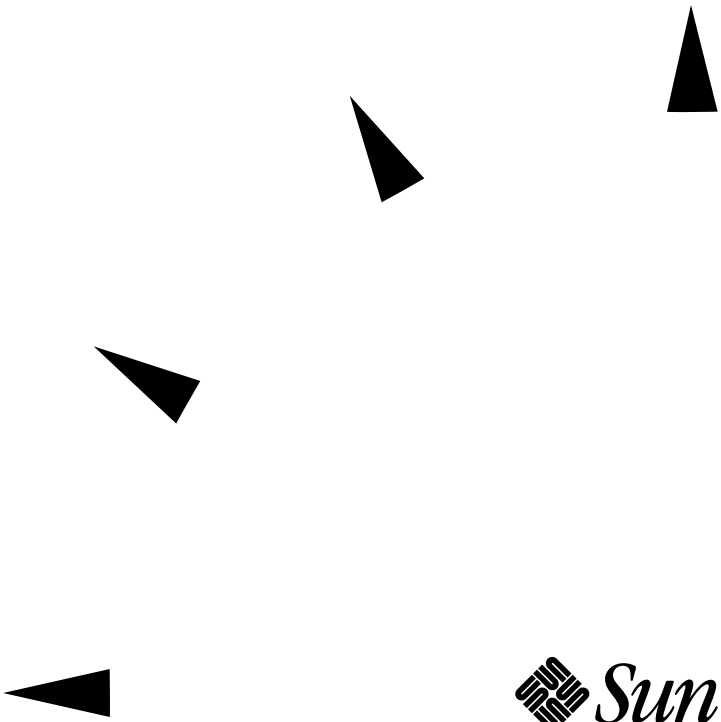
XIL Test Suite User's Guide



SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-5906-06
Revision 50, March 1997

Sun microsystems

THE NETWORK IS THE COMPUTER™

Please
Recycle

Adobe PostScript

Contents

List of Figures

List of Tables

Preface

What Is the Solaris XIL Imaging Library?

The Solaris™ XIL™ Imaging Library is the Solaris software's foundation imaging library. Foundation libraries are the lowest-level device-independent software layer of Solaris software. This level of interface is designed to support a wide variety of common functions; higher-level libraries can be built on top of the foundation, or an application can use the foundation layer directly.

The XIL Imaging Library is suitable for use by libraries or applications that require imaging or digital video capabilities, such as document imaging, color prepress, or digital video generation and playback. The current version of the XIL Imaging Library is multithreaded and supports tiling of images.

Prerequisites

You should be thoroughly familiar with the XIL Imaging Library and should have read the *XIL Programmer's Guide* before you begin working with the XIL Test Suite.

What Is the XIL Test Suite?

The XIL Test Suite is a suite of test programs that enable you to test XIL functionality and a set of functions that enable you to write your own test programs.

What's in This Book?

Chapter 1, "Overview of the XIL Test Suite" is an overview of the XIL Test Suite.

Chapter 2, "Running Test Programs in the XIL Test Suite" describes how to run test programs in the XIL Test Suite. Requirements for setting up and running the `Xilch` master control program, or harness, are described in detail.

Chapter 3, "Writing Test Programs in the XIL Test Suite Environment" describes how to write new test programs that will run under the `Xilch` harness. The functions in the XIL Test Suite library are described, and other pertinent details are discussed.

The XIL Test Suite may be useful to some system developers as they enhance XIL functionality by porting the XIL library to new devices or by creating new molecules. The usefulness of the XIL Test Suite to system developers is also described in this chapter.

Appendix A, "XIL Test Suite Directory Structure" describes the XIL Test Suite directory structure.

Appendix B, "Example Test Program" contains the text of an example test program.

Appendix C, "Equivalence Testing Example" provides an example of the equivalence testing functions.

Related Books

For information about the XIL Imaging Library, refer to the *XIL Programmer's Guide*, which provided in the SDK portion of XIL 1.3 AnswerBook™. This book explains how to use XIL functions to develop application programming interfaces (APIs) and end-user applications.

A companion to the *XIL Programmer's Guide* is the *XIL Reference Manual*, which is also provided in the XIL 1.3 AnswerBook. The reference manual contains man pages for all of the functions in the XIL library.

Because programming with the XIL library can be closely tied to programming with the X library, you may also find it useful to consult the *Xlib Programming Manual* and the *Xlib Reference Manual*.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

XIL 1.3 New Features

This document lists the new features in XIL 1.3. For a list and descriptions of the new XIL 1.3 functions, see the *XIL Programmer's Guide* in the *Solaris XIL 1.3 AnswerBook*.

MT-Safe and MT-Hot

The XIL 1.3 library is MT-safe. You can write multithreaded applications without putting locks around XIL functions. The library also is MT-hot. It takes advantage of multiprocessor systems without applications having to be rewritten.

Tiled Storage

XIL 1.3 stores very large images in separate buffers of contiguous memory called tiles. If a region of an image within a tile boundary is needed, only that tile is loaded into memory, thereby increasing performance.

Tiled storage is backwards compatible with existing XIL applications and can be completely transparent.

Optionally, you can manipulate XIL images stored as tiles by using tiling functions. Keep in mind, however, that XIL uses tiling behind the scenes to enhance performance whether or not an application makes explicit calls to tiling functions.

XIL 1.3 includes a new `XilStorage` storage object. `XilStorage` provides access to an image's data storage directly. Unlike the backwards-compatible `XilMemoryStorage` structure, the `XilStorage` object supports tiled as well as contiguous storage.

New Data Type

XIL 1.3 supports the 754 32-bit, single-precision, IEEE floating point data type. Using this data type allows you to develop highly sophisticated scientific imaging applications.

Temporary Images

XIL 1.3 supports temporary images. Temporary images are images used as an intermediate step in creating a subsequent image. They may only be written to, and read from, once. Temporary images are advantageous in that they improve XIL's ability to defer operations efficiently and circumvent having to call `xil_toss()`.

Storage Formats

XIL_GENERAL

XIL 1.3 supports a new `XIL_GENERAL` storage format. This format allows you the flexibility of obtaining input for each band of a multiband image from a separate source. Furthermore, each band can be in any of the formats that XIL supports.

XIL_BAND_SEQUENTIAL

XIL 1.3 now supports the `XIL_BAND_SEQUENTIAL` format for all data types, not just `XIL_BIT` images.

KCMS Integration

The XIL 1.3 library includes Kodak Color Management System (KCMS™) support. With KCMS integration, you can achieve as close as possible color matching between a display image and the actual stored image.

Additional XIL Functions

In addition, the XIL 1.3 library includes general object, double buffering, and other miscellaneous functions.

Backwards Compatibility

XIL 1.3 still supports the existing functions `xil_set_memory_storage()` and `xil_get_memory_storage()`, but these functions cannot be used in combination with the new XIL 1.3 functions for tiled images or the new storage formats.

Note – Existing applications should run with XIL 1.3 *without* being modified or recompiled.

Overview of the XIL Test Suite 1

The XIL Test Suite enables you, as a system developer, to verify that new functionality (devices and molecules) that you have added to the XIL Imaging Library is performing accurately. The XIL Test Suite provides a set of test programs to verify XIL functionality and provides a set of functions that enable you to write your own test programs.

System developers may find the XIL Test Suite useful during certain phases of the development process; however, using the XIL Test Suite is not a complete testing solution. In general, it is most useful for developers who are writing XIL molecules or who are writing handlers for compute devices such as accelerators. In these cases, you might choose to test your code against reference data.

The usefulness of the XIL Test Suite to your development process depends on the product you are developing:

- If you are writing a compute device handler, you can use the XIL Test Suite to verify that your version of the XIL atomic functions is equivalent to the XIL implementation.
- If you are writing XIL molecules, you can write additional test programs that would invoke your molecule and test it against the corresponding atomic functions in the XIL Imaging Library.
- If you are developing an input device handler, the XIL Test Suite is not useful in directly testing it.

- If you are developing a storage device handler, the XIL Test Suite cannot directly test it, but it can test an associated compute handler. If the tests for the operators that use the storage handler pass for a variety of images, it is an indication that the storage device handler is performing properly.

For more information on these XIL device types, consult the *XIL Device Porting and Extensibility Guide*.

Test Programs

The XIL Test Suite test programs were designed to test XIL Imaging Library functions using a wide range of input parameters. Stored versions of data, referred to as *reference data*, were produced when running the test programs to verify the XIL functions. Reference data are images and colormaps, for example. Because reference data requires large amounts of disk space for storage, *reference signatures* were created from the reference data. These reference signatures are provided with the XIL Test Suite for use in regression testing. For information about how the XIL Test Suite uses test programs and references, see Chapter 2, “Running Test Programs in the XIL Test Suite.”

Because of the low-level nature of the XIL Imaging Library, it is critical that the functions in this library perform as expected. Therefore, the code for test programs was reviewed to ensure accuracy, and reference data was verified by visual inspection.

Reference Data

You use reference data to verify the accuracy of the development library. Again, reference data is stored versions of data produced by tests programs and has been verified as accurate. Therefore, reference data can be used to debug development code.

Reference data is not provided but can be generated, as discussed in Chapter 2, “Running Test Programs in the XIL Test Suite.” The primary disadvantage of using reference data is the large amount of disk space required if images are saved for all useful permutations of image sizes, pixel types, and other parameters. The disk space used by these images can be tens of gigabytes.

Also, to verify the accuracy of development code, you can specify a tolerance range for comparisons when you run the XIL Test Suite. With a specified tolerance range, you can successfully verify data even if there are minor

discrepancies caused by floating point precision effects. These discrepancies occur mostly in operations such as convolution and compression / decompression.

New options have been added to the XIL Test Suite for improvement of image comparison tolerance. An absolute and relative tolerance between the current and reference images can be defined.

It is also possible to save discrepancies between current and reference images for later visual inspection of discrepancies or for printing the differences between them.

Reference Signatures

Reference signatures are used for regression testing. Reference signatures represent a Cyclic Redundancy Codes (CRC) checksum derived from reference data. For more information on the CRC algorithm used in the XIL Test Suite, see the document entitled “A Painless Guide to CRC Error Detection Algorithms” by Ross Williams. You can locate this document on the web at `ftp://ftp.rocksoft.com/clients/rocksoft`. A reference signature uniquely identifies the data, just as a person’s signature uniquely identifies that person. Reference signatures are used to avoid storing a large amount of data.

Regression testing is performed by producing a signature for the functions being tested and comparing that signature to the corresponding stored reference signature. If differences are found, errors are assumed to be in the functions being tested.

Benchmarking

The XIL Test Suite includes functions that allow you to create benchmarking programs to measure the performance of XIL functions. Performance is measured in frames per second. Chapter 3, “Writing Test Programs in the XIL Test Suite Environment” includes a discussion of how to write benchmarking programs.

1

What Is Tested?

To thoroughly verify each function, the XIL Test Suite provides tests that are appropriate to the function and to the platform on which you are running. The following three sections, “Test Conditions,” “Deferred Execution,” and “Architecture / Platform Testing” discuss XIL function testing.

Test Conditions

The XIL Test Suite uses a set of general and function-specific test conditions to test each function. In the general case, all functions are tested under permutations of the following conditions when possible and appropriate:

- All pixel types
- Various image sizes
- Various numbers of bands
- Various image origins
- With and without regions of interest (ROIs)
- Images as parent and child images
- With random (noise) and ordered images as the source

Many of the functions are tested with function-specific test conditions. For example, the geometric operations test for additional parameters, such as rotation angles, scale factors, and interpolation types.

Deferred Execution

In general, testing of the deferred execution mechanism is no different from testing a particular function, except that you verify images after a *group* of function calls instead of after individual function calls. Consult the *XIL Programmer’s Guide* for information on deferred execution.

