image. Otherwise, an 11 is written to the image.

When you use the dither mask in this way, you're essentially performing averaging. This averaging helps prevent undesirable contours from appearing in the dithered image.

Table 7-4 below lists the XIL utility functions that affect dither masks and indicates what these functions do.

Table 7-4 Utility Functions for Dither Masks

Function Name	What It Does
`xil_dithermask_destroy`	Deallocates the memory used by a dither mask
`xil_dithermask_get_height`	Returns the height of a dither mask in pixels
`xil_dithermask_get_width`	Returns the width of a dither mask in pixels
`xil_dithermask_get_nbands`	Returns the number of bands in a dither mask
`xil_dithermask_create_copy`	Returns a copy of a dither mask
`xil_dithermask_set_name`	Sets the name of a dither mask
`xil_dithermask_get_name`	Returns a copy of a dither mask's name

When to Use Each Dithering Function

When you're trying to decide on the best way to perform a dither operation, it's helpful to think of the matrix shown in Table 7-5:

Table 7-5 Review of Dithering Operations

	Generic Lookup	Colorcube
xil_nearest_color()	✔	✔
xil_error_diffusion()	✔	✔
xil_ordered_dither()		✔

If you look at the right column—labeled "Colorcube"—you'll see that each of the dithering functions can perform its inverse lookup using a colorcube. If you compare the functions used with a colorcube, you'll see that choosing one over another involves a trade-off between speed and quality. The function `xil_nearest_color()` is the fastest, but produces the poorest quality image, and `xil_error_diffusion()` is the slowest, but produces the best quality image. The function `xil_ordered_dither()` is somewhat faster than `xil_error_diffusion()`, but does not produce quite as nice an image.

Similarly, if you compare operations that use generic lookup tables, the trade-off is one between speed and quality. The guidelines mentioned in the preceding paragraph apply, except that `xil_ordered_dither()` cannot work with a generic lookup table.

Now, if you read the table from left to right, you can compare operations that use generic lookup tables with corresponding operations that use colorcubes. Operations that use colorcubes are much faster than their counterparts because operations that use colorcubes do not need to search a lookup table. On the other hand, when you use a colorcube for dithering, you lose some of your control over the quality of the dithered image, because you cannot specify the exact values in the colorcube. You also lose the ability to use the colors in the currently installed colormap, again because you don't specify the values to be written to the lookup table.

Color-Space Conversion

The XIL library enables you to convert an image from any one to any other of the following color spaces:

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

The procedure for converting an image from one color space to another is outlined below.

1. **Set the color-space attribute of your source image.**
   When you create an XIL image, it does not have a color space associated with it. To associate one with it, you must call the function `xil_colorspace_get_by_name()` to get the appropriate color-space data structure; then, you call `xil_set_colorspace()` to actually set the attribute.

The code fragment below shows a program getting a data structure of type `XilColorspace` that describes a linear RGB color space.

```
XilColorspace colorspace;

colorspace = xil_colorspace_get_by_name(state, "rgblinear");
```

The parameter `state` is a handle to the system state, and `rgblinear` is a string identifying the color space. The strings you use to identify the various XIL color spaces are listed in Table 7-6.

Table 7-6 Strings Used to Specify Color Spaces

Color Space	String
CCIR Rec. 709 RGB	`rgb709`
Linear version of CCIR Rec. 709 RGB	`rgblinear`
CCIR Rec. 709 YC_bC_r	`ycc709`
Luminance-only space derived from CCIR Rec. 709 YC_bC_r	`y709`
Linear version of `y709`	`ylinear`
CCIR Rec. 601 YC_bC_r	`ycc601`
Luminance-only space derived from CCIR Rec. 601 YC_bC_r	`y601`
YC_bC_r color space defined by Kodak for Photo CD	`photoycc`
Linear CMY	`cmy`
Linear CMYK	`cmyk`

After you've gotten this color-space data structure, you set an image's color-space attribute using the function `xil_set_colorspace()`.

```
XilImage image;
XilColorspace colorspace;

xil_set_colorspace(image, colorspace);
```

2. **Create a destination image, and assign it the color space to which you want to convert your source image.**
   The destination image must have the same width, height, and data type as the source image, and it must have a number of bands appropriate to the color space you will assign it. That is, if you want to assign the destination image the color space `ycc709`, it must have three bands. You assign a color space to the destination using the same method you used to assign a color space to the source image.

Note – It is possible to perform an in-place color conversion if the destination image is a child of the source and has the same width and height as the source. This procedure is not generally recommended, however, because the color-conversion operation will overwrite some or all of the data in the source image.

---

3. **Perform the color conversion.**
   You perform the color conversion by calling the function `xil_color_convert()`. This function takes handles to your source and destination images as its two parameters.

Note – Regions of interest are ignored when you perform a color conversion.

There are many applications for this type of color conversion. For example, you may want to:

- Convert an image from a YC_bC_r color space to an RGB color space in order to display the image
- Convert an image from an RGB color space to a YC_bC_r color space to prepare it as input to the JPEG or CellB compressor
- Convert an image to the CMY or CMYK color space to prepare it to be printed on a subtractive color printer
- Convert an RGB or YC_bC_r image to the `y709`, `ylinear`, or `y601` color space as a way of converting a 24-bit image to an 8-bit grayscale image
- Convert an `XIL_BYTE` image from a linear color space to a gamma-corrected color space to prevent contouring in low-intensity regions of the image

Black Generation

The XIL library also enables you to perform undercolor removal and black generation on CMYK images—images to be printed on a four-color printing system. Normally, the process of undercolor removal and black generation goes something like this. For each pixel in the image:

- Black (K) equals the minimum of C, M, and Y
- C = C - K
- M = M - K
- Y = Y - K

However, the XIL function that performs undercolor removal and black generation gives you more control over how much cyan, magenta, and yellow will be removed from your image and over how much black will be added to it. This function is called `xil_black_generation()`, and its function prototype is shown below.

```
void xil_black_generation(XilImage src, XilImage dst,
    float black, float undercolor);
```

The parameters `src` and `dst` are handles to your source and destination images. Both images must be CMYK images. Therefore, if you want to process a CMY image, you must first convert it to a CMYK image. For instructions on how to do this, see the section “Color-Space Conversion” on page 122. In-place operations are allowed.

The parameters `black` and `undercolor` are used as follows. For each pixel in your source image:

- K = (`black` * minimum of C, M, and Y)
- C = C - (`undercolor` * minimum of C, M, and Y)
- M = M - (`undercolor` * minimum of C, M, and Y)
- Y = Y - (`undercolor` * minimum of C, M, and Y)

Thus, if both `black` and `undercolor` are set to 1.0, you'll get standard undercolor removal. You'll frequently achieve better results, though, if the percentage of black you add to your image is slightly higher than the percentage of C, M, and Y you remove from it. For example, you might try using the following call:

```
XilImage src, dst;

xil_black_generation(src, dst, 0.7, 0.5);
```

Note – Regions of interest are ignored when you perform undercolor removal.