Basic XIL Program 2

To introduce programming with the XIL library, this chapter takes a look at a simple XIL program called `example1`. This program reads an 8-bit grayscale image from a file and displays it in an X window. The program also enables you to take the one's complement of the image displayed in the window, which produces a negative of the image.

The source files for this program—`example1.c`, `fileio.c`, and `window.c`—can be found in the directory `$XILHOME/examples/example1`. This directory also contains an image file that the program should use as input and a makefile you can use to build the program.

After briefly discussing how to build and run `example1`, this chapter walks you through the program's code, pointing out some of the features of the program that will be part of most or all XIL programs. The topics covered are:

- Including header files
- Initializing the library
- Acquiring an input image
- Creating an output image
- Processing the image
- Closing the library

The chapter also talks briefly about building XIL programs.

2

Running the Program

If you want to run `example1` before reading the discussion of how it is written, follow these steps:

1. **Change your working directory to `$XILHOME/examples/example1`. (You should have set the environment variable `$XILHOME` when you installed the XIL software.)**
2. **Build the program by typing `make`.**
3. **Execute the program from the directory in which you built it by typing**

   ```
   % example1 brainscan.header
   ```

   At this point, you will see an X-ray of a brain displayed in an X window.
4. **To take the one's complement of the image in the window, move your pointer into the program's window and press any key on your keyboard. A second keypress will restore the image to its original state (by causing a second bitwise negation to be performed).**
5. **Exit the program by moving your pointer into the program's window and clicking any mouse button.**

The following sections provide an explanation of the XIL-related code in `example1`'s source files.

Including Header Files

The first XIL-related item in `example1.c` is a preprocessor directive that includes the header file `xil.h`. All XIL programs must include this header file and, therefore, must include the line

```
#include <xil/xil.h>
```

This header file defines a number of symbolic constants and enumeration constants and contains function prototypes for all of the functions in the XIL library.

Initializing the Library

If you skip down to the `main` routine in `example1.c`, you'll note that after reading command-line arguments, the example calls the routine `xil_open()`.

```
XilSystemState state;

state = xil_open();
if (state == NULL) {
    /* XIL's error handler will print an error msg on stderr */
    exit(1);
}
```

All XIL programs must call this function to initialize the library before using other functions from the XIL library.

As part of the initialization process, `xil_open()` returns a data structure of type `XilSystemState` called `state`. The library uses this data structure to keep track of information such as what XIL images (and other XIL data structures) have been created during the current session and what the program's error-handling routine is. You'll notice later in the program that `state` must be passed to any XIL function that creates an image.

Note – Since someone might want to use XIL routines and routines from an API built on top of the XIL library in the same program, it is possible to have multiple `XilSystemState` structures.

Acquiring an Input Image

All XIL applications process images in some way. As you will see later, these images can come from a variety of sources, and how you bring an image into an application depends on its source. The program `example1`, however, only considers the case in which a program gets its input, or *source*, image from a disk file. To read such a file, you must perform the series of steps outlined below:

1. Read the image file's header (or take whatever steps are appropriate) to determine the image's width, height, number of bands, and data type. You will need this information when you perform the next step.

2. Create an XIL image of the correct dimensions and data type. This image will serve as an empty container in which you can place the pixel values from your file.

3. *Export* the XIL image to obtain information about how the image's pixel values will be stored in memory.

   At all times, an image is either in *XIL space* or *application space*. While it is in XIL space (or *imported*), you can operate on the image only by using XIL functions; you do not know the address of the image or its pixel values. On the other hand, when the image is exported, you can get a pointer to the storage allocated to hold the image's pixel values and write data in that memory (or modify existing data). You can also change the pointer to the storage.

4. Read the pixel values from your file, and write them to the storage you received a pointer to in the preceding step. *It may be necessary to convert the format of this data as you copy it.* The XIL library supports images of several data types, and for each type, it specifies a memory format. These memory formats are discussed in the section "Exporting XIL Images" on page 36.

5. Import the XIL image so that all subsequent operations on the image must be performed using XIL functions.

The sections below discuss each of these steps in more detail and show how the steps are performed in `example1`. The code fragments shown are taken from the source file `fileio.c`.

