Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) Developer's Guide

Order Number A00334-002

Revision History

Revision	Revision History	Date
0.1	Prototype ICTS Developer's Guide	08/1999
0.2	Prototype 2 ICTS Developer's Guide	12/1999
0.3	Incroporated IPMB, ICMB Information	04/2000
0.4	Update for IPMI 1.5	08/2001
0.5	Updated text, remove ICMB section, and added tlm_CMD, tlm_CMDex and Test_State.	01/2002
0.6	Updated for IPMI 2.0	08/2004

Contents

Figures

Tables

1 Overview

The Conformance Test Suite Framework provides a system for checking pass/fail conformance with the Intelligent Platform Management Interface specification. The Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) Developer's Guide provides a test development tutorial and the reference information to allow creation of custom tests, test suites, and libraries. This book also provides information about API support libraries provided by the base framework and for the support of transport modules.

The ICTS system includes a graphic user interface from which to load, run, and view test results. In addition, you can configure the host, target, and user-specific-characteristics of the interface to support the project or projects on which you are working.

The ICTS system does not support text editing, so you will use your favorite code or text editor to develop or customize test code. For information about how to install, configure, and run the test framework, refer to the Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User's Guide.

1.1 Purpose of the IPMI Conformance Test Suite

The IPMI 1.0/1.5/2.0 Conformance Test Suite (ICTS) consists of a software framework within which you may execute custom tests or a collection of predefined pass-fail tests for determining platform conformance with the IPMI 1.0/1.5/2.0 specification. The software framework includes a graphical user interface that provides configuration, loading, execution, and response access to the tests. The interface also allows recording of configuration file information for the host environment, the platform, the target, and user preferences. Using a text editor to customize configuration files allows creation of multiple cross-platform test sets. Saved configuration files and test sets, manipulated through the graphic interface, allow multiple board and/or project conformance testing from a single host platform.

NOTE

The test suite does not imply or enforce compliance requirements.

1.2 Audience

This document supports firmware development engineers desiring IPMI 1.0/1.5/2.0 conformance. The document assumes familiarity with the IPMI 1.0/1.5/2.0 specification, engineering practices related to cross-platform development, and general testing theory. For additional information concerning the IPMI specification and cross-platform issues, refer to the Reference Documents section of this chapter.

1.3 Overview of Test Process

Use of the IPMI 1.0/1.5/2.0 Conformance Test Suite assumes a host environment and a system to be tested. Before use, you must install and configure the IPMI 1.0/1.5/2.0 Conformance Test Suite (ICTS). Configurations can be saved, so configuration may not be required to run tests. Once configuration files for a host and target are complete, testing involves invoking the testing framework, loading configuration files, loading tests, running tests, and analyzing the results.

1.3.1 ICTS Architecture Overview

The ICTS framework resides on a host system and provides a graphic interface for loading, running, and analyzing IPMI conformance tests. Once installed and configured, the system provides a set of automatically loaded tests, access to a test library, and access to library support modules for communications with transport modules.

The test framework provides an interface for loading and running tests. Tests indicate their results in two different ways, as text messages in the framework message window for human consumption, and through internal status values which are passed up from child to parent tests and eventually to the framework itself. Parent tests utilize these status values to generate summary reports of child test results in the message window.

The FTF also loads and allows tests access to message procedures from the message library. The message library facilitates sending of messages to the firmware on the target system by providing the FTF and tests with a procedure interface to transport modules to allow accurate communication access to the target.

1.3.2 ICTS Architecture Layers

The FTF is a multi-layered tool primarily implemented in the Tcl scripting language. *Transport Modules* may be either C or Tcl. The following list contains the names of the major components and descriptions of their function.

Base Framework — An application program. In combination with libraries and loaded *Test Modules* it is a complete test application. The base framework provides the user interface and the environment in which tests run.

Test Module – Conducts tests and reports a pass/fail result by conducting the test itself or by executing other test modules as sub-components. The *Base Framework* loads *Test Modules* in order to create a complete test application program.

Tool Module – Conducts procedures that do not generate pass/fail results. The Tool Module has access to the same target and framework interface resources provided to a test module.