Because the current release of the XIL Library is multithreaded and supports tiled images, control options are added to support testing XIL’s threading and tiling performance.

Architecture/Platform Testing

You can verify XIL functions on all supported platforms and architectures. To verify XIL functions on a particular platform, you must run test programs on the platform you wish to verify. By doing this, you verify the consistency of test cases from platform to platform.

The Parts of the XIL Test Suite

The XIL Test Suite has three parts:

- `Xilch`, the master control program, or harness, that runs all of the tests in the test suite.
- `libts`, the test suite library that provides utility functions and functions to generate and verify data. Also, you can write your own test program using these functions.
- A set of test programs, which are dynamically linked with functions in `libts`, that verify the accuracy of the XIL functions being tested.

Figure 1-1 shows the test suite's parts.

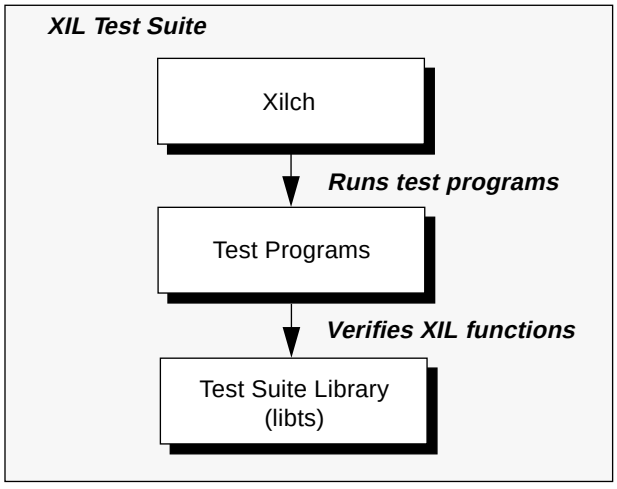


Figure 1-1 Parts of the XIL Test Suite

1

Test programs consist of `libts` functions. These functions are used to exercise the XIL function that is being verified. You can invoke a test program through the `Xilch` harness, or you can run the test program by itself.

The `Xilch` harness accepts a variety of options that allow you to precisely control what is being tested or created. See Chapter 2, “Running Test Programs in the XIL Test Suite,” for a detailed discussion about how the XIL Test Suite works.

Running Test Programs in the XIL Test Suite

2

This chapter describes how to run the XIL Test Suite. It explains how the XIL Test Suite works, explains what options you can set when running the test suite, and provides examples of invocations.

Getting Started

Before you begin using the XIL Test Suite, make sure that you have the correct packages of the Solaris Driver Developer's Kit (DDK) installed, that you have the correct resources, and that you set the needed environment variables.

Packaging

You must have the following Driver Developer's Kit packages installed to run the XIL Test Suite and write your own test programs.

- SUNWxildh
- SUNWxiltg
- SUNWxilts, if you are running on SPARC
- SUNWxiltx, if you are running on x86

Resources You Need

The XIL Test Suite is supported on SPARC® and x86 platforms with the following minimum configurations:

- SPARC: A SPARCstation™ 2 (with a minimum of 100MB of swap space) running the current version of Solaris system software
- x86: A 486 (with 100MB of swap space) running the current version of Solaris system software

Before you can run the XIL Test Suite, you also need the following:

- The XIL Imaging Library
- SPARC: The SPARCompiler™ C 2.0.1 and SPARCWorks™ 2.0.1 software or later (if you plan to write your own test programs)

Note – See the the *XIL Programmer's Guide* in the *Solaris XIL 1.3 AnswerBook* for recommended SPARC compilers.

- x86: The ProCompiler™ C 2.0.1 and ProWorks™ 2.0.1 software (if you plan to write your own test programs)

Environment Variables

Before running the XIL Test Suite, you must set some environment variables. If you used the default installation directory when you installed the XIL library, you probably have already set the `XILHOME` environment variable as follows:

```
% setenv XILHOME /opt/SUNWits/Graphics-sw/xil
```

Assuming that you've installed the Solaris XIL Imaging Library Driver Developer's Kit (of which the XIL Test Suite is a part) according to the defaults, you must set the other needed environment variables as follows:

1. **Set the** `XILCHHOME` **environment variable to the location of the current** `Xilch` **directory:**
   ```
   % setenv XILCHHOME /opt/SUNWddk/xil/ddk_2.4/Xilch/arch
   ```
   where *arch* is either `sparc` or `i386` depending on which platform you are running.
2. **Update your** `LD_LIBRARY_PATH` **to include** `$XILCHHOME/lib`:
   ```
   % setenv LD_LIBRARY_PATH \
   $XILHOME/lib:$XILCHHOME/lib:$LD_LIBRARY_PATH
   ```

3. **If you want to run XGL tests, you must set the `XGLHOME` environment variable to point to the top directory of your XGL installation and appropriately append `LC_LIBRARY_PATH`.**
   See your XGL documentation for details.

4. **If you do not want to tile images during the test, you need to set the environment variable `XILCH_NO_TILES` to `TRUE`.**

5. **If you do not want to use multithreading capability of the XIL library, you must set the `XIL_DEBUG` environment variable:**
   `%setenv XIL_DEBUG=threads=`*ncpu*

   where *ncpu* is the number of CPUs on the machine.

Environment Notes

To display decompression tests, you must be running the OpenWindows™ environment or Common Desktop Environment (CDE). Also, a user cannot use a system to run tests if he or she is not the user logged into the console.

How the XIL Test Suite Works

As discussed in Chapter 1, “Overview of the XIL Test Suite,” the XIL Test Suite has three parts:

- `Xilch`, the master control program, or harness, that runs all of the tests in the test suite.
- `libts`, the test suite library that provides utility functions and functions to generate and verify data. Also, you can write your own test program using these functions.
- A set of test programs, which are dynamically linked with functions in `libts`, that generate test data.

You can run `Xilch` to:

- Verify XIL functions against reference signatures
- Verify XIL functions against reference data (for example, images and color maps)
- Create reference signatures for XIL functions and place the signatures in a reference directory

- Create reference data for XIL functions and place the data in a reference directory

When you create test data, `Xilch` determines which platform you are running on and generates the test data for that platform. Reference signatures and data may be platform-specific, or they may be generic (same result on multiple platforms). The SPARC platform is considered the “generic” platform. By default, the reference directory for the SPARC platform is named `generic` and contains all reference signatures for all test programs.

When you invoke a test program through `Xilch`, the test program dynamically links with the XIL Imaging Library. If you are verifying an XIL function, test signatures or data are created for the function and compared to the stored reference signatures or data for that function.

Figure 2-1 is an example of how `Xilch` works. In the figure, the testlist file, a file that contains a list of test programs, begins by starting `test1`. In this example, `test1` calls:

- `ts_get_src1_image()`, which retrieves input data as specified in the test matrix file via the function in `libts`.
- `xil_add()`, which makes a call to the XIL Imaging Library.
- `ts_verify()`, which verifies that the test data is identical to the stored reference data. The comparison is made through `libts`. Results of the comparison are displayed on your terminal (`stdout`). Or, when you invoke `Xilch`, you can specify that results print to a log file of a specified name.

Note – If you invoked `Xilch` to create reference data, the test data generated by each test program is placed in a reference directory; no comparison is made.

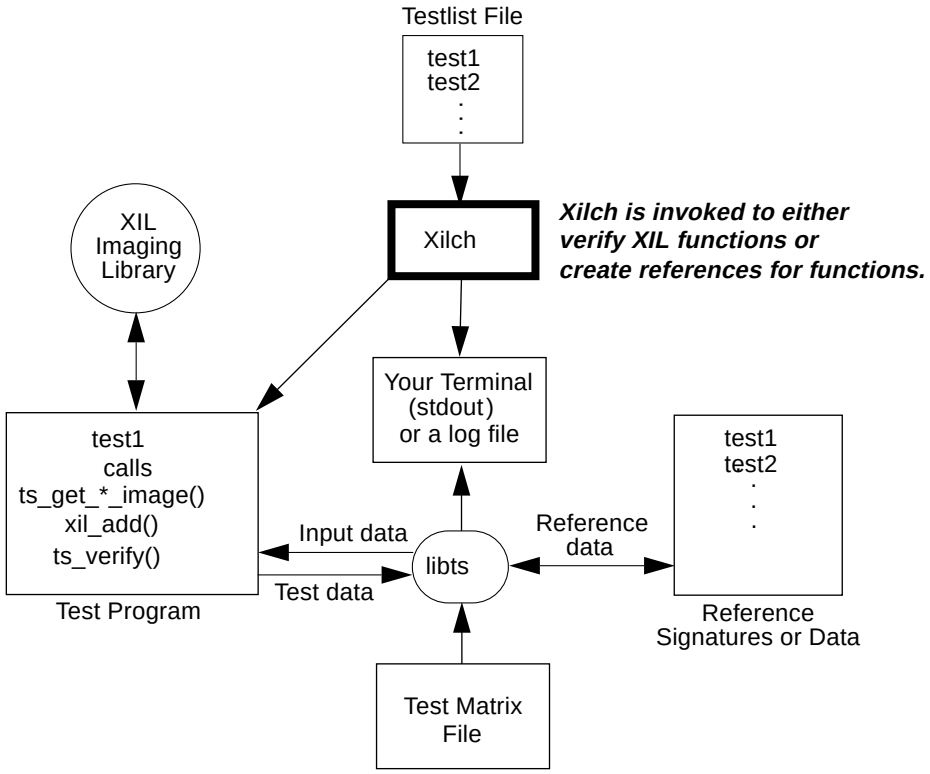


Figure 2-1 `Xilch` Example

Verifying Against References

As discussed in Chapter 1, “Overview of the XIL Test Suite,” you can use reference signatures and data to verify the accuracy of XIL Imaging Library functionality. Reference signatures are used primarily for regression testing, and a set of reference signatures for the XIL Imaging Library functions are provided with the XIL Test Suite. Reference data is used primarily for verifying the accuracy of new development code.

When you want to verify against reference signatures, you need not specify an option when running `Xilch` because the `-test_signatures` option is the default.

When you want to verify against reference data, you specify the `-test_data` option when running `Xilch`. This option provides a more thorough testing (pixel-by-pixel comparison), but it also takes longer to execute. It is a very convenient test when you are developing new functionality in your library and need to compare execution of new functions against the old reference data.

Creating References

Reference data for the XIL Imaging Library functions are not provided with the XIL Test Suite; only reference signatures are provided. However, you can create reference data by running all of the XIL Test Suite test programs with the `-create_data` option when running `Xilch`.

Also, if you run the test programs using different test matrix files (input data) than the test matrix files provided with the XIL Test Suite, you will need to create your own set of references (signature and / or data) for the new input data. Test matrix files are discussed in the section “The Test Matrix File” on page 17.

Files Used or Created by `Xilch`

`Xilch` uses two files and can create one file when it runs. When you invoke `Xilch`, you can specify which testlist and test matrix file to use. Default versions of both of these files are called automatically if you don’t specify them on the `Xilch` command line.

When you invoke `Xilch`, you can specify that a log file be created that will contain information reported by `Xilch` as it runs. By default, this information is displayed at your terminal and not printed to a log file.

With the `-save_images` option, you can also create a reference image and test the image (image created by a particular test), the absolute error image, and an absolute threshold image. These images can be displayed as `.vff` images and discrepancies visually inspected. There is also a file created that prints differences between the reference and absolute image on a pixel-by-pixel basis. To limit the file size, the number of errors recorded has a pre-determined upper bound.

The Testlist File

The testlist file is a list of test programs, one per line, and any options for those tests. If you do not specify a testlist file, `Xilch` first looks for a file called `testlist` in the current directory; then, it looks in `$XILCHHOME/config`. The default testlist file contains many of the available test programs. If you want a testlist file that contains all the test programs, you must create your own testlist file. However, to spread the execution time of `Xilch` tests, it is recommended that you create several test files and run separate `Xilch` tests, depending on the category of functions being tested.