Step 1: Reading the File's Header

As mentioned above, you need to read the header of your image file to determine the following image attributes:

- Width of the image in pixels
- Height of the image in pixels
- Number of bands, or channels, in the image
- Data type of the image

You need this information to create an XIL image in which to store the file's pixel values.

How you get this information from your file's header depends on the format of that header. The function `load_file()` shows you how this task was handled in one case. The header for the image file supplied for use with the example is shown below.

```
512 512 8 1
brainscan.data
```

The code the example uses to get the necessary image attributes from the header looks like this.

```
XilImage
load_file (XilSystemState state, char *filename)
{
   unsigned int width, height, nbands, bits;
   XilDataType datatype;
   FILE *in_file;
   char datafile[256];

   /* Open the header file */
   if ((in_file = fopen(filename, "r")) == NULL) {
      fprintf(stderr,"Could not open %s\n", filename);
      exit(1);
   }

   /* Get header information */
   fscanf(in_file, "%d%d%d%d", &width, &height, &bits, &nbands);
   fscanf(in_file, "%s", datafile);
   fclose(in_file);

   if ((bits == 8) && (nbands == 1)) {
      datatype = XIL_BYTE;
   }
   else {
      fprintf(stderr,"Program requires a 1 band, 8 bit image\n");
      exit(1);
   }
```

The routine `load_file()` is passed the name of the image file, opens the file, and reads the four integers in the first line of the header. These integers represent the width of the image, its height, the number of bits used to represent one band of one pixel, and the number of bands in the image.

You can pass the width, height, and number-of-bands arguments directly when you call the routine that creates your XIL image (in step 2). However, before making that call, you must use the number-of-bits value to set a variable of the enumeration type `XilDataType`. You set this variable to one of the following constants: `XIL_BIT`, `XIL_BYTE`, or `XIL_SHORT`. These enumeration constants correspond to the three types of data the XIL library supports: 1-bit data, 8-bit unsigned data, and 16-bit signed data. The variable is set in the line

```
datatype = XIL_BYTE;
```

Since this program expects a single-band, 8-bit image as input, an error occurs if `bits` does not equal 8.

Note – The other item read from the file is the name of the file that contains the actual pixel values. This file is opened and read in step 4.

Step 2: Creating an XIL Image

Once you have read the appropriate values from your image file's header, you can create your XIL image by calling `xil_create()`.

```
XilImage image;

/* Create the image to read the data into */
image = xil_create(state, width, height, nbands, datatype);
if (image == NULL) {
   /* XIL's error handler will print an error msg to stderr */
   exit(1);
}
```

With the exception of `state`, the parameters being passed to `xil_create()` are the variables set in the preceding step. The `state` parameter is the data structure returned by the call to `xil_open()` earlier in the program.

The return value of `xil_create()` is a handle to an XIL image (a data structure of type `XilImage`). This is the image you export in the next step.

Step 3: Exporting the Image

You now need to export the image you just created and to get some information about how the image's pixel values will be stored in memory. The example uses the following code to accomplish these tasks:

```
XilImage image;
XilMemoryStorage storage;
Xil_boolean status;

/* Get the pointer to the image data */
if (xil_export(image) == XIL_FAILURE) {
    /* XIL's error handler will print an error msg to stderr */
    exit(1);
}
status = xil_get_memory_storage(image, &storage);
if (status == FALSE) {
   /* XIL's error handler will print an error msg to stderr */
   exit(1);
}
```

The call to `xil_export()`, whose only parameter is a handle to the image, marks the image as exported. By exporting the image, the application gains access to information about the image that enables it to modify the image directly. For example, the application can get a pointer to the location at which the image's pixel values will be stored. For further information about exported images, see the section "Exporting XIL Images" on page 36.

The example program exports the image so that it can find out the address at which the image's pixel values will be stored, the number of bytes between pixels, and the number of bytes between scanlines. To get this information, the example calls the function `xil_get_memory_storage()`.

This function takes as its parameters a handle to your XIL image and the address of a union of structures named `storage`. This union is of the defined type `XilMemoryStorage`.

```
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;
```

Note – `Xil_unsigned8` is a data-type name created in `xil.h`. You should use this name (instead of `unsigned char`) whenever you need to get the size of a single pixel value in an 8-bit XIL image. There is a corresponding type name, `Xil_signed16`, for 16-bit data.

