Error Handling 8

The list below summarizes the most important facts about XIL error handling.

- Error handling is asynchronous. For example, assume that your program includes a call to `xil_copy()` that attempts to copy a three-band image to a single-band image. This is an error, and at some point the library will report that the source and destination images do not contain the same number of bands. However, this error may not be reported when the line in your program containing the `xil_copy()` is called. This is true because the XIL library's deferred execution scheme enables it to store many operations before executing them. (See Chapter 21, "Acceleration in XIL Programs," for more information on this deferred execution feature.) Later, when a set of stored operations is performed, the error will be reported.

  The fact that error reporting is asynchronous does not really affect the way you handle errors in your XIL programs, but it is something you should be aware of.

- The amount of error handling that you can do by simply checking the return values of XIL functions is limited. About half of the functions in the XIL library return `void`, and for the remaining functions, while you may be able to tell from the return value whether an error occurred, you won't be able to determine exactly what the error was. To handle errors properly, your program needs to examine the error data structure (of type `XilError`) that the library creates whenever an error occurs. This structure contains such information as an error ID and a string containing a description of the error.

- Your application can deal with the error structure referred to above in one of two ways:

- One option is to do nothing. In this case, the library's default error handler will look at the error structure and print a message to `stderr`. As shown in "An Example" on page 135, this option isn't recommended because the default handler prints information that is intended for application developers and isn't useful to an end user.
- Your other option is to write your own error handler and to install it; this is the recommended option. For information on writing such an error handler, see the next section "Writing an Error Handler," and for information on installing the routine, see the section "Installing and Chaining Error Handlers" on page 136.

Note – You can actually have more than one error handler installed at a time. In addition, you can chain these error handlers so that if the first error handler in the chain does not deal with an error, it can pass the error structure containing information about the error to a second error handler, and so on. This subject is explained in "Installing and Chaining Error Handlers" on page 136.

Writing an Error Handler

Any error handler you write should take one parameter, a handle to the error structure the library created when the last error occurred. In addition, the function should return a value of type `Xil_boolean`. Thus, the header of your function should look like this:

```
Xil_boolean function-name(XilError error);
```

The function should return `TRUE` if it has handled the error and `FALSE` if it has not.

The XIL library contains a number of routines you can use in building the body of your error handler. These are discussed in the next section, "Functions You Can Call in Your Error Handler." The section after that—"An Example" on page 135—shows how you might use these functions to write an error handler.

Functions You Can Call in Your Error Handler

This section discusses the XIL functions you can use in writing your own error handler. These functions fall broadly into two groups: those that get information from the error structure that the library passes to your error handler and those that retrieve information from structures of type `XilObject`.

Getting Information from an Error Structure

The functions listed in this section all take as their only parameter the name of the error structure the library has passed to your error handler.

xil_error_get_string()

This function returns an internationalized string (`char *`) that describes the error that just occurred. For a complete list of possible messages, see Appendix B, "XIL Error Messages."

You should either use this string immediately or make a copy of it because the pointer to it will become invalid after the next call your program makes to `dgettext(3I)` or to one of the XIL error functions that returns a string. Also, you should not free or modify this string.

xil_error_get_id()

This function returns a string (`char *`) of the form `di-`*n*, where is *n* is a unique identifier. For a complete list of possible error IDs, see Appendix B, "XIL Error Messages."

You should not free or modify this string.

xil_error_get_category()

This function returns an enumeration constant of type `XilErrorCategory`. The declaration of this enumeration is shown below.

```
enum XilErrorCategory {
    XIL_ERROR_SYSTEM,
    XIL_ERROR_RESOURCE,
    XIL_ERROR_ARITHMETIC,
    XIL_ERROR_CIS_DATA,
    XIL_ERROR_USER,
    XIL_ERROR_CONFIGURATION,
    XIL_ERROR_OTHER};
```

Table 8-1 explains briefly what types of error fall into each category.