If you want to use a testlist file other than the default file when running `Xilch`, you specify the testlist file to `Xilch` using the `-f` option. For example,

```
% Xilch -f my_testlist
```

If one of the test programs in the testlist file fails, `Xilch` continues to run through the rest of the programs in the list except in the case of a deadlock in a multithreaded environment. Currently there is no provision for removing deadlocked tests.

When you create a testlist file, you can specify conditions for a particular test by using `Xilch` options. For example, suppose you want to use a different test matrix file for a test program named `my_test` in your testlist file. (See “The Test Matrix File” on page 17 for a description of a test matrix file.) In this case, after `my_test` in your testlist file, you would use the `-m` option to specify the name of the test matrix file you want to use. For example,

```
% $XILCHHOME/bin/my_test -m my_test_matrix
```

Then, when you invoke `Xilch`, you can either use the default test matrix file or specify another test matrix file. When `Xilch` reaches `my_test` in your testlist file, it will substitute the test matrix file specified for `my_test`.

Note – Testlist file options override command-line options.

An easy way to create your own customized testlist file is to start with the `testlist` file in `$XILCHHOME/config`. Copy it to a file with a name you choose. Then edit the file, removing the test programs you don’t want to run

and inserting additional programs. The following table lists the test programs available at the time of the printing of this manual. For the most up-to-date list of test programs, look in the `$XILCHHOME/bin` directory.

Table 2-1 Test Programs Available in `$XILCHHOME/bin` *(1 of 3)*

XIL Function Type	Test Program
Arithmetic	`absolute_test`
	`arith_test`
	`logical_test`
Compression / CIS	`cbm_test`
	`cell_compress_test`
	`cell_decompress_test`
	`cellb_compressor_test`
	`cellb_decompress_test`
	`fax_test`
	`fax_compressor_test`
	`fax_decompress_test`
	`jpeg_compressor_test`
	`jpeg_decompress_test`
	`jpegll_test`
	`mpeg1_decompress_test`
	`px64_decompress_test`
Geometry	`affine_test`
	`rotate_test`
	`scale_test`
	`subsample_adaptive_test`
	`subsample_binary_to_gray_test`
	`tablewarp_test`
	`translate_test`
	`transpose_test`
Miscellaneous	`absolute_test`

Table 2-1 Test Programs Available in `$XILCHHOME/bin` *(2 of 3)*

XIL Function Type	Test Program
	`band_combine_test`
	`black_generation_test`
	`color_convert_test`
	`combined_lookup_test`
	`convolve_test`
	`copy_pattern_test`
	`copy_test`
	`copy_with_planemask_test`
	`create_copy_test`
	`device_test`
	`dilate_test`
	`edge_detection_test`
	`erode_test`
	`extrema_test`
	`fb_rw_test`
	`histogram_test`
	`interpolation_table_test`
	`lookup_convert_test`
	`minmax_test`
	`photocd_test`
	`rescale_test`
	`set_value_test`
	`threshold_test`
	`tile_use_test`
	`xil_lookup_test`
Molecules	`memory_test`
Object	`dag_test`

Table 2-1 Test Programs Available in `$XILCHHOME/bin` *(3 of 3)*

XIL Function Type	Test Program
	`dithermask_test`
	`image_test`
	`kernel_test`
	`lookup_test`
	`non_std_roi_test`
	`roi_test`
	`roi_get_as_image_test`
	`state_test`
	`sel_test`
Presentation	`blend_test`
	`choose_colormap_test`
	`error_diffusion_test`
	`fill_test`
	`nearest_color_test`
	`ordered_dither_test`
	`paint_test`
	`soft_fill_test`
	`squeeze_range_test`
Utility	`cast_test`

Most of the names of the test programs include the name of the XIL function that's tested. For example, the `Xilch` test program `affine_test` verifies the correct functioning of the XIL function `xil_affine()`. However, the names listed below are not obvious.

`cbm_test`	Tests CIS Buffer Manager functionality
`fb_rw_test`	Tests frame buffer read-write functionality

The Test Matrix File

A test matrix file provides descriptions of input data (images) to `Xilch`. A test program loops through the test matrix file, retrieving images via `libts` functions, until it runs out of images. If you do not specify a test matrix file, the default file, `xilch_tests` (in `$XILCHHOME/config`), is used. Chapter 3, "Writing Test Programs in the XIL Test Suite Environment," provides an example of a test matrix file.

The test programs for some XIL functions are designed to work with specialized matrix files, which are also in `$XILCHHOME/config`. For example, the matrix file called `fax_tests` is designed to be used with the test program that verifies the fax decompression portion of the XIL library.

Note – The images referenced in the test matrix files are located in the directory `$XILCHHOME/data/images/std`. These images are all in .vff format. This is an unsupported, Sun-internal format. The format of these images may change in future releases.

You specify the `fax_tests` matrix file by using the `-m` option in your testlist file. In this case when you invoke `Xilch`, it uses the default matrix file for every test program in your testlist file except for the fax test program, where it uses the `fax_tests` matrix file that you specified in your testlist file. If you decide to add a test program to the XIL Test Suite, you might want to create new test matrix files as discussed next.

The content of a test matrix file is tightly coupled with `Xilch` reference data (that is, reference data reflects the parameters used in the matrix file when they were created during the `Xilch` run with that file). Therefore, if you decide to change the matrix file for future runs, you must also recreate the reference data and signatures based on the new matrix file.

Writing a Test Matrix File

Images can be generated from parameters or loaded from a specified file. The functions `ts_get_src[123]_image()` and `ts_get_dst[123]_image()` return source and destination images based on the tables defined in the test matrix file (explained in the next section). See Code Example 3-2 for an example of how this is done. A test matrix file contains *tags*, which specify the images used by a test program.

Tags of a Test Matrix File

The tags that can occur in a test matrix file are described in detail in this section. For an application showing how these tags are used in a test matrix file, see “Test Matrix File Example” on page 20.

Note – Tags and labels are case insensitive; parameters are case sensitive. All parameters must be specified in the order shown, separated by whitespaces; there are no default values. Also, the values of the parameters must be of the correct data type.

`Table [SRC1|SRC2|SRC3|DST1/DST2/DST3]` *xsize ysize nbands datatype imagecontent value xorigin yorigin filename*

The `Table` tag denotes the beginning of a table of image data. The `Table` tag must be followed by one of the following table labels: `SRC1`, `SRC2`, `SRC3`, `DST1`, `DST2`, or `DST3`. SRC and DST labels correspond to the parameter names in the `libts` functions `ts_get_src[123]_image()` and `ts_get_dst[123]_image()`. Parameters of the `Table` tag are:

xsize	Width of the image.
ysize	Height of the image.
nbands	Number of bands in the image.
datatype	`XIL_BIT` (unsigned), `XIL_BYTE` (unsigned), `XIL_SHORT` (signed), or `XIL_SHORT` (floating point).
imagecontent	`TS_RANDOM`: image filled with numbers from a random number generator (user-specified seed number), `TS_CONSTANT`: image filled with a user-specified constant, `TS_RAMP`: image filled with increasing numbers, or `TS_NOFILL`: image filled with whatever random values are in memory.
value	Value associated with the *imagecontent*. It can be a seed or a constant. This number must be less than 256 for `XIL_BYTE TS_CONSTANT` images. For `XIL_SHORT TS_CONSTANT` images, this number must be between -32,768 and 32,767.

xorigin	x coordinate of the origin.
yorigin	y coordinate of the origin.
filename	File name of a stored image file in .vff format. All other parameters in the line, except *xorigin* and *yorigin*, are ignored but must appear as placeholders. If you do not specify a prestored image, then you must enter “NULL” for this parameter (see the example). If a file name is specified, its parameters supersede other specified parameters, such as *xsize* and *ysize*.

`Child` *<Label> ch_x ch_y ch_xsize ch_ysize ch_startband ch_num_of_bands*

The `Child` tag specifies the characteristics of a child image. A Label name is not required, but can be used as an optional identifier. A `Child` tag always immediately follows the parameters of its parent image (see example test matrix files). Parameters of the `Child` tag are:

ch_x	x coordinate offset from the parent image.
ch_y	y coordinate offset from the parent image.
ch_xsize	Width of the child image.
ch_ysize	Height of the child image.
ch_startband	Specifies which band in the parent image the child image starts in (0 based).
ch_num_of_bands	Number of bands in the child image.

`ROI` *roi1_x roi1_y roi1_xsize roi1_ysize roi2_x roi2_y etc...*

The `ROI` tag is used to specify the characteristics of a region of interest. The *Label* parameter option is not available with this tag. Parameters of the `ROI` tag are:

roi1_x	x coordinate for the first region.
roi1_y	y coordinate for the first region.
roi1_xsize	Width of the first region.
roi1_ysize	Height of the first region.

`End` *<Label>*

The `End` tag indicates the end of a `Table`. A *Label* name, corresponding to the Table label, is optional (see "Test Matrix File Example" on page 20).

`Table REF1` *ref_name*

The `REF1` label lists the reference names for the `ts_*_verify()` function. For `ts_*_verify()` to look in the test matrix file for reference names, you must specify the `-ref_matrix` option when running `Xilch`.

ref_name	Name of a reference that corresponds to the sequential calls to `ts_*_verify()` in the test program.

Requirements and Limitations

Test programs that use two source images should call `ts_get_src1()` and `ts_get_src2()`. A test program can create its own destination image, or it can call `ts_get_dst1()`. The XIL Test Suite library makes no consistency checks between tables. For example, it would not notice if the number of bands in the third image of the SRC2 table was the same as the number for the third image in the SRC1 table.

Test Matrix File Example

The following example will give you an idea of the kind of information contained in a typical test matrix file. Note that each file can contain as many parent and child images as you want. Child images must always follow the parent image with which they are associated; ROI tags must always follow the image, parent, or child with which they are associated.

This example specifies one source table (SRC1) that has two parent images and a child image associated with the second parent image, and one destination table (DST1) that has two parent images.

The Source Table

The first parent image is a 256 x 256, single-banded, short image initially filled with random data, using 12989 as the seed; its origin is (0.0, 0.0). `NULL` means no file is associated with this image. The second parent image is a 257 x 193, 3-

banded, unfilled bit image with origin coordinates of (10.0, 10.0). The child image is offset from its parent (the image specified directly above it) in x by 7 and y by 13. It's a 23 x 37, 2-banded image that begins in the second band of its parent.

The Destination Table

The first parent image is a 256 x 256, single-banded, short image filled with constant data (0's); its origin is (10.0, 10.0). The second parent image is a 257 x 193, 3-banded, bit image initially filled with random data, using 73142 as the seed; its origin is (0.0, 0.0).

```
TABLE SRC1
    256 256 1 XIL_SHORT TS_RANDOM 12989 0.0 0.0 NULL
    257 193 3 XIL_BIT TS_NOFILL 0 10.0 10.0 NULL
CHILD 7 13 23 37 1 2
END SRC1

TABLE DST1
    256 256 1 XIL_SHORT TS_CONSTANT 0 10.0 10.0 NULL
    257 193 XIL_BIT TS_RANDOM 73142 0.0 0.0 NULL
END DST1
```

The Log File

By default, the information reported by `Xilch` is displayed on your terminal (`stdout`). You can specify that this information be printed to a log file by using the `-l` option when you invoke `Xilch`.

The information reported by `Xilch` consists of a header and the results of the test programs as they iterate through the test matrix file. The header contains the following information:

`Xilch` Started	Time the `Xilch` run began
User	User name of the person running `Xilch`
Host	Name of machine running `Xilch`
Invocation	What the user typed to invoke this run of `Xilch`

XILHOME	The value of this environment variable
XILCHHOME	The value of this environment variable
XILCHREFDATA	The value of this environment variable

At the end of the log file is information stating the number of programs executed, the number of failures, and the percentage of program success. You can look at this information to see quickly the results of the test program.