Thus, the example can refer to the address of the pixel values for the exported image by writing `storage.byte.data`. Similarly, it can specify the number of bytes between pixels and the number of bytes between scanlines by writing `storage.byte.pixel_stride` and `storage.byte.scanline_stride` respectively.

Note – Any time you plan to use an image's data, you must export the image by calling `xil_export()`, then get pointers to its data with `xil_get_memory_storage()`. These pointers to the data are valid only until the image is imported. For more information, see "Exporting XIL Images" on page 36.

Step 4: Copying Data from a File to Your XIL Image

The example now knows the location of the pixel values it wants to read (they're stored in `datafile`) and the address (`storage.byte.data`) at which to write those values. The principal remaining question is whether the format of the pixel data in the file matches the memory format for the XIL image. Since this example reads a single-band (grayscale) image, the formats will match. The XIL image containing 8-bit data requires that data be stored in a

pixel-sequential format, and the values in the file are stored in a pixel-sequential format. Given this situation, the example can copy the file data to the proper location in memory using the code shown below:

```
unsigned int width, height, nbands;
XilMemoryStorage storage;
FILE *in_file;
int size;

/* Open the data file */
if ((in_file = fopen(datafile, "r")) == NULL) {
   fprintf(stderr, "Could not open %s\n", datafile);
   exit(1);
}

/* Load the image data */
if ((storage.byte.pixel_stride == nbands) &&
       (storage.byte.scanline_stride == nbands * width)) {
   size = nbands * width * height;
   if (fread((char *)storage.byte.data, sizeof(Xil_unsigned8),
          size, in_file) != size) {
      fprintf(stderr, "Error reading data in %s\n", datafile);
      exit(1);
   }
} else {
   int h, w, n;
   Xil_unsigned8 *scanline = storage.byte.data;
   for ( h = 0; h < height; h++) {
      Xil_unsigned8 *row = scanline;
      for (w = 0; w < width; w++) {
         Xil_unsigned8 *pixel = row;
         for (n = 0; n < nbands; n++) {
            *pixel++ = fgetc(in_file);
         }
         row += storage.byte.pixel_stride;
      }
      scanline += storage.byte.scanline_stride;
   }
}
```

Note that if the pixel stride equals the number of bands in the image (that is, there are no unused bytes between pixels in the XIL image), the data in the file `datafile` can be copied to the image in a block. Otherwise, the example must

use the values of `storage.byte.pixel_stride` and `storage.byte.scanline_stride` to determine where to write each pixel value it reads from the file.

What if the example were reading an RGB image from a file and the 8-bit RGB values in the file were stored in band-sequential format? In that case, the example would need not only to copy the data from the file to an XIL image, but to convert the format of that data as well. See the example code below.

Note – This code assumes the file contains blue values, followed by green values, followed by red values. Because this image is being read from a file, this is a valid assumption. However, you can never make this assumption about an `XilImage` in XIL memory space; in memory, an `XilImage`'s format is arbitrary and must be accessed only through `xil_get_memory_storage()`.

```
unsigned int width, height, nbands;
XilMemoryStorage storage;
FILE *infile;
int i, h, w;

/* Load the image data */
for (i = 0; i < nbands; i++) {
    Xil_unsigned8 *scanline = storage.byte.data + i;
    for (h = 0; h < height; h++) {
        Xil_unsigned8 *pixel = scanline;
        for (w = 0; w < width; w++) {
            *pixel = getc(in_file);
            pixel += storage.byte.pixel_stride;
        }
        scanline += storage.byte.scanline_stride;
    }
}
```

Step 5: Importing the Image

Once you have read image data from a file into an XIL image, you should import the image so that all further operations on it must be performed using XIL functions. You import the image with a call to `xil_import()`:

```
XilImage image;

xil_import(image, 1);
```

Note – Importing an image is the opposite of exporting it. When you import an image, any pointer to the image's pixel values you had while the image was exported becomes invalid, so you can no longer modify the image directly. You must make all modifications using XIL functions. To access the data again, you must export the image again, and make another call to `xil_get_memory_storage()` to get new pointers to the data. For more information, see "Exporting XIL Images" on page 36.