Message Library – A set of transport-independent message passing routines using services provided by loaded *Transport Modules.* The Message Library selects the default transport in the absence of an explicit requested by the caller.

Transport Modules – A set of loadable modules, one module per transport, that provides primitives for target interface communication. The module contains additional administrative primitives not used directly by the *Test Modules*, but which are used by the *Base Framework* on behalf of the *Test Modules*.

In addition to these components, the framework includes several *Companion Libraries* that provide various utility services to the *Test Modules*.

Figure 1-1 shows the relationship between the various components of the ICTS firmware test framework.

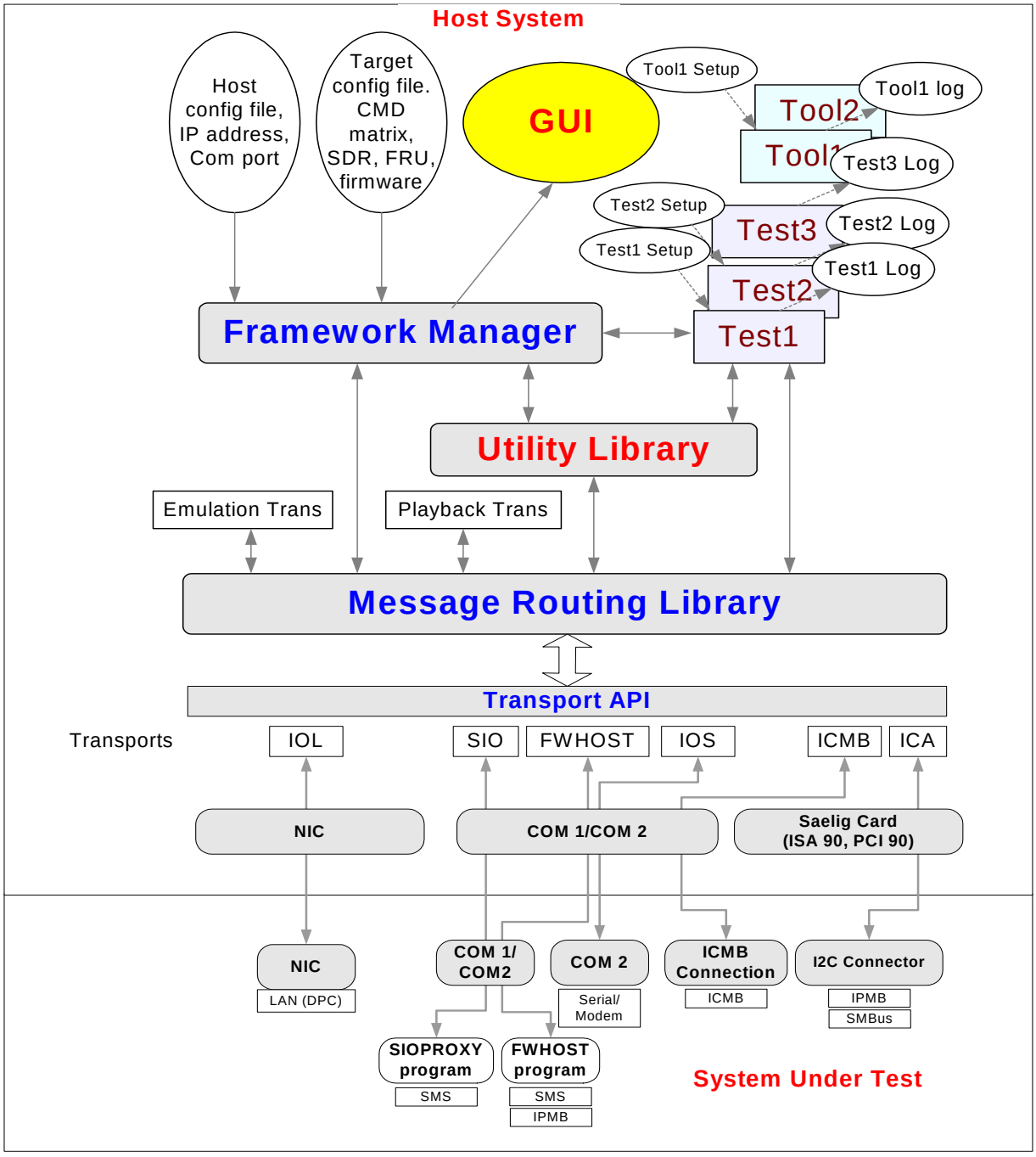

Figure 1-1. Firmware Test Framework Architecture

1.4 Customization Opportunities

The installed ICTS includes Tcl source code for many tests. The FTF provides four main areas of customization. The user interface allows customization of its appearance as well as customization for use in projects that require repeated testing or test modification. The framework allows inclusion of new test modules, new test libraries, and new transport modules.

1.4.1 Test Modules

Test modules consist of a test or group of tests written in Tcl. You can use any text editor to create new tests and test modules. Additionally, you can copy and modify existing tests and test modules from the source directories of the installed framework. For additional information on the tests included in the framework, refer to the *Intelligent Platform Interface (IPMI) Conformance Test Suite (ICTS) User's Guide*.

1.4.2 Libraries

Libraries are collections of Tcl procedures which may be shared by test modules. You can create new libraries or copy and modify the libraries included in the framework. For additional information on the libraries included in the framework, refer to the *Libraries* chapter of this manual.

1.4.3 Transport Modules

Transport modules are implemented in some combination of C and Tcl. Transport modules provide sets of Tcl language procedures that support communication across architecture layers. A given transport module supports a particular hardware/software architecture. Multiple transport modules may coexist within the framework. Their coexistence is managed by the message library. You can modify or create transport modules to utilized specialized hardware or device drivers.

1.5 ICTS Conformance Scope