Table 8-1 XIL Error Categories

Category	Explanation
`XIL_ERROR_SYSTEM`	The library cannot perform an operation correctly. This type of error is usually a secondary error caused by a user, resource, or configuration error.
`XIL_ERROR_RESOURCE`	Usually means that there is insufficient memory for the library to create an image or other data structure.
`XIL_ERROR_ARITHMETIC`	Indicates an arithmetic error such as a divide by 0.
`XIL_ERROR_CIS_DATA`	Means that the datastream being decompressed does not conform to the appropriate datastream definition.

Table 8-1 XIL Error Categories

Category	Explanation
XIL_ERROR_USER	The programmer has specified an invalid parameter to an XIL function. For example, for a lookup operation, the data type of the source image and the input data type of the lookup table might not match.
XIL_ERROR_CONFIGURATION	Usually indicates that the XIL software has not been installed properly or that an environment variable has not been set properly. The specific error may be that the runtime system is unable to find a loadable driver it needs.
XIL_ERROR_OTHER	Miscellaneous errors, such as Xlib and DGA errors.

xil_error_get_category_string()

This function returns a string (`char *`) that indicates which category of error has occurred. The strings for the possible categories are

- `System`
- `Resource`
- `Arithmetic`
- `Cis Data`
- `User`
- `Configuration`
- `Other`

As with the string returned by `xil_error_get_string()`, you should either use this string immediately or make a copy of it because the pointer to it will become invalid after the next call your program makes to `dgettext(3I)` or to one of the XIL error functions that returns a string. Also, you should not free this string.

xil_error_get_primary()

This function returns a value of type `Xil_boolean`, either `TRUE` or `FALSE`. A value of `TRUE` indicates that the error being processed was the primary cause of a problem you're seeing. For example, if you run out of memory and

`xil_create()` is unable to create an image, the primary error produced will be a resource error. The XIL library may also generate secondary errors as the `NULL` image is used internally.

xil_error_get_location()

This function returns a string (`char *`) that indicates where in the XIL library the error was generated. This location will help Customer Support to resolve any problems you report to them.

You should not free this string.

xil_error_get_object()

This function returns a handle of type `XilObject` to the object affected by the error. This object may be:

- An image
- An image type
- A lookup table
- A compressed image sequence (CIS)
- A dither mask
- A kernel
- A structuring element
- A region of interest
- A histogram

Once you have this handle, you can use the functions discussed in the next section to obtain further information about the object.

Note – If the error did not affect a particular object, you will receive a `NULL` handle.

Getting Information About the Object Affected by the Error

If your error handler uses the function `xil_error_get_object()` to retrieve a handle to the object affected by the last error, you can use the functions discussed below to request further information about the object.

xil_object_get_error_string()

This function attempts to get from an object a string containing additional information about the object. For instance, if you query a CIS object, you will get a string containing information about the values of the start frame, read frame, and write frame at the time of the error.

The prototype for this function is shown below.

```
void xil_object_get_error_string(XilObject object,
    char *string, int string_size);
```

The parameter `object` is the object handle returned by `xil_error_get_object()`; `string` is an array of `chars` in which the string will be returned; and `string_size` is the maximum number of characters the array can contain.

In the code fragment below, an error handler uses `xil_object_get_error_string()` to request information about the object affected by an error and prints any information that is available.

```
#define MAX 1024

Xil_boolean UserDefinedErrorFunc(XilError error)
{
    XilObject object;
    char buffer[MAX];

    object = xil_error_get_object(error);
    if (object) {
        xil_object_get_error_string(object, buffer, MAX);
        if (buffer[0] != 0)
            fprintf(stderr, "object info: %s\n", buffer);
```

xil_object_get_type()

You use this function to determine what kind of object the last error affected. The prototype for the function is shown below.

```
XilObjectType xil_object_get_type(XilObject object);
```