The argument `image` is a handle to the image being imported, and the `1` is a change flag. This flag indicates that the image was modified while it was exported; in this case, it was filled with pixel values.

Creating an Output Image

Because `example1` displays its output, its output image must be a special type of image called a *display image*. This type of image is written to a screen's frame buffer and is displayed in an X window.

To create this display image, the example program:

1. Creates an X window the size of the image it wants to display.
2. Uses the function `xil_create_from_window()` to create a display image from the X window discussed in step 1. This routine turns the X window into an XIL image that can be used as a destination for an XIL operation.

The sections below discuss these steps in more detail.

Creating an X Window

Most of the Xlib code in this example appears in the function `open_window()`, which is defined in the source file `window.c`. This function:

- Establishes a connection with the X server
- Finds the best X visual to use when displaying a grayscale image on a particular display
- Creates the X colormap that will be used when the image is displayed
- Creates the X window in which the image will be displayed

To create the X window, the function calls `XCreateWindow()`.

```
Display *xdisplay;
Window xwindow;
XVisualInfo *vinfo;

xwindow = XCreateWindow(xdisplay, DefaultRootWindow(xdisplay),
    20, 20, width, height, 0, vinfo->depth, InputOutput,
    vinfo->visual, CWColormap, &setwinattr);
if (xwindow == NULL) {
   fprintf(stderr, "Unable to create window\n");
   return (0);
}
```

The return value of `XCreateWindow()` is an X window ID, which is used in the creation of the XIL display image.

Creating a Display Image

Before you can use an XIL function to display an image, you must create a display image. As mentioned earlier, to do this, you call the XIL function `xil_create_from_window()`. This routine turns an X window into a special type of XIL image so that the window can be used as the source or (as in this case) the destination image for an operation.

```
Display *xdisplay;
Window xwindow;
XilSystemState state;
XilImage display;

display = xil_create_from_window (state, xdisplay, xwindow);
```

This call returns a handle to the newly created display image. The parameters you pass to the routine are a handle to the system-state data structure returned by `xil_open()`, a pointer to a structure describing your display, and the ID of your X window.

At this point, the program is ready to do some image processing of the source image it read earlier.

Processing an Image

Once `example1` has obtained handles to a source and a destination image, it is ready to use XIL functions to process the source image and eventually write the processed image to the display. The functions used in this part of the program can be thought of as serving two purposes.

One purpose is to alter the source image so that it can be written to the destination image. For example, in the sample program, the source image contains 8-bit data, but the display image may just be 1 bit deep (if the screen on which the image is to be shown is monochrome). In this case, the source image must be converted to 1-bit data before it can be copied to the display image. This kind of conversion may require that you create an intermediate image.

The second purpose is to change the source image to alter its appearance in some way. For instance, you might darken the image or enhance the edges in the image. The `example1` program enables you to take the one's complement of its source image.

Making Source and Destination Images Compatible

The program `example1` handles only one type of source image: an 8-bit grayscale image. However, it handles several types of display images. It may write its output to:

- A 3-band `XIL_BYTE` display image whose X colormap is read only.
- A 1-band `XIL_BYTE` display image. There are actually two cases here because the image's X colormap (grayscale) may be read-write or read only.
- A 1-band `XIL_BIT` display image whose X colormap is read only.

In only one of the cases mentioned above can the source image be copied to the display image as is. The following sections briefly describe how each case must be handled.

Three-Band, Eight-Bit Destination—Read-Only Colormap

In this case, the source image contains one band, and the display image contains three. To solve this problem, `example1` creates a temporary 3-band `XIL_BYTE` image, called `retained_image`, and copies the source image to each band of the temporary image.

```
if (vinfo.class == TrueColor) {
    /* Copy the grayscale image into each of the bands */
    retained_image = xil_create(state, width, height, 3,
        XIL_BYTE);
    band0 = xil_create_child(retained_image, 0, 0, width, height,
        0, 1);
    band1 = xil_create_child(retained_image, 0, 0, width, height,
        1, 1);
    band2 = xil_create_child(retained_image, 0, 0, width, height,
        2, 1);
    xil_copy(image, band0);
    xil_copy(image, band1);
    xil_copy(image, band2);
}
```

Note that the example actually creates three band children of the temporary image and copies the source image to those children. Once this has been done, `retained_image` can be copied to the display image.