The ICTS Framework automatically loads a test suite for pass-fail verification of IPMI 1.0/1.5 specifications for function module access commands and data storage requirements for SDR, SEL and FRU. The test suite does not test the internal workings of the target system. This section describes the testing scope of the ICTS Framework tests. For additional information about using the standard tests, refer to the *Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User's Guide*.

1.5.1 ICTS Supports

ICTS supports the following IPMI conformance requirements:

Management controllers, including BMC, should implement IPM functions.

- BMC should implement any mandatory functions such as IPM device, system interface, SDR repository Watchdog timer, Event Receiver, SEL Interface, Internal Event Generator.
- Each mandatory function should be present and implemented as defined by the IPMI specification.

- Any conditionally mandatory IPMI 1.0/1.5 command should be present and implemented as defined by the IPMI specification if the specified condition from IPMI 1.0/1.5 is met or present.
- Any optional IPMI 1.0/1.5 commands should be implemented as defined by the IPMI specification if the command is present.
- Any data stored in SEL or SDR should follow the format specified in the IPMI 1.0/1.5 specification.
- Mandatory logical devices should be implemented, such as BMC, SEL and SDR.
- If the IPMB is present, the mandatory IPMI commands should be transferred via it (unless a command is specified as mandatory for only the system interface).
- If the IPMB is present, the additional mandatory IPMB messaging support commands in the BMC must be implemented as defined by the IMPI specification.
- If the IPMB is present, it is highly recommended that the Initialization Agent should be implemented. If it is implemented then it must be done as defined by the IPMI specification.
- The platform must provide one of the system interfaces, either KCS, SMIC or BT.
- IPMI 2.0 specification

1.5.2 ICTS Does Not Support

ICTS does not support the following test classes:

- Target platform-specific functions are neither tested nor verified.
- The following list of functions are example of functions that do not fall under the ICTS scope:
  - OEM-specific functions. Network function code 30h-3Fh
  - Actual data field in OEM SEL Record Type C0h-DFh, Type E0h-FFh
  - Actual data field in OEM SDR, SDR Type 0C0h
  - Firmware commands. Network Function Code 08, 09
- ICTS does not measure target platform quality.
- ICTS does not do stealth tests and exception tests, unless explicitly mentioned in the IPMI 1.0/1.5 specifications.
- ICTS does not detect management controllers in the target system. It only takes target system information from the SDR.