The following is an excerpt from a `Xilch` run that has no failures.

Code Example 2-1 Log File Example *(1 of 2)*

```
*************************************************************
Xilch Started: May 20 94 01:03:13
User: xil
Host: emulsion
Invocation: bin/Xilch -l Xilch_run.log
XILHOME: /home/xil/devel/sparc/sparc
XILCHHOME: /home/xil/xilch/devel/sparc/sparc
XILCHREFHOME: /home/xil/xilch/devel/sparc/data/references
-------------------------------------------------------------
> /home/xil/xilch/devel/sparc/sparc/bin/arith_test: Started
01:03:14
> /home/xil/xilch/devel/sparc/sparc/bin/arith_test: Passed
01:05:30
-------------------------------------------------------------
> /home/xil/xilch/devel/sparc/sparc/bin/black_generation_test:
Started 01:05:30
> /home/xil/xilch/devel/sparc/sparc/bin/black_generation_test:
Passed 01:06:50
-------------------------------------------------------------
> /home/xil/xilch/devel/sparc/sparc/bin/blend_test -m
blend_tests: Started 01:06:50
> /home/xil/xilch/devel/sparc/sparc/bin/blend_test -m
blend_tests: Passed 01:12:24
-------------------------------------------------------------
> /home/xil/xilch/devel/sparc/sparc/bin/cast_test -m
quick_tests: Started 01:12:25
> /home/xil/xilch/devel/sparc/sparc/bin/cast_test -m
quick_tests: Passed 01:13:13
.
.
.
-------------------------------------------------------------
```

Code Example 2-1 Log File Example *(2 of 2)*

```
************************************************************
> /home/xil/xilch/devel/sparc/sparc/bin/xil_lookup_test: Started
01:37:30
> /home/xil/xilch/devel/sparc/sparc/bin/xil_lookup_test: Passed
01:38:59
------------------------------------------------------------

>>>>> Number of programs executed: 27
>>>>> Number of program failures: 0
>>>>> Program success %: 100.00%

Xilch Done: May 20 994 01:38:59
```

`Xilch` *Command Line Options*

The `Xilch` harness runs the test programs in the test suite. Use its options to vary testing conditions. You have already learned about some of the options you can use with `Xilch`: `-f` to specify a testlist file, `-m` to specify a test matrix file, and `-l` to specify a log file. Following is a complete list of the command line options.

Table 2-2 `Xilch` Command Line Options *(1 of 4)*

Option	Description
`-create_data`	Creates reference data in the reference directory. This option cannot be used with `-create_signatures`, `-test_signatures`, or `-test_data`. See also `-ref_directory`.
`-create_signatures`	Creates reference signatures in the reference directory. This option cannot be used with `-create_data`, `-test_signatures`, or `-test_data`. See also `-ref_directory`.
`-D`	Turns on display mode. This option displays both the test image and the reference image, if any.

Table 2-2 `Xilch` Command Line Options *(2 of 4)*

Option	Description
`-f` *file*	Use *file* as the testlist file. The default file is `testlist` in the current directory, and if not found there, the file `testlist` in `$XILCHHOME/config`. Each line in the testlist file specifies the name of a test program to run and its options, if any.
`-l` *logfile*	Uses *logfile* to log test information reported by `Xilch` and the test suite library. The default log file is `stdout`.
`-m` *file*	Uses *file* as the test matrix file for all test programs in this `Xilch` run. If file cannot be found in the current directory, then `$XILCHHOME/config` is searched. If *file* is still not found, then `$XILCHHOME/config/xilch_tests` is used. This option does not really affect the operation of `Xilch` itself. Instead, it is passed to the test program and used by the test suite library to find and/or generate its test images.
`-no_auto`	Turns off automatic testing.
`-no_ref_backup`	Does not back up the references before creating new references. By default a backup reference is created.
`-percent` *value*	Uses *value*, a floating point number between 0 and 100, as the acceptable percentage difference for comparisons. A value of `0` means that the test data and the reference data must match perfectly. A value of `10` means up to 10% of the pixels can be different.
`-platform` *string*	Species the name of the platform *(string)* for which you want to create references or verify data (for example, x86). The default value is the CPU on which the test is run. For example, if you run the tests on a SPARC platform, the default platform name is `generic`; if you run the tests on an `x86` platform, the default platform name is `x86`. This can be used to create accelerator-specific references. When creating references, `Xilch` uses the specified platform to create the references and stores them in a directory for that platform. When testing, `Xilch` uses the references stored for the specified platform.

Table 2-2 `Xilch` Command Line Options *(3 of 4)*

Option	Description
`-ref_directory` *dir*	Uses the directory *dir* to obtain references when testing or to store references when creating references. When this option is specified, the value overrides the default directory name or the directory name specified in the environment variable `XILCHREFDATA`. When testing, the reference directory you specify must have subdirectories: signatures for signature testing and `data` for data testing. When creating reference signatures or data, `Xilch` creates subdirectories if they do not exist. This option is useful to create references in a temporary location during development.
`-t` *prog args*	Runs only the test program *prog*. `Xilch` runs in single-test mode when this option is specified and ignores any testlist file. When you specify only one test program to run, you can also specify any of the other `Xilch` command line options, *args*, other than `-f`.
`-test_data`	Tests images against data. This option cannot be used with `-test_signatures`, `-create_data`, or `-create_signatures`.
`-test_signatures`	Tests images against signatures. This is the default. This option cannot be used with `-test_data`, `-create_data`, or `-create_signatures`.
`-tol` *value*	Uses *value* as the floating point tolerance for comparisons. For `ts_*_verify()`, this option is meaningful when used with the `-test_data` option. For XIL_SHORT images, *value* can be any number between 0 and 65535. For XIL_BYTE images, *value* can be any number between 0 and 255. In both these cases, a *value* of 0 means that the test data perfectly matches the reference data. If you specify the maximum *value*, the comparison always succeeds. For `XIL_BIT` images, the tolerance value is ignored.

Table 2-2 `Xilch` Command Line Options *(4 of 4)*

Option	Description
	For most XIL functions, you should specify a tolerance of 0, because you should be able to achieve a perfect pixel-to-pixel comparison. However, this perfect correspondence is not achievable for the geometric operators. It also may not be possible for dithering, error diffusion, and compression operations. In these cases, your tolerance value probably needs to be greater than 0, keeping in mind that you want to achieve as close a match as possible.
`-use_ref_backup`	Uses the backup references for each test case. By default, when you create references, `Xilch` backs up the existing references before creating the new references. When you specify this option on the `Xilch` command line, `Xilch` uses the backup references. See also `-no_ref_backup`. This option is valid only when either the `-test_data` or `-test_signatures` option is specified.
`-v`	Runs in verbose mode. This option also reports additional information for tests run without the harness. See "Additional Options For Running Individual Tests" on page 29 for details.
`-use_ref_matrix`	Uses the reference names listed in the test matrix file instead of the names specified in the calls to `ts_*_verify()`. When you use this option, the last parameter of `ts_*_verify()`, *ref_name*, is ignored. This option can be used only with `-test_data` and is intended for use during test development, not regression testing.

Note – All of the `Xilch` options can be used inside a testlist file.

Example Invocations

In this section, several example invocations of `Xilch`, using different options, are provided to give you a feel for ways to use the test suite environment.

```
% Xilch
```

`Xilch` is invoked with the following default conditions:

- The file `testlist` in the current directory is used as the testlist file. If this file is not found, then `$XILCHHOME/config/testlist` is used.
- Test information is logged to `stdout`.
- The file `xilch_tests` in `$XILCHHOME/config` is used as the default test matrix file.
- Terse test information is provided.
- Images are tested against reference signatures.

```
% Xilch -f mytests -l mytests.log -v
```

`Xilch` is invoked with the following conditions:

- The file `mytests` in the current directory is used as the testlist file.
- Test information is logged to the file `mytests.log`.
- The file `xilch_tests` in `$XILCHHOME/config` is used as the default test matrix file.
- Verbose test information is provided.
- Images are tested against reference signatures.

```
% Xilch -create_signatures -platform my_accelerator -m my_list
```

`Xilch` is invoked with the following conditions:

- The file `testlist` in the current directory is used as the testlist file. If this file is not found, then `$XILCHHOME/config/testlist` is used.
- Reference signatures are created for the platform `my_accelerator` and stored in a directory for that platform; no comparisons are made.
- The file `my_list` is used as the test matrix file.
- Information is logged to `stdout`. When you run `Xilch` to create references, a log file is created even though no verification of data is performed.

```
% Xilch -test_data
```

`Xilch` is invoked with the following conditions:

- The file `testlist` in the current directory is used as the testlist file. If this file is not found, then `$XILCHHOME/config/testlist` is used.
- Test information is logged to `stdout`.
- The file `xilch_tests` in `$XILCHHOME/config` is used as the default test matrix file.
- Terse test information is provided.
- Images are tested against reference data.

```
% Xilch -t rotate_test -v
```

`Xilch` is invoked with the following conditions:

- The test program called `rotate_test` is run. No other test programs are run.
- Test information is logged to `stdout`.
- The file `xilch_tests` in `$XILCHHOME/config` is used as the default test matrix file.
- Verbose test information is provided.
- Images are tested against reference signatures.

```
% Xilch -f mytests -m quick_tests -test_data -v
```

`Xilch` is invoked with the following conditions:

- The file `mytests` in the current directory is used as the testlist file.
- Test information is logged to `stdout`.
- The file `quick_tests` is used as the test matrix file.
- Verbose test information is provided.
- Images are tested against reference data.

Error Messages

You will encounter the error messages listed in Table 2-3 if `Xilch` cannot find the reference directory, or cannot find a signature or data in the reference directory. These error messages are sent to the log file.

Table 2-3 `Xilch` Error Messages

Error Message	Explanation
`Cannot access the reference directory. Exiting.`	`Xilch` cannot find the reference directory.
`Could not load ref crc (VERIFY)`	`Xilch` cannot find a specific reference signature.
`Could not find ref image (VERIFY)`	`Xilch` cannot find a specific reference data.

Running a Test Program Without Running `Xilch`

All of the supplied test programs can be run without using the `Xilch` harness. You might find this mode useful for debugging new test programs that you create to verify code that you write to work with the XIL library. You might also find this mode useful to exercise your version of an XIL function, because the existing `Xilch` tests do a good job of checking a given operation's functionality.

All of the `Xilch` options discussed in this chapter (besides those mentioned in this section) can be used when running test programs.

Additional Options For Running Individual Tests

The following XIL Test Suite options provide for tolerance in image comparisons, allowing small hardware arithmetic rounding errors in the image algorithms:

- `-abs_tol`
- `-rel_tol`
- `-nbhd`

Use the following reporting options with the tolerance options:

- `-detail_info`

- `-save_images`

In addition, you must use the `Xilch` harness `-v` option, which is enhanced to report information specific to the above tolerance options.

Terms and Individual Test Option Descriptions

Individual test options assume you are familiar with the terms in Table 2-4.

Table 2-4 Terms for Comparing Image Tolerances

Term	Description
abs_error	This value is computed as the absolute value of the difference between the reference (ref) and image (im) pixel values: abs_error = abs(im - ref) If the value of abs_error is greater than abs_error_max, abs_error, position, new image, and reference image values are reported.
rel_error	This value is computed using neighborhood average: rel_error = abs_error / (neighborhood_average + 1.0) If the value of rel_error is greater than rel_error_max, rel_error, abs_error_at_rel_error_max, (x,y) position, image, and reference pixel values are reported. abs_error and rel_error are reported to the detail file with (x,y) position up to `MAXERRORS`. `MAXERRORS` is a constant currently set to 64*64*64*4.