The parameter to the function is the object handle returned by `xil_get_error_object()`. The return value is an enumeration constant of type `XilObjectType`. The declaration of this enumeration is written as follows:

```
enum XilObjectType {
   XIL_IMAGE,
   XIL_IMAGE_TYPE,
   XIL_LOOKUP,
   XIL_CIS,
   XIL_DITHER_MASK,
   XIL_KERNEL,
   XIL_SEL,
   XIL_ROI,
   XIL_ROI_LIST,
   XIL_HISTOGRAM,
};
```

Once you know the type of object that was affected by the last error, you can cast the object handle returned by `xil_get_error_object()` to a handle to the particular type of object you have. For example, if the object is an image, you can cast the handle of type `XilObject` to one of type `XilImage`. Doing so gives you the ability to write code like the following:

```
XilObject object;

object = xil_error_get_object(error);
if (object)
    if (xil_object_get_type(object) == XIL_IMAGE)
        fprintf(stderr, "image bands: %d\n",
            xil_get_nbands((XilImage)object));
```

Because the error handler has a handle to the image, it can use any of the XIL functions that read image attributes to get additional information about the image. In this case, the error handler is finding out how many bands are in the image.

An Example

The library's default error handler prints a message like the one shown below each time an error occurs.

```
XilDefaultErrorFunc:
   error category: User
     error string: Destination colorspace not specified
         error id: di-203
   primary error detected at location colorConvert79 in XIL
```

As you can see, it indicates that the XIL default error handler handled the error, and it prints the category of the error, an error message, an error ID, and the location at which the error occurred, indicating whether the error was a primary or secondary error.

This information should be useful to the application developer because it verifies that the XIL handler handled this error, and also prints the error location, which is useful when contacting Customer Support. However, you wouldn't want your application to identify the error handler and error location because this information isn't useful to an end user and, moreover, would be confusing to that user. The code below shows how you could implement an error handler that doesn't identify the error handler and doesn't print the error location.

```
Xil_boolean UserDefinedErrorFunc(XilError error)
{
    if (xil_error_get_primary(error) == TRUE)
        fprintf(stderr, "\nPrimary Error:\n");
    else
        fprintf(stderr, "\nSecondary Error:\n");
    fprintf(stderr, "   error category: %s\n",
        xil_error_get_category_string(error));
    fprintf(stderr, "     error string: %s\n",
        xil_error_get_string(error));
    fprintf(stderr, "         error id: %s\n",
        xil_error_get_id(error));
    return TRUE;
}
```

8

Installing and Chaining Error Handlers

By default, when an XIL error occurs, the library creates an error structure of type `XilError` and passes that structure to the default error handler. To override this behavior, you can write your own error handler and install it. Once you install your error handler, the library will call it instead of the default error handler when an error occurs. If you write and install more than one error handler, the library always calls the most recently installed error routine. For more information on installing error handlers, see the next section, "Installing Error Handlers."

As mentioned above, the XIL library allows you to have more than one user-defined error routine installed at the same time. At first glance, this seems a waste because the library can only call the most recently installed routine when an error occurs. However, having more than one user-defined error handler installed is often useful because the library allows you to call one error handler from another, an action referred to as *chaining*. For an explanation of why you might want to chain error routines and how you go about chaining them, see the section "Chaining Error Handlers" on page 139.

Installing Error Handlers

To install an error handler you've written, you call the function `xil_install_error_handler()`, whose prototype is shown below.

```
int xil_install_error_handler(XilSystemState state,
    XilErrorFunc func);
```

The function's return value indicates whether the installation was successful. A value of `XIL_SUCCESS` means that it was successful, and a value of `XIL_FAILURE` means that it wasn't.