One-Band, Eight-Bit Destination—Read-Write Colormap

In this case, the program could write the source image directly to the display image. However, doing that would require that the program create and use a colormap with 256 entries. If such a colormap were installed, it might cause other X clients, including the window manager, to display their output in false colors.

To avoid this problem, the example creates an X colormap of 256 entries, but writes its grayscale ramp in the last 216 entries. The program then uses a call to `xil_rescale()` to ensure that all the values in the image to be displayed fall between 40 and 255.

One-Band, Eight-Bit Destination—Read-Only Colormap

In this case, the example can write the source image directly to the display image because the program's X colormap is a read-only grayscale ramp.

One-Band, One-Bit Destination—Read-Only Colormap

In this case, the example creates a single-bit XIL image (using `xil_create()`) and then uses the function `xil_error_diffusion()` to dither the 8-bit image to a 1-bit image.

Additional Processing

Besides processing the source image so that it can be written to the display, `example1` enables you to take the one's complement of the brain scan by pressing any key while your pointer is in the program's window. The code used to perform this operation is shown below:

```
Display *xdisplay;
XilImage retained_image, display;
unsigned int width, height;
XEvent event;

/* Refresh window as necessary till program terminates */
while (1) {
    XNextEvent(xdisplay, &event);
    if (event.xany.type == KeyPress) {
        xil_not(retained_image, retained_image);
        if (vinfo.class == GrayScale)
            xil_add_const(image, offset, image);
        xil_copy(retained_image, display);
    }
    else if (event.xany.type == Expose)
        xil_copy(retained_image, display);
```

If the example gets a keypress event, it calls `xil_not()` to take the one's complement of the source image. This single statement works for all cases except the case where the program is using a `GrayScale` X visual. In that case, the values in `retained_image` before the call to `xil_not()` fall in the range 40 to 255. After the call to `xil_not()`, they fall in the range 0 to 215. Therefore, before the one's complement is displayed, the example adds 40 to each value in the image using the function `xil_add_const()`.

You may want to experiment with removing from the example the code used to take the one's complement of the brain scan and replacing it with code that performs another image-processing operation. You might try any of the arithmetic and logical operations that work with one source image and a constant. These operations are discussed in Chapter 9, "Arithmetic, Relational, and Logical Functions." Or you might try replacing the logical negation with a geometric operation. The XIL geometric operations are the subject of Chapter 10, "Geometric Functions."

Closing the Library

Before exiting, all XIL programs should deallocate the memory associated with any XIL data structures that haven't already been destroyed. Because the system-state data structure keeps a list of all the data structures XIL has created (and not destroyed), you can destroy all of these data structures by calling `xil_close()`, which takes the system state as its only parameter.

```
XilSystemState state;

xil_close(state);
```

Note – If your program creates a display image and you do not destroy it with a call to `xil_destroy()`, you must close the XIL library (`xil_close()`) before you break your connection with the X server and display (`XCloseDisplay()`).

Building an XIL Program

The directory `$XILHOME/examples/example1`, which contains the source files for `example1`, also contains the Makefile you use to build the program. The Makefile's contents are shown below.

```
# You may get better performance with these 2.0.1 flags
# OPTFLAGS = -xcg89 -Wa,-cg92
CFLAGS = $(OPTFLAGS) -I$(XILHOME)/include -I$(OPENWINHOME)/include

LIBS = -L$(XILHOME)/lib -L$(OPENWINHOME)/lib -R/opt/SUNWits/Graphics-
sw/xil/lib:usr/openwin/libs

install := LIBS += -lxil -lX11 -ldl -ldga -lthread
debug_install := LIBS += -lxil -lX11 -ldl -ldga
debug_install := CFLAGS += -g

install: example1 brainscan.header brainscan.data
debug_install: example1 brainscan.header brainscan.data

debug: debug_install

example1: example1.o window.o fileio.o
        $(CC) -o example1 example1.o window.o fileio.o $(LIBS)

clean:
        rm -f a.out core *.o example1
...
```

The main thing to note here is the list of libraries being used to link the program. When you link an XIL program, you must use these libraries (with the possible exception of the X11 library) *in this order* in your makefile or on your command line.