For additional information on the IPMI 1.0/1.5 specification, refer to the specification listed in the *Reference Documents* section of this chapter.

1.6 Reference Documents

In addition to the information in this manual, the following documents may provide information useful to testing for IPMI 1.0/1.5 conformances:

- *Intelligent Platform Management Interface Specification* v2.0 Revision 1.0, © 2004 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation.
- *Intelligent Platform Management Interface Specification* v1.5 Revision 1.0, © 2001 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation.
- *Intelligent Platform Management Interface Specification* v1.0 Revision 1.1, © 1999 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation.

- *Intelligent Platform Management Bus Communications Protocol Specification* v1.0, rev. 1.2 © 2000 Intel Corporation.
- *Intelligent Chassis Management Bus Bridge Specification v1.0,* rev. 1.2, © 2000 Intel Corporation.
- *System Management Bus (SMBus) Specification, Version 2.0,* ©2000, Duracell Inc., Fujitsu Personal Systems Inc., Intel Corporation, Linear Technology Corporation, Maxim Integrated Products, Mitsubishi Electric Corporation, Moltech Power Systems, PowerSmart Inc., Toshiba Battery Co., Ltd., Unitrode Corporation, USAR Systems.

1.7 Glossary

This section contains a terms used throughout this document. For additional information regarding terms, refer to the documents listed in the Reference Documents section of this manual.

API

Application Program Interface.

EFI

Extensible Firmware Interface

FTF

Firmware Test Framework.

FRU

Field Replaceable Unit. A module or component that will typically be replaced in its entirety as part of a field service or repair operation.

Host

The machine executing the test, which may or may not be the same as the target.

ICTS

Intelligent Platform Management Interface (IPMI) Conformance Test Suite.

Interface

The local communication path on the target machine (I2C, SMS, etc.)

IPMB

Intelligent Platform Management Bus. Name for the architecture, protocol, and implementation of a special bus that interconnects the baseboard and chassis electronics and provides a communications medium for system platform management information. The bus is built on I2C and provides a communications path between management controllers such as the BMC, the ICMB bridge controller, and the chassis management controller.

IPMI

Intelligent Platform Management Interface.

SDR

Sensor Data Record. A data record that provides platform management sensor type, locations, event generation, and access information.

Target

The machine under test, which may or may not be the same as the host.

Transport Layer

The data communication path between the host and target machines (Local, RS-232, TCP/IP LAN, etc.).

UI

User Interface

IPM

Intelligent Platform Management

BMC

Baseboard Management Controller

SEL

System Event Log

Saelig card

A card providing the standard I2C interface to access the IPMB

2 Developing IPMI Conformance Test Modules

This chapter provides detailed information on developing individual tests, developing test libraries, and making your tests available in the framework. The chapter includes a tutorial that covers the basics of test creating using Tcl/Tk. Before engaging in the tutorial, be sure your system is installed and configured correctly. Installation and Configuration information appears in the Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User Guide.

2.1 Test Developer's Tutorial

This section is a tutorial demonstrating creation of simple tests. The material included here allows you to grasp the basic concepts of test creation and customizing tests already available in the ICTS system. The tutorial will walk you through creation of the Hello World program, the addition of diagnostics to help in development of the test, and the addition of analysis capture within the test.

For the purposes of this tutorial, a simple test has the following characteristics:

- It does not load or invoke any child tests.
- It does not invoke instances of itself.
- It does not send or receive any "raw" messages.
- It is transport-independent.
- It does not use Variable Description Tables (VDT).

2.2 The Hello World Test

The Hello World test in the following example is an arbitrarily chosen routine familiar to many programmers. For more realistic example tests, examine the test tcl files located at `C:\FTF\tests\`. The Hello World program discussed in the following pages is in the file `C:\FTF\templates\tests\hello1.tcl`.

The following example demonstrates use of comments, definition of a process, use of the `ftf_msg` statement to route information to the framework screen, and the use of the return statement to show success.