Table 2-5 describes the individual test options.

Table 2-5 Individual Test Options

Option	Description
`-abs_tol` *value*	Uses *value*, a floating point number, to compare the absolute pixel value of an image and a reference image to within a specified absolute degree of tolerance. This option generally is useful for determining the success or failure of image comparisons with hardware acceleration. Test failure occurs if abs_error ever exceeds `-abs_tol` *value*.
`-detail_info` *dir*	Uses the directory *dir* to report the first `MAXERRORS` errors for abs_error and rel_error.
`-nbhd` *n*	Uses the unsigned int *n* to set the neighborhood size parameter for use with the `-rel_tol` option, so that the base value is the intensity average for a `nbhd`-sized as (2n+1)x(2n+1) and centered on (x,y). This option helps prevent small values for the reference image (x,y) from causing inordinately large values for rel_error. It is useful for improving `-rel_tol` for a test generating a large number of errors due to a wide range of pixel-value differences or for fine-tuning a previous test run using `-rel_tol`. Recommended values: 0, 1 (default is 0).
`-rel_tol` *value*	Uses *value*, a floating point number, to normalize abs_error to the reference pixel value or neighborhood average of the reference pixel value. Neighborhood averaging prevents unusually small base values from causing rel_error to artificially exceed tolerance values. This option is very useful for testing XIL functions involving geometric transformations, multiplications (for example, convolution), and decompression. The normalization is not to any fixed absolute scale (such as a percentage value), since the ratio base is added to 1.0 before dividing to prevent division by 0. The exact level of `-rel_tol` has no significance other than for relative comparison purposes. Test failure occurs if rel_error ever exceeds the value of `-rel_tol`. Combining `-abs_tol` and `-rel_tol` provides added flexibility when comparing images, because it requires both options to exceed tolerances simultaneously to determine a test failure. This technique filters spurious abs_error or rel_error values by the other option, allowing a stronger comparison criterion to be used.

Table 2-5 Individual Test Options *(Continued)*

Option	Description
`-save_images` *dir*	Uses the *dir* directory to save the compared images, the verify test image, the absolute error image, and the absolute error threshold image. This option allows the tester to visualize test images using an image visualization tool or otherwise manipulate test objects (CIS, LUT) beyond the test program. It returns information similar to what the `-detail_info` option returns. This option is used only in `VERIFY` run mode. It is added to all tests using the `-abs_tol` option. Conventions for naming files are: `v.`*n* -verify test image `c0.`*n* -compare image0 `c1.`*n* -compare image1 `e.`*n* -absolute error image `t.`*n* -absolute error threshold image `s.`*n* -default test image where *n* is the sequential numbering of the file in the save directory determined by the counter for the image type.
`-v`	This option is an enhancement of the `Xilch` verbose option (described in Table 2-3). It reports max_abs_error (`-abs_tol`), max_rel_error (`-rel_tol`), rel_error_at_abs_error_max, and abs_error_at_max_rel_error (`-abs_tol` and `-rel_tol`).

Note – The `Xilch` options `-create_signatures` and `-test_signatures` have no meaning when used with the tolerance options described in this section.

Using the Tolerance Options

Individual tests are usually executed with the `-detail_info` *dir* option as well as one or more of the following `Xilch` harness options:

- `-create_data`
- `-m` *file*
- `-ref_directory` *dir*
- `-test_data`
- `-v`

Because the `-detail_info` *dir* option is an individual test option rather than a `Xilch` harness option, it must be set on an individual command line for each test you run. This option allows you to specify a separate directory name for each test to avoid overwriting the detail results files.

Note – Overwriting a results file still occurs for the same test when that test is re-run.

Invocation of A Single Test From A Command Line

This example runs the `affine_test` on general interpolation for byte images and saves the data so that the results can be viewed:

```
affine_test -v -test_data -i general \
-detail_info /export/xilch/logs/detail/general \
-save_images /export/xilch/logs/err_imgs/general
```

Note – The '\' indicates continuation of the command line.

The `detail_info` directory (`/export/xilch/logs/detail/general` in this example) gathers a text file report of the errors. The `save_images` directory (`/export/xilch/logs/err_imgs/general`) creates the `*.vff` images for visual inspection.

Using a Test List File

If you want to run several individual tests, you can use the tolerance options with the `Xilch -f` option, for example

```
% Xilch -f my_testlist
```

In this case, the `my_testlist` file would contain a list of the individual tests to be executed with the tolerance options and their values. The format of each test would be the same as if you had invoked that individual test from the command line. See "Invocation of A Single Test From A Command Line" for an example of how an individual test entry might appear in the `my_testlist` file.

2

Writing Test Programs in the XIL Test Suite Environment 3

This chapter provides the information you need to write test programs to verify new functionality (for example, a new molecule) that you have added to the XIL library. For information on how to add functionality to the XIL library, see the *XIL Device Porting and Extensibility Guide*.

Example Test Program

As described in Chapter 2, “Running Test Programs in the XIL Test Suite,” all test programs use calls to the XIL library and to the XIL Test Suite library (`libts`) to verify that test programs produce signatures or data identical to the stored reference signatures or data. The functions available in `libts` are described in detail later in this chapter.

The simple example of a test program in Code Example 3-1 gives you an idea of what such a program should do. The program `image_example_test.c` tests the XIL library function `xil_add()`. The full text of this program is in Appendix B, “Example Test Program.” The program itself is located in the `$XILCHHOME/../src/tests/examples` directory.

After the usual declarations (notice that a tolerance of 0.0 is specified, meaning that results from the test program must match the stored reference), the first thing any test program must do is open the XIL library with `xil_open()` and the XIL Test Suite library with `ts_init()`, and start the test case with `ts_start_test_case()`.

Code Example 3-1 Test Program Example: Initializing the Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xil/xil.h>
#include <xilch/ts.h>


main (
int argc,
char**argv )
{
 int       i;
 int       errors = 0;
 int       total_errors = 0;
 char      *specfile = NULL;
 float     tol = 0.0;
 XilImage      src1, src2, dst1;
 XilSystemState              xil_State;


 /*
 * process command line arguments
 */

 for (i = 1; i < argc; i++) {
    if (!strcmp (argv[i], “-m”)) {
        specfile = argv[++i];
    }
    if (!strcmp (argv[i], “-tol”)) {
        tol = atof (argv[++i]);
    }
 }

 /*
 * initialize XIL and the test suite library
 */
```

Code Example 3-1 Test Program Example: Initializing the Libraries *(Continued)*

```
 if ((xil_State = xil_open ()) == (XilSystemState) NULL) {
    fprintf (stderr, “Error initializing XIL\n”);
    exit (1);
 }

 if (ts_init (&argc, argv, xil_State) == TS_FAILURE) {
    exit (1);
 }
ts_start_case(“image_example_test”);
```

Next, the program retrieves the images it needs for testing with calls to `ts_get_src[123]_image` and `ts_get_dst[123]_image`.

```
 /*
 * get some test images
 */

 src1 = ts_get_src1_image (specfile);
 src2 = ts_get_src2_image (specfile);
 dst1 = ts_get_dst1_image (specfile);
```

These functions retrieve test images as specified in the test matrix file `specfile`. If a file name had not been specified (`specfile` == NULL), the default test matrix file, `xilch_tests`, would have been used.

After a call to the XIL function you are testing (in this case, `xil_add()`), the next step is to verify your results with a call to `ts_verify()`. When the test program gets to the `ts_verify()` call, the test signature or data is compared to the stored reference.

This part of the program does one other thing. It contains a loop to call a series of images, rather than just one set of images. The loop continues to retrieve images from `specfile`, the test matrix file, until all the images in that file have been retrieved.

Code Example 3-2 contains the loop with the call to `ts_verify()`. Note that the function’s *comment* parameter does not have to be NULL. (This parameter puts a comment in the log file.)

Code Example 3-2 Test Program Example: Looping over Images in the Test Matrix File

```
/*
 * loop over the images in the test matrix file
 */

 while ( (src1 != (XilImage) NULL) &&
     (src2 != (XilImage) NULL) &&
     (dst1 != (XilImage) NULL) ) {

    /*
     * do the operation(s), add two images into a destination
     */

    xil_add (src1, src2, dst1);

    /*
     * verify the results
     */

    errors = ts_verify (“add”, NULL, dst1, tol, NULL);

    if (errors != 0) {

        /*
        * save the src and dst images for further study
        */

        ts_save (src1, “add_src1”);
        ts_save (src2, “add_src2”);
        ts_save (dst1, “add_dst”);

        total_errors++;

        exit (total_errors);
    }

    /*
     * get rid of the images
     */

    ts_destroy (src1);
```

Code Example 3-2 Test Program Example: Looping over Images in the Test Matrix File *(Continued)*

```
    ts_destroy (src2);
    ts_destroy (dst1);

    /*
     * get some more test images
     */

    src1 = ts_get_src1_image (specfile);
    src2 = ts_get_src2_image (specfile);
    dst1 = ts_get_dst1_image (specfile);

  }
```

The final part of the program ends the test case, and closes the XIL Test Suite library and the XIL library.

```
  /*
  * return the total number of errors (required) and close XIL
  */
  ts_end_test_case();
  xil_close (xil_State);

  total_errors = ts_end ();

  exit (total_errors);
}
```

As you have seen, this example program gives you a basic idea of the kinds of things you need to include in test programs you might develop. These include:

1. Calls to initialize the relevant libraries.
2. Calls to retrieve the images that will be used.
3. Calls to the functionality you are testing.
4. Calls to `ts_verify()` to compare the results of your test program to the stored reference.

5. Calls to close the relevant libraries.

Available XIL Test Suite Library Functions

The XIL Test Suite library provides many more functions than the ones used in this example program.Table 3-1 lists all the functions available in `libts` by function category. Syntax and usage of these functions are described in detail at the end of this chapter.

Table 3-1 Functions Available in the XIL Test Suite Library (`libts`) *(1 of 3)*

Function Category	Function Name
General Functions	`ts_init()`
	`ts_end()`
	`ts_benchmark_start()`
	`ts_benchmark_end()`
	`ts_expect_errors()`
	`ts_log()`
	`ts_start_test_case()`
	`ts_end_test_case()`
Image Functions	`ts_verify()`
	`ts_getref()`
	`ts_compare()`
	`ts_image_gen()`
	`ts_load()`
	`ts_save()`
	`ts_checksum()`

	`ts_get_src1_image()`
	`ts_get_src2_image()`
	`ts_get_src3_image()`
	`ts_get_dst1_image()`
	`ts_get_dst2_image()`
	`ts_get_dst3_image()`

	`ts_get_complex_roi()`

Table 3-1 Functions Available in the XIL Test Suite Library (`libts`) *(2 of 3)*

Function Category	Function Name
	`ts_destroy()`
	`ts_display()`
Lookup Table Functions	`ts_lookup_verify()`
	`ts_lookup_verify_contents()`
	`ts_lookup_getref()`
	`ts_lookup_compare()`
	`ts_lookup_compare_contents()`
	`ts_lookup_contents_checksum()`
	`ts_lookup_load()`
	`ts_lookup_save()`
CIS Functions	`ts_cis_verify()`
	`ts_cis_getref()`
	`ts_cis_compare()`
	`ts_cis_checksum()`
	`ts_cis_load()`
	`ts_cis_save()`
	`ts_cis_destroy()`
Float and Integer Functions	`ts_float_verify()`
	`ts_float_getref()`
	`ts_float_load()`
	`ts_float_save()`
	`ts_int_verify()`
	`ts_int_getref()`
	`ts_int_load()`
	`ts_int_save()`

Table 3-1 Functions Available in the XIL Test Suite Library (`libts`) *(3 of 3)*

Function Category	Function Name
Equivalence Testing Functions	`ts_automatic_tests()`
	`ts_verify_needed()`
	`ts_child_check()`
	`ts_roi_check()`

Other Useful Examples

In the same directory with `image_test_example.c` are two other examples that might be useful:

- `$XILCHHOME/../src/tests/examples/example_bench.c`

  A simple benchmarking program

- `$XILCHHOME/../src/tests/examples/Makefile`

  An example Makefile

The `examples` subdirectory also includes the source code for two typical test programs: `fax_test.c` and `rotate_test.c`.

See the section "*Writing and Using Benchmarking Programs*" for details on benchmarking programs.

After You Write Your Test Program

When you write your own test program, you cannot use the references shipped with the XIL Test Suite to verify XIL functions; you must create your own references that correspond to the new test program.

Unless you use the image data provided in the `xilch_tests` test matrix file, you will need to create your own test matrix for your test program to use. This data must conform to the test matrix file format, which is described in Chapter 2, "Running Test Programs in the XIL Test Suite."

After you have created your test matrix file and written your test program, you should run your program alone (not under `Xilch`) to debug it. This procedure is described in Chapter 2, "Running Test Programs in the XIL Test Suite."

To build your test program, you must link with the following libraries: `libts`, `libfileio`, `libvff`, `libxil`, and `libm`. For an example, see the `Makefile` in `$XILCHHOME/../src/tests/examples`.

Writing and Using Benchmarking Programs

You can write benchmarking programs with XIL Test Suite library functions that will measure the performance, in frames per second, of a given XIL function. As mentioned earlier, source for an example benchmarking program is located in `$XILCHHOME/../src/tests/examples/example_bench.c`.

Here's the code for a simple benchmarking function that tests the performance of `xil_threshold()`:

```
/*
 * benchmark loop
 */
ts_benchmark_start ("threshold");
for (j = 0; j < iterations; j++) {
    xil_threshold (src1, src1, lowvalue, highvalue, mapvalue);
    xil_sync (src1);
}
ts_benchmark_end (iterations, comment);
```

The `libts` function `ts_benchmark_start()` is used to get the starting time of the program, and `ts_benchmark_end()` gets the time the program ended, and calculates how many frames per second `xil_threshold()` takes to run.

You get more accurate results if you specify a fairly high number for *iterations*, because this lessens the impact of program overhead. You specify the number of iterations on the command line. The `xil_sync()` call is needed to prevent the operation from being deferred.

Running Your Benchmarking Program

You can run your benchmarking program in several ways. The simplest way is to run it outside of `Xilch`. For example, if your program name were `bench` and you wanted to run it for 20 iterations, you would type the following at the command line:

```
% bench -I 20
```

The output gives you the performance of the function that `bench` tests.

Or you could run your program through `Xilch`. To do this, type:

```
% Xilch -t "bench -I 20"
```

The output gives you the performance of the function that `bench` tests.

Running your benchmarking programs through `Xilch` gives you one advantage over running the programs outside of `Xilch`—you can create a testlist file containing all your benchmarking programs and use the `-f Xilch` option to run that file. For example, suppose you had a testlist file called `benches` that contained your benchmarking programs. To run them all through `Xilch`, type:

```
% Xilch -f benches
```

Of course, you could vary the number of iterations by specifying them for individual programs in your testlist file.

Using Equivalence Testing Functions

The XIL Test Suite library provides a set of functions that enables the test writer to test certain XIL features (such as child images and ROIs) without requiring any visual verification. These functions are:

- `ts_child_check()`
- `ts_roi_check()`
- `ts_automatic_tests()`

These functions produce a "reference result" and a "test result." (These functions do not test against reference signatures or data). These two results should be identical if the XIL feature under test is working correctly. These

functions use the FUNCFORMAT structure to receive descriptions of XIL operations. See Appendix C, “Equivalence Testing Example,” for an example of how to use these functions.

The Test Suite Library

The Test Suite library, `libts`, provides functions to aid the test program writer in activities such as verifying images, generating test images, and logging test information. These functions are divided into categories (general, image, lookup table, CIS, float and integer, and equivalence testing functions) and are described on the following pages.

Include Files

The file `<xilch/ts.h>` must be included after `<xil/xil.h>`.

General Functions

ts_init

```
int ts_init (int *argc, char **argv, XilSystemState xil_state);
```

Description

`ts_init()` initializes the Test Suite library. It handles the command line arguments specific to the Test Suite library, which include the options `-f`, `-r`, and `-t`. This function must be called before any other Test Suite functions and after `xil_open(3)` has been called. If it is successful, `ts_init()` returns `TS_SUCCESS`. Otherwise, it returns `TS_FAILURE`.

Parameters

argc	The number of command-line arguments the program was invoked with.
argv	A pointer to an array of character strings that contain the arguments, one per string.
xil_state	The system state created when `xil_open` is invoked

ts_end

```
int ts_end ();
```

Description

`ts_end()` releases the Test Suite library and all of its resources. It should be called at the end of the test program. If it is successful, `ts_end()` returns `TS_SUCCESS`; otherwise, it returns `TS_FAILURE`.

Note – This function does not destroy images when it is called. You must call `ts_destroy()` to destroy images.

---

ts_benchmark_start

```
int ts_benchmark_start (char *funcname);
```

Description

`ts_benchmark_start()` marks the beginning of benchmark code and starts a timer. Appropriate entries are made in the log file. This function returns the time it started.

Parameters

funcname	A string containing the name of the function being benchmarked.

ts_benchmark_end

`int ts_benchmark_end (int` *iterations*`, char *`*comment*`);`

Description

`ts_benchmark_end()` marks the end of benchmark code. Appropriate entries are made in the log file. The function `ts_benchmark_end()` returns the elapsed time since the last call to `ts_benchmark_start()`.

Parameters

iterations	The number of command-line arguments the program was invoked with.
comment	A string containing a comment written to the log file if verbose mode is on. The string can be NULL.

ts_expect_errors

`void ts_expect_errors (int` *linenum*`, char *`*error_str*`, char *`*error_str, ...,* `NULL);`

Description

`ts_expect_errors()` installs an XIL error handler and begins looking for the “expected” errors to occur.

Parameters

This function accepts a variable number of arguments terminated by a NULL.

linenum	The current line number. Usually this parameter is the macro `__LINE__`.
error_str	XIL error strings that identify expected errors. These strings are of the form `di-`*number*.

ts_log

```
int ts_log (char *comment);
```

Description

`ts_log()` writes the supplied comment to the log file if verbose mode is on. It returns `TS_SUCCESS` if it is successful; otherwise, it returns `TS_FAILURE`.

Parameters

comment	A string containing a comment written to the log file if verbose mode is on. The string can be NULL.

ts_start_test_case

```
int ts_start_test_case (char *test_case_name);
```

Description

`ts_start_test_case()` starts a test case.

Parameters

test_case_name	The name of the test case. The name must be unique in the Test Suite.

ts_end_test_case

```
void ts_end_test_case ();
```

Description

`ts_end_test_case()` ends a test case.

Image Functions

ts_verify

```
int ts_verify (char *funcname, char *comment, XilImage im,
float tolerance, char *ref_name);
```

Description

`ts_verify()` verifies a test program against stored reference signatures or data. This function returns the number of differences it finds.

Note – As a debugging aid, the environment variable `XILCHSAVE` can be used to save the two images that were verified against each other and the last source images retrieved via `ts_get_src[123]_image()`. The format of `XILCHSAVE` is a list of numbers separated by a colon, where the numbers represent the images that are a result of calls to `ts_verify()` that you want to save. The count starts at 0. For example, if you had 5 calls to `ts_verify()` and you wanted to save the images from the first and fourth calls, `XILCHSAVE` would look like this: 0:3.

Parameters

funcname	A string containing the name of the function that generated the image to be verified.
comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
im	The name of the image to be verified.
tolerance	Specifies a tolerance to be used in the comparison.
ref_name	Specifies the reference name that the library uses. If *ref_name* is NULL, the library uses a default name for the reference. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_getref()` is used (see `ts_getref()`). *ref_name* must be unique within the test case. The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* will increment the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` is run in verbose mode, reference names are printed to the log file.

ts_getref

```
XilImage ts_getref ();
```

Description

`ts_getref()` retrieves reference data, if any. It returns `NULL` if there is no reference data. The reference data it retrieves is the data named by the current default reference name—the reference name used by the most recent call to `ts_*_verify()` in the current test case. See `ts_verify()` for an explanation

of reference names. If you call `ts_getref()` before calling `ts_verify()` in the current test case, `NULL` is returned. You cannot call `ts_getref()` outside a test case.

ts_compare

`int ts_compare (XilImage` *im1*`, XilImage` *im2*`, float` *tolerance*`);`

Description

`ts_compare()` compares two images. It returns the number of differences it finds.

Parameters

im1 and *im2*	The names of the images to be compared.
tolerance	Specifies a tolerance to be used in the comparison.

ts_image_gen

`int ts_image_gen (XilImage` *im*`, int` *type*`, int *`*value*`);`

Description

`ts_image_gen()` generates an image from a set of predefined image types.

Parameters

im	The previously created image handle. `ts_image_gen()` returns `TS_SUCCESS` if it is successful; otherwise, it returns `TS_FAILURE`.
type	One of the following: TS_RANDOM, TS_CONSTANT, or TS_RAMP.
value	A pointer to a vector that specifies the random number seed for TS_RANDOM, or the constant value for TS_CONSTANT. *value* is not used with TS_RAMP. This vector should match in size with the number of bands in the image *im*.

ts_load

```
XilImage ts_load (char *filename);
```

Description

`ts_load()` loads the image from a file and returns a handle to that image if successful. Otherwise, it returns `NULL`. Currently, this function only handles images in .vff format. This is an unsupported, Sun-internal file format. The file format accepted by this function may change in future releases.

Parameters

filename	The name of the file from which `ts_load()` loads the image.

ts_save

```
int ts_save (XilImage im, char *filename);
```

Description

`ts_save()` saves an image into a file. Images are saved in .vff format. If this function runs successfully, it returns the value `0` (zero); otherwise, it returns the value `1`.

Parameters

im	The name of the image being saved.
filename	The name of the file into which `ts_save()` saves the image.

ts_checksum

```
unsigned int ts_checksum (XilImage im);
```

Description

`ts_checksum()` returns a 32-bit CRC code for an image.

Parameters

im	The name of the image on which to perform a checksum calculation.

ts_get_src[123]_image and ts_get_dst[123]_image

```
XilImage ts_get_src1_image (char *filename);
XilImage ts_get_src2_image (char *filename);
XilImage ts_get_src3_image (char *filename);
XilImage ts_get_dst1_image (char *filename);
XilImage ts_get_dst2_image (char *filename);
XilImage ts_get_dst3_image (char *filename);
```

Description

`ts_get_src[123]_image()` and `ts_get_dst[123]_image` retrieve test images from a test matrix file. `ts_get_src[123]_image` and `ts_get_dst[123]_image` return images each time they are called until there are no more images specified in the test matrix file, at which time they return NULL. The next call after a returned NULL restarts the cycle of images (in other words, a return of NULL signals the end of a cycle).

Parameters

filename	Specifies the name of the test matrix file to use. If *filename* is NULL, then the default test matrix file `xilch_tests` is used.

ts_get_complex_roi

```
XilRoi ts_get_complex_roi (int xsize, int ysize);
```

Description

`ts_get_complex_roi` returns a ROI object. After you have finished using the ROI, use the XIL function `xil_roi_destroy()` to delete the ROI.

Parameters

xsize and *ysize*	Dimensions of the image to which you want to attach the ROI.

ts_destroy

```
int ts_destroy (XilImage im);
```

Description

Use this function to destroy any image created by the Test Suite library, because it also destroys the parent image if the passed image is a child image. If it is successful, `ts_destroy()` returns `TS_SUCCESS`. Otherwise, it returns `TS_FAILURE`.

Parameters

im	Name of the image to destroy.

ts_display