Table 2-1 below provides a brief explanation of what each library is used for.

Table 2-1 Libraries Used in Linking XIL Programs

Library Name	Library Referred To
`-lxil`	The XIL library.
`-lX11`	The X library. If your program doesn't display any output in an X window, you can omit this library.
`-ldl`	The dynamic-linking library.
`-ldga`	The Direct Graphics Access library.
`-lthread`	The multithread library. Linking to this library now ensures full binary compatibility with the next XIL release, which will be thread-enabled. Applications that don't link with `-lthread` will run fine, but they won't take advantage of any thread support included in the next XIL release (see "The XIL Library is Multithread Unsafe" on page 7).

SPARC – The SPARCompiler™ C 3.0 or later compiler and SPARCworks™ C 3.0 or later debugger are the recommended tools for building XIL applications; each can be used to build and debug XIL applications that have been linked with `-lthread`. However, linking with `-lthread` doesn't work with the SPARCworks C 2.0.1 debugger (it does work with the SPARCompiler C 2.0.1 compiler); see the **Note** below.

x86 – The ProCompiler™ C 2.0.1 or later compiler and the ProWorks™ C 2.0.1 or later debugger are the recommended tools for building XIL applications. The ProCompiler C 2.0.1 compiler can compile XIL applications that link with `-lthread`, but linking with `-lthread` doesn't work with the ProWorks C 2.0.1 debugger; see the **Note** below.

Note – If your debugger doesn't work with `-lthread`, you may have to link without `-lthread` to debug your application (as shown on page 29), and then link with `-lthread` when the application is complete. Be aware, however, that compiling without `-lthread` generates a different binary file than compiling with `-lthread`. Thus, a feature that works fine in the binary file that isn't linked with `-lthread` may not work in the binary file that is linked with `-lthread`; or, the feature may not work in the binary file that is linked whereas it does work in the binary file that isn't linked.

Conditionally Compiling Code for Different XIL Versions

Beginning with release 1.2, the XIL library lets you conditionally compile code so you can take advantage of new interfaces in a current release while still supporting earlier releases of the library. To permit compile-time decisions, the library uses preprocessor directives to define symbols that identify the current XIL major and minor version numbers; these symbols—`XIL_API_MAJOR_VERSION` and `XIL_API_MINOR_VERSION`—are available to your application when you include the `xil.h` header file (see "Including Header Files" on page 12).

By convention, the major version number is the numeral preceding the decimal point in the XIL version number, and the minor version number is the numeral following the decimal point. Table 2-2 shows what the major and minor version numbers would be for various XIL releases.

Table 2-2 Examples of Major and Minor Version Numbers

XIL Version Number	Major Version Number	Minor Version Number
1.2	1	2
1.3	1	3
2.0	2	0
2.1	2	1

By using these symbols, you can maintain a single source-code file that can be compiled against different versions of the XIL library. For example, version 1.2 of the XIL library introduced `XilDevice` objects, which can be used to initialize device attributes before creating the corresponding device image; earlier versions of the library can't use this object. The following code fragment

shows how you might use the XIL_API_MAJOR_VERSION and XIL_API_MINOR_VERSION symbols to write code that uses an XilDevice object when you compile your application with the 1.2 version of the library, but that omits the object when you compile with earlier versions:

```
#if (XIL_API_MAJOR_VERSION == 1 && XIL_API_MINOR_VERSION >= 2) ||
    (XIL_API_MAJOR_VERSION > 1)
{
  /*create a device object */
  XilDevice device;
  device = xil_device_create(state, "SUNWrtvc");
  /* initialize the DEVICE_NAME attribute */
  xil_device_set_value(device, "DEVICE_NAME", (void *)devname);
  /* create a device image with the initialized attribute */
 rtvc_image = xil_create_from_device(state, "SUNWrtvc", device);
  xil_device_destroy(device);
}
#else
  /* create a device image without initialized attributes */
  rtvc_image = xil_create_from_device(state, "SUNWrtvc",
                                      (void *) devname);
#endif
```

In this example, a device image with an initialized attribute is created only for XIL versions 1.2 and later; otherwise, the device image is created without initialized attributes.

See "New Features" on page xxxi for a discussion of the new features introduced in the current release of the library.