The simplest framework interpretable version of Hello World as an FTF-compliant test module appears here:

```
# Hello1.tcl

proc test_main { } {

    ftf_msg "Hello, World!"

    return 0

}
```

The "#" character marks a comment. The actual test contains one procedure:

```
proc test_main { } {
```

The `test_main` procedure conducts a test, in this case displaying a message on the framework screen:

```
ftf_msg "Hello, World!"
```

After displaying the message, the procedure reports the result by returning a zero to indicate that the test passed:

```
return 0
}
```

2.3 Debug and Verbose Levels

The above example provides the basic form for a test and could be implemented without consideration of debugging because of its simplicity. However, more complex tests will require some debugging.

Using the same basic test, the following example demonstrates use of `d` parameter to the `ftf_msg` statement to create debugging messages.

Below is the same `test_main` procedure with entry and exit debug messages added. Notice the `ftf_msg d` parameter that indicates the marked messages as debug messages. The Hello World message is not a part of the debug process, so it is not marked with the `d` parameter:

```
# Hello2.tcl
proc test_main { } {
    ftf_msg "Enter: test_main" d
    ftf_msg "Hello, World!"
    ftf_msg "Exit: test_main" d
    return 0
}
```

2.3.1 Setting the Level of Debug

The `d` parameter can take a numerical modifier that forces the message to appear only when the framework debug level is set to that number or higher. In the example above, the debug messages only display if the debug level is one or higher. Possible `d` modifier values are zero through three. In the following example message statement, the message executes only if the framework debug level is two or greater:

```
ftf_msg "Enter: test_main" d2
```

The `d` modifiers have the effects listed below:

- `d0` is equivalent to having no d parameter at all.
- `d1` is equivalent to "d". Messages appear for debug levels of one or greater.

- `d2` causes messages to appear for debug levels of two or greater.
- `d3` causes messages to appear only when the debug level is three.

Set the framework debug levels from the Options menu of the framework's user interface. The user configuration file also allows setting debug levels. For additional information on the user interface and configuration files, refer to the *Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User's Guide*.

On a per-test basis, debug levels can be set through the Options menu or by including the framework `ftf_setlevel` procedure. To determine the current debug setting within a test, use the query procedure `ftf_getlevel`.

2.3.2 Setting the Verbose Level

Verbose levels provide level-controlled display of detailed information regarding the results or progress of the test. Verbose levels are intended for use by the end-user rather than the developer. However, verbose levels may be useful during development.

On a per-test basis, debug levels can be set through the Options menu or by including the framework `ftf_setlevel` procedure. To determine the current verbose setting within a test, use the query procedure `ftf_getlevel`. The verbose message parameter is `v`. As with the debug level, the parameter takes a numerical modifier, zero through three that determines the verbose level setting at which the message will appear.

2.3.3 Verbose or Debug from the Framework

To change the verbose or debug output level, you perform essentially the same procedure. You can change the global output level by beginning the procedure from the Levels item on the Options menu. To change the output level or a particular test and all of its children, begin the procedure from the Levels item on the menu for a selected test.

Figure 2-1 shows the menu for the test, "My Custom Test." Figure 2-2 shows the Options menu:

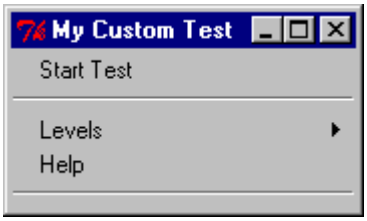

Figure 2-1. Test Menu Showing Levels Item

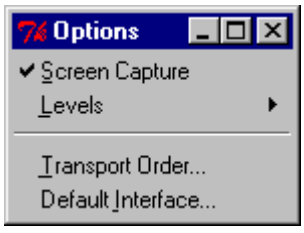

Figure 2-2. Options Menu Showing Levels Item

To change the output level, complete the following procedure:

1. *Click* on the Levels option from the appropriate menu. The Levels drop-down appears. Figure 2-3 shows the Levels drop-down menu.