```
void ts_display (XilImage im);
```

Description

This function displays the image specified.

Parameters

im	Name of the image to display.

Lookup Table Functions

ts_lookup_verify

```
int ts_lookup_verify (char *funcname, char *comment,
XilLookup lut, float tolerance, char *ref_name);
```

Description

`ts_lookup_verify()` verifies a lookup table (LUT) generated by the test program against the corresponding stored reference. It returns the number of differences it finds. This function is sensitive to the position of the entries in colormaps. In other words, if entry [i] in one colormap is BGR, then entry [i] in the other colormap must also be BGR. Otherwise, the function returns the differences resulting from entry position differences.

See also `ts_lookup_verify_contents()`.

Parameters

funcname	A string containing the name of the function that generated the LUT to be verified.
comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
lut	The LUT to be verified.
tolerance	Specifies a tolerance to be used in the comparison. For an exact comparison, use a tolerance of 0.0.
ref_name	Specifies the reference name that the library uses. If *ref_name* is NULL, the library uses a default name. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_lookup_getref()` is used. *ref_name* must be unique within the test case. The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* increments the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` runs in verbose mode, reference names are printed to the log file.

ts_lookup_verify_contents

`int ts_lookup_verify_contents (char *`*funcname*`, char *`*comment*`, XilLookup` *lut*`, float` *tolerance*`, char *`*ref_name*`)`

Description

`ts_lookup_verify_contents()` verifies a lookup table (LUT) generated by the test program against the corresponding stored reference. It returns the number of differences it finds. This function is *not* sensitive to the position of the entries in colormaps. In other words, if entry [i] is BGR and entry [j] is bgr

in one colormap, and entry [i] is bgr and entry [j] is BGR in the other colormap, and all other entries in the two colormaps match, then this function will not find any differences. `ts_lookup_verify()` would find differences in this case.

Parameters

funcname	A string containing the name of the function that generated the LUT to be verified.
comment	A string containing a comment written to the log file.
lut	The LUT to be verified.
tolerance	Specifies a tolerance to be used in the comparison. For an exact comparison, use a tolerance of 0.0.
ref_name	Specifies the reference name that the library uses. If *ref_name* is NULL, the library uses a default name. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_lookup_getref()` is used. *ref_name* must be unique within the test case. The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* increments the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` runs in verbose mode, reference names are printed to the log file.

ts_lookup_checksum

```
unsigned int ts_lookup_checksum (XilLookup lut);
```

Description

`ts_lookup_checksum()` returns a 32-bit CRC code for a LUT. This function performs a checksum calculation. Unlike `ts_lookup_contents_checksum()`, it does not produce identical CRC codes for lookups that have the same entries but with different ordering. The orderings must be the same to get the same CRC codes. `ts_lookup_checksum()` corresponds to `ts_lookup_verify()`.

Parameters

lut	The LUT on which to perform a checksum calculation.

ts_lookup_contents_checksum

```
unsigned int ts_lookup_contents_checksum (XilLookup lut);
```

Description

`ts_lookup_contents_checksum()` returns a 32-bit CRC code for a LUT. This function sorts the lookup table specified and then performs a checksum calculation. It produces identical CRC codes for lookups that have the same entries but with different ordering. `ts_lookup_contents_checksum()` corresponds to `ts_lookup_verify_contents()`.

Parameters

lut	The LUT on which to perform a checksum calculation.

ts_lookup_getref

```
XilLookup ts_lookup_getref ();
```

Description

`ts_lookup_getref()` retrieves reference data, if any. It returns `NULL` if there is no reference data. The reference data it retrieves is the data named by the current default reference name—the reference name used by the most recent call to `ts_*_verify()` in the current test case. See `ts_verify()` for an explanation of reference names. If you call `ts_lookup_getref` before calling `ts_lookup_verify` in the current test case, `NULL` is returned. You cannot call `ts_lookup_getref` outside a test case.

ts_lookup_compare

`int ts_lookup_compare (XilLookup` *lut1*`, XilLookup` *lut2*`, float` *tolerance*`);`

Description

`ts_lookup_compare()` compares two lookup tables. It returns the number of differences it finds. This function is sensitive to the position of the entries in colormaps. In other words, if entry [i] in one colormap is BGR, then entry [i] in the other colormap must also be BGR. Otherwise, the function returns the differences resulting from entry position differences.

See also `ts_lookup_compare_contents()`.

Parameters

lut1 and *lut2*	The LUTs to compare.
tolerance	Specifies a tolerance to use in the comparison.

ts_lookup_compare_contents

`int ts_lookup_compare_contents (XilLookup` *lut1*`, XilLookup` *lut2*`, float` *tolerance*`);`

Description

`ts_lookup_compare_contents()` compares two lookup tables. It returns the number of differences it finds. This function is *not* sensitive to the position of entries in colormaps. In other words, if entry [i] is BGR and entry [j] is bgr in one colormap, and entry [i] is bgr and entry [j] is BGR in the other colormap, and all other entries in the two colormaps match, then this function will not find any differences. `ts_lookup_compare()` would find differences in this case.

Parameters

lut1 and *lut2*	The LUTs to compare.
tolerance	Specifies a tolerance to use in the comparison.

ts_lookup_load

`XilLookup ts_lookup_load (char *`*filename*`);`

Description

`ts_lookup_load()` loads the lookup table (LUT) from a file and returns a handle to that LUT if it is successful. Otherwise, it returns `NULL`.

Parameters

filename	The name of the file from which `ts_lookup_load()` loads the LUT.

ts_lookup_save

```
int ts_lookup_save (XilLookup lut, char *filename);
```

Description

`ts_lookup_save()` saves a lookup table into a file. If this function runs successfully, it returns the value `1`. Otherwise, it returns the value `0` (zero).

Parameters

lut	The name of the lookup table being saved.
filename	The name of the file into which `ts_lookup_save()` saves the LUT.

CIS Functions

ts_cis_verify

```
int ts_cis_verify (char *funcname, char *comment, XilCis cis,
float tolerance, char *ref_name);
```

Description

`ts_cis_verify()` verifies a compressed image sequence (CIS) generated by the test program against the corresponding stored reference. This function returns the number of differences it finds.

Parameters

funcname	A string containing the name of the function that generated the CIS to verify.
comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
cis	The CIS to verify.
tolerance	Specifies a tolerance to use in the comparison.

ref_name Specifies the reference name that the library uses. If *ref_name* is NULL, the library uses a default name. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_cis_getref()` is used. *ref_name* must be unique within the test case.

The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* will increment the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` is run in verbose mode, reference names are printed to the log file.

ts_cis_getref

```
XilCis ts_cis_getref ();
```

Description

`ts_cis_getref()` retrieves the named reference data, if any. It returns `NULL` if there is no reference data. The reference data it retrieves is the data named by the current default reference name—the reference name used by the most recent call to `ts_*_verify()` in the current test case. See `ts_verify()` for an explanation of reference names. If you call `ts_cis_getref()` before calling `ts_cis_verify()` in the current test case, `NULL` is returned. You cannot call `ts_cis_getref()` outside a test case.

ts_cis_compare

`int ts_cis_compare (XilCis` *cis1*`, XilCis` *cis2*`, float` *tolerance*`);`

Description

`ts_cis_compare()` compares two CISs. This function returns the number of differences it finds.

Parameters

cis1 and *cis2*	The CISs to be compared.
tolerance	Specifies a tolerance to be used in the comparison.

ts_cis_checksum

`unsigned int ts_cis_checksum (XilCis` *cis*`);`

Description

`ts_cis_checksum()` returns a 32-bit CRC code for a CIS.

Parameters

cis	The CIS on which to perform a checksum calculation.

ts_cis_load

`XilCis ts_cis_load (char *`*cis_type*`, char *`*filename*`, int` *partial*`, int` *copy*`, int` *nbytes*`);`

Description

`ts_cis_load()` loads the compressed image sequence (CIS) from a file and returns a handle to that CIS if successful. Otherwise, it returns `NULL`.

Parameters

cis_type	A string that describes the type of compression, for example, “Mpeg1” or “JpegLL.”
filename	The name of the file from which `ts_cis_load()` loads the CIS.
partial	Indicates that there are partial frames in the bitstream.
copy	Loads a copy of the CIS when the value is `1`. When the value is `0` (zero), it returns a pointer to the CIS.
nbytes	Specifies the number of bytes the function loads. This parameter needs to be specified only if the function is loading partial bitstreams.

ts_cis_save

```
int ts_cis_save (XilCis cis, char *filename);
```

Description

`ts_cis_save` saves the compressed image sequence (CIS) into a file. If this function runs successfully, it returns the value `1`. Otherwise, it returns the value `0` (zero). If `ts_cis_save()` runs when verbose mode is on, it prints the number of frames and number of bytes it saved.

Parameters

cis	The CIS to save.
filename	The name of the file into which `ts_cis_save()` saves the CIS.

ts_cis_destroy

```
int ts_cis_destroy (XilCis cis)
```

Description

`ts_cis_destroy()` destroys a CIS. You should use this function only to destroy a CIS that was created by `ts_cis_load()`. If this function runs successfully, it returns the value `1`. Otherwise, it returns the value `0` (zero).

Parameters

cis	The CIS to destroy.

Float and Integer Functions

ts_float_verify

```
int ts_float_verify (char *funcname, char *comment,
float *fvec, int vecsize, float tolerance, char *ref_name);
```

Description

`ts_float_verify()` verifies a floating point vector generated by the test program against the corresponding stored reference. This function returns the number of differences it finds.

Parameters

funcname	A string containing the name of the function that generated the floating point vector to verify.
comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
fvec	A pointer to the floating point vector to be verified.

vecsize	The size of the floating point vector.
tolerance	Specifies a tolerance to use in the comparison.
ref_name	Specifies the reference name that the library will use. If *ref_name* is NULL, the library uses a default name. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_float_getref()` is used. *ref_name* must be unique within the test case. The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* will increment the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` is run in verbose mode, reference names are printed to the log file.

ts_float_getref

```
float *ts_float_getref (unsigned int *);
```

Description

`ts_float_getref()` returns the number of items in the array of the unsigned int argument pointer. You cannot call `ts_float_getref()` outside a test case.

ts_float_load

```
float *ts_float_load (char *filename, unsigned int *nvalues);
```

Description

`ts_float_load()` loads a floating point vector from a file and returns a pointer to the float if successful. Otherwise, it returns `NULL`.

Parameters

filename	The name of the file from which `ts_float_load()` loads the floating point vector.
nvalues	The number of values in the vector.

ts_float_save

`int ts_float_save (float *`*vector*`, unsigned int` *nvalues*`, char *`*filename*`);`

Description

`ts_float_save()` saves a floating point vector into a file. If this function runs successfully, it returns the value `1`; otherwise, it returns the value `0` (zero).

Parameters

vector	A pointer to the floating point vector.
nvalues	The number of values in the vector.
filename	The name of the file into which `ts_float_save()` saves the floating point vector.

ts_int_verify

`int ts_int_verify (char *`*funcname*`, char *`*comment*`, int *`*ivec*`, int` *vecsize*`, float` *tolerance*`, char *`*ref_name*`);`

Description

`ts_int_verify()` verifies an integer vector generated by the test program against the corresponding stored reference. This function returns the number of differences it finds.

Parameters

funcname	A string containing the name of the function that generated the integer vector to be verified.
comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
ivec	A pointer to the integer vector to be verified.
vecsize	The size of the integer vector.
tolerance	Specifies a tolerance to be used in the comparison.
ref_name	Specifies the reference name that the library will use. If *ref_name* is NULL, the library uses a default name. Explicit naming is useful when a variable number of verifications is performed within a single test case or when `ts_int_getref()` is used. *ref_name* must be unique within the test case. The library assigns default reference names in the following way. The call to `ts_start_ref()` initializes the default reference name to "r.0." Each call to `ts_*_verify()` that has NULL specified as *ref_name* will increment the default reference name. For example, the reference name used in the first call to `ts_*_verify()` that has NULL specified as *ref_name* is "r.1" and the second is "r.2." When you supply a reference name, you should not specify a default name. When `Xilch` is run in verbose mode, reference names are printed to the log file.

ts_int_getref