The parameter `state` is the system-state data structure you received when you initialized the library, and `func` is a pointer to your error handler. (`XilErrorFunc` is a defined type representing a pointer to a function.) Thus, the call you would use to install the error handler shown in the section "An Example" on page 135 would look like this.

```
XilSystemState state;

xil_install_error_handler(state, UserDefinedErrorFunc);
```

When you install an error handler, the library adds it to a list of error handlers that it recognizes. Figure 8-1 shows what the list would look like at this point.

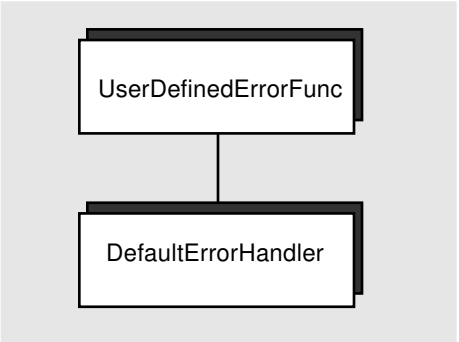

Figure 8-1 List of Error Handlers

If you then installed a third error handler, called `UserDefinedErrorFunc2`, the list would grow as shown in Figure 8-2.

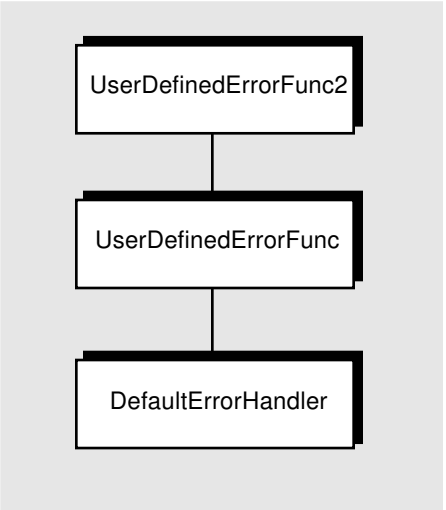

Figure 8-2 Adding to the List of Error Handlers

When an XIL error occurs, the library always calls the topmost routine (the last routine to be installed) in the list.

The library also includes a routine for removing error handlers from this list. Its name is `xil_remove_error_handler()`.

```
void xil_remove_error_handler(XilSystemState state,
    XilErrorFunc func);
```

The parameters to this function are the same as those for `xil_install_error_handler()`.

Note – If the same error handler has been installed twice, only the most recent version of the error handler will be removed.

Chaining Error Handlers

Again, having more than one error handler installed is useful because the library enables you to call one error handler from another. From the topmost error handler in a list of handlers, you can call either the next-to-last error handler to be installed or the default error handler. Thus, in your topmost error routine, you might handle only resource errors. If the error you're passed is a resource error, you take whatever action is appropriate and return `TRUE`; otherwise, you call the next error handler in the list. This next error handler might also check for one special type of error. Like the previous routine, if it is passed the proper type of error, it handles the error and returns `TRUE`; if it receives another type of error, it might call the error handler below it on the list. This process can continue until the error handler being called is the default error handler. This routine will handle any XIL error and always returns `TRUE`.

The functions you use to chain error handlers are `xil_call_next_error_handler()` and `xil_default_error_handler()`.

```
Xil_boolean xil_call_next_error_handler(XilError error);

Xil_boolean xil_default_error_handler(XilError error);
```

The parameter `error` is the error structure that the library passed to the topmost error handler in its list of error handlers (the last one to be installed). The return value of both functions indicates whether the error routine being called handled the error. A return value of `TRUE` indicates that the routine did handle the error, and a return value of `FALSE` indicates that it didn't.

The brief example below shows the routine `xil_call_next_error_handler()` being used in an error-handling routine. This routine simply ensures that if the error is a resource error, the program will exit after handling the error in the normal way.

```
Xil_boolean resource_errors(XilError error)
{
    int return_value;

    return_value = xil_call_next_error_handler(error);
    if (xil_error_get_category(error) == XIL_ERROR_RESOURCE)
        exit(1);
    return return_value;
}
```