```
int *ts_int_getref (unsigned int *);
```

Description

`ts_int_getref()` returns the number of items in the array of the unsigned int argument pointer. You cannot call `ts_int_getref()` outside a test case.

ts_int_load

```
int *ts_int_load (char *filename, unsigned int *nvalues);
```

Description

`ts_int_load()` loads an integer vector from a file and returns a pointer to the int if successful. Otherwise, it returns `NULL`.

Parameters

filename	The name of the file into which `ts_int_load()` loads the integer vector.
nvalues	The number of values in the integer vector.

ts_int_save

```
int ts_int_save (int *vector, unsigned int nvalues,
char *filename);
```

Description

`ts_int_save()` saves an integer vector into a file. If this function runs successfully, it returns the value `1`; otherwise, it returns the value `0` (zero).

Parameters

vector	A pointer to the integer vector.
nvalues	The number of values in the integer vector.
filename	The name of the file into which `ts_int_save()` saves the integer vector.

3

Equivalence Testing Functions

ts_child_check

```
int ts_child_check (char *comment, FUNCFORMAT *format_list,
int operation_count, float tol, Display *display);
```

Description

`ts_child_check()` tests the behavior of a child image. The behavior is tested by producing a reference result and a test result and then comparing the test and reference results. A reference result is produced by performing the given operations on specified images. A test result is produced by performing the given operations on src and dst images, which are equivalent to the given images except that they are children of parent images.

Note – The test result should be exactly the same as the reference result.

This function returns the number of errors found. `0` (zero) is returned if the test succeeds. This function cannot be used with `xil_lookup_convert`, `xil_histogram`, and `xil_compress`.

Parameters

comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
format_list	An operation list and can contain more than one operation. Child images are created for the source images of the first operation and for the destination image of the last operation. Other elements of the format list are not used. Comparisons are based on the results of the last operation. `xil_decompress` can be used only as the first operation in the format_list. See Appendix C, “Equivalence Testing Example,” for an explanation of format lists.

operation_count	The number of items in the format list.
tol	The tolerance for comparison. Normally 0.0 is specified, except for molecule testing.
display	A pointer to a structure that is returned when you initially connect to the X server. If a value is specified for *display*, a window is created in which the destination child testing is performed. If `NULL` is specified for *display*, this function uses a memory image instead.

ts_roi_check

```
int ts_roi_check (char *comment, FUNCFORMAT *format_list,
int operation_count, float tol);
```

Description

`ts_roi_check()` tests the behavior of a ROI. The behavior is tested by producing a reference result and comparing this result with the ROI. This function cannot be used with `xil_lookup_convert`, `xil_choose_colormap`, `xil_squeeze_range`, `xil_histogram`, `xil_compress`, or `xil_extrema`.

Parameters

comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
format_list	An operation list and can contain more than one operation. This is for testing molecules. The ROI is placed onto the source images of the first operation and the destination image of the last operation. Other elements of the format list are not used. Comparisons are based on the results of the last operation. `xil_decompress`s can be used only as the first operation in the *format_list*. See Appendix C, “Equivalence Testing Example,” for an explanation of format lists.

operation_count	The number of items in the format list.
tol	The tolerance for comparison. Normally 0.0 is specified, except for molecule testing.

ts_automatic_tests

```
int ts_automatic_tests (char *comment, FUNCFORMAT *format_list,
int operation_count, float tol, Display *display);
```

Description

`ts_automatic_test()` calls all of the equivalence testing functions (`ts_child_check()` and `ts_roi_check()`) if you invoked `Xilch` with either the `-test_signatures` or `-test_data` option. This function returns the total of all of the pixels that differ in all of the automated tests. Equivalence tests do not modify any of the image arguments.

Parameters

comment	A string containing a comment written to the log file if verbose mode is on. It can be NULL.
format_list	An operation list, which can contain more than one operation. See Appendix C, “Equivalence Testing Example,” for an explanation of format lists.
operation_count	The number of items in the format list.
tol	The tolerance for comparison. Normally 0.0 is specified, except for molecule testing.
display	A pointer to a structure that is returned when you initially connect to the X server. If a value is specified for *display*, a window is created in which the destination child testing is performed. If `NULL` is specified for *display*, this function uses a memory image instead.

XIL Test Suite Directory Structure A

The Top Level

The default installation directory for the XIL Test Suite is `/opt` with the hierarchy `SUNWddk/xil/ddk_2.4/Xilch`. The top level of the XIL Test Suite source tree contains the `Xilch` directory. The directory contains a `README` file and several subdirectories:

arch/	*arch* is either `sparc` or `i386` depending on which platform you are running. Subdirectories here contain all the files of the XIL Test Suite library, except the data files.
`data/`	Data files for all platforms.
`src/`	Source code for a few example test programs.

A

The Subdirectories

The contents of the subdirectories follows.

arch/

`bin/`	The `Xilch` program itself and all the test programs
`config/`	Default and customized testlist and test matrix files
`data/`	Symbolic link to `../data`
`include/`	Necessary include files
`lib/`	The test suite library files

`data/`

`images`	Image data files
`movies`	Movie data files
`photo_cd`	Photo CD data files
`references/signatures/generic`	Generic reference signatures
`references/signatures/x86`	x86-specific references (shipped only with x86 packages)

`src/`

`tests/examples`	Contains the source code for five example programs: `Makefile`, `example_bench.c`, `fax_test.c`, `image_example.c`, and `rotate_test.c`

Example Test Program B

This appendix contains the text of the example test program called `image_example_test.c`. The source can be found in the `$XILCHHOME/../src/tests/examples` directory.

Code Example B-1 Example Test Program, `image_example_test.c` *(1 of 4)*

```
/* @(#)image_example_test.c1.4 93/11/19
 *
 * Example image test program that tests xil_add ()
 *
 * Options:
 *  -m <file> use “file” as test specification file
 * -tol # tolerance to use for image comparisons
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xil/xil.h>
#include <xilch/ts.h>


main (
int argc,
char**argv )
```

Code Example B-1 Example Test Program, `image_example_test.c` *(2 of 4)*

```
{
 int       i;
 int       errors = 0;
 int       total_errors = 0;
 char   *specfile = NULL;
 float  tol = 0.0;
 XilImagesrc1, src2, dst1;
 XilSystemStatexil_State;


 /*
 * process command line arguments
 */

 for (i = 1; i < argc; i++) {
    if (!strcmp (argv[i], “-m”)) {
     specfile = argv[++i];
    }
    if (!strcmp (argv[i], “-tol”)) {
     tol = atof (argv[++i]);
    }
 }

 /*
 * initialize XIL and the test suite library
 */

 if ((xil_State = xil_open ()) == (XilSystemState) NULL) {
    fprintf (stderr, “Error initializing XIL\n”);
    exit (1);
 }

 if (ts_init (&argc, argv, xil_State) == TS_FAILURE) {
 exit (1);
 }

 ts_start_test_case(“image_example_test”);

 /*
 * get some test images
 */
 src1 = ts_get_src1_image (specfile);
```

Code Example B-1 Example Test Program, `image_example_test.c` *(3 of 4)*

```
src2 = ts_get_src2_image (specfile);
dst1 = ts_get_dst1_image (specfile);

/*
* loop over the images in the test matrix file
*/

while ( (src1 != (XilImage) NULL) &&
    (src2 != (XilImage) NULL) &&
    (dst1 != (XilImage) NULL) ) {

   /*
    * do the operation(s), add two images into a destination
    */

   xil_add (src1, src2, dst1);

   /*
    * verify the results
    */

   errors = ts_verify (“add”, NULL, dst1, tol, NULL);

   if (errors != 0) {

    /*
    * save the src and dst images for further study
    */

    ts_save (src1, “add_src1”);
    ts_save (src2, “add_src2”);
    ts_save (dst1, “add_dst”);

    total_errors++;

    exit (total_errors);
   }

   /*
    * get rid of the images
    */
   ts_destroy (src1);
```

Code Example B-1 Example Test Program, `image_example_test.c` *(4 of 4)*

```
    ts_destroy (src2);
    ts_destroy (dst1);

    /*
     * get some more test images
     */

    src1 = ts_get_src1_image (specfile);
    src2 = ts_get_src2_image (specfile);
    dst1 = ts_get_dst1_image (specfile);

 }

 /*
 * return the total number of errors (required) and close XIL
 */

 ts_end_test_case();

 xil_close (xil_State);

 total_errors = ts_end ();

 exit (total_errors);
}
```

Equivalence Testing Example C

This appendix explains how to write a test program that uses the equivalence testing functions and provides an example.

The equivalence testing functions test certain XIL features (such as ROIs and child images) without requiring any visual verification. These functions use the TS_FUNCFORMAT structure to receive descriptions of XIL functions. This structure is a union of structures that each correspond to an argument list for an XIL function. Take the following steps to write a test program that uses the equivalence testing functions:

1. **Declare the TS_FUNCFORMAT union variable as follows:**

```
TS_FUNCFORMAT      format;
```

2. **Declare the specific format pointer.**
   The format pointer is TS_*FN*_FORMAT, where *FN* is the name of the XIL function in all upper case. For example,

```
TS_XIL_CHOOSE_COLORMAP_FORMAT *choose_cmap_fmt;
```

3. **Cast the** TS_FUNCFORMAT **union pointer to a pointer for the structure needed for the XIL function.**
   For example,

```
choose_cmap_fmt = (TS_XIL_CHOOSE_COLORMAP_FORMAT*)&format;
```

4. **Initialize the code element of the format.**
   The code element must be set to `TS_`*FN*`_FORMAT_CODE`, where *FN* is the name of the XIL function in all upper case. For example,

```
choose_cmap_fmt->code = TS_XIL_CHOOSE_COLORMAP_FORMAT_CODE;
```

5. **Initialize the function element of the format, which must be set to the XIL function being tested.**
   For example,

```
choose_cmap_fmt->function = xil_choose_colormap;
```

6. **Specify values for the arguments of the XIL function.**
   You must specify a value for each of the XIL function's arguments. You do not identify the arguments by name; instead, you use `arg1`, `arg2`, and so on. For example,

```
choose_cmap_fmt->arg1 = src1;
choose_cmap_fmt->arg2 = 255;
```

`ts_automatic_tests` *Example*

Code Example C-1 is the code for testing the child image behavior of the XIL function, `xil_choose_colormap()`. The `ts_automatic_tests()` function is used.

Code Example C-1 `ts_automatic_tests()` Example

```
/*
* Declare TS_FUNCFORMAT union variable and specific format
    pointer.
*/
    TS_FUNCFORMAT  format;
    TS_XIL_CHOOSE_COLORMAP_FORMAT*choose_cmap_fmt;
/*
* Initialize variables.
*/
    choose_cmap_fmt = (TS_XIL_CHOOSE_COLORMAP_FORMAT*)&format;
    choose_cmap_fmt->code = TS_XIL_CHOOSE_COLORMAP_FORMAT_CODE;
    choose_cmap_fmt->function = xil_choose_colormap;
    choose_cmap_fmt->arg1 = src1;
    choose_cmap_fmt->arg2 = 255;
```

Code Example C-1 `ts_automatic_tests()` Example *(Continued)*

```
    errors = ts_automatic_tests (“choose_colormap”, &format, 1,
        0.0, NULL);
        .
        .
```

C

Index

V

X