Figure 2-3. The Levels Drop-Down

2. *Click* Verbose… to change the verbose output level. *Click* Debug… to change the debug output level. The Verbose Setting dialog box and the Debug setting dialog box are identical except for their titles and effects. The dialog appropriate to your selection appears. Figure 2-4 shows the Logging dialog box used for setting the verbose and debug levels:

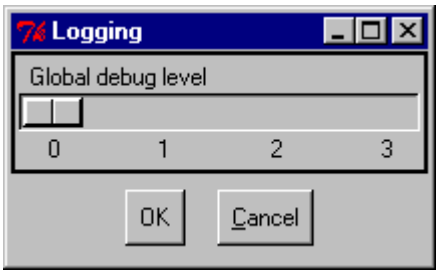

Figure 2-4. Global Debug Dialog Box

3. Select the level of output you desire.
4. *Click* OK to enable the global verbose setting you have selected. *Click* on Cancel to return to the interface without changing the level.

2.4 Reporting Test Results

This section describes two methods by which tests report results. Tests report either through the return value of the procedures `test_main`, or through API calls to the firmware test library.

2.4.1 test_main Return Values

All valid tests report results through the `return` statement. The `test_main` procedure must return a numeric value. The values for returns from `test_main` fit a defined paradigm. The possible `test_main` return values and their meanings appear in the following list:

- Zero—the test completed and passed.
- A positive value—the test completed but did not pass. Define the meaning of the positive values in each test. If a test will be used as a child test, the return can signal the nature of the failure to the parent.
- A negative value—the test could not be conducted or is not applicable. As with positive return values, the meaning of specific negative values is test-dependent.

In the Hello World example, the test completes successfully if each line executes and the messages appear on the screen. If control falls to the return statement, a zero is returned. Any other result is failure, so no other return statements are needed.

```
return 0
```

2.4.2 Reporting Test Results Through the Firmware Library

The Firmware Test Library provides standardized test result procedures to allow ease and consistency of test reporting. Test modules in a formal test suite must report through Firmware Test Library calls. All other tests may report, but it is not required. The pass and fail procedures are in the file `C:\FTF\libs\lib_gen.tcl`.

The following table lists the test procedures and their purposes:

Table 2-1. Firmware Library Test Result Procedures

Procedure Name	Function
`print_pass "Message"`	Reports a pass and prints `Message` to the screen.
`print_na "Message"`	Reports N/A and prints `Message` to the screen.
`print_warn "Message"`	Reports a warning and prints `Message` to the screen.
`print_fail "Message"`	Reports a fail and prints `Message` to the screen.

Report a test-pass event with *one* call to the `print_pass` procedure. Report a test-failure event with *one* call to the `print_fail` procedure. The argument to all of these procedures is a text string for display on the screen.

WARNING

It is important that each pass/fail/NA event make only one call because `print_pass`, `print_fail`, and `print_na` increment global counter variables maintained by the library. If you want to display additional information without incrementing the counters, use the `ftf_msg` procedure provided by the framework. There are no counters associated with the `print_warn` procedure.

The following example shows the Hello World test with a `print_pass` statement to report the pass result:

```
# Hello4.tcl
proc test_main { } {
ftf_msg "Hello, World!"
print_pass "Said hello to world."
return 0
}
```

2.4.2.1 Pass/Fail Example: even1.tcl

The following example test, `Even1.tcl`, is more complex than the Hello World example and represents a more accurate picture of test creation. `Even1` consists of three procedures. The `test_main` procedure calls the `set_seed` procedure to get a seed value that is passed to the random number generator, `rand`. If `rand` provides an even value, the test passes; otherwise, it fails. In the event of a pass, the result number for return is set to 0, `print_pass` prints a message to the screen, and the pass counter is incremented automatically. In the event of failure, the result number for return is set to 1, `print_fail` prints a message to the screen, and the fail counter is incremented automatically. In both events, `ftf_msg` prints the value of the number being tested.

```
# Even1.tcl
# Create a variable for the seed value and set it to 0
variable seed 0

# Create a non-zero seed value for random number generation
proc set_seed { newseed } {
variable seed
set seed $newseed
}

# Generate a random number based on a seed.
proc rand { } {
variable seed
set seed [expr ($seed*9301+49297)%233280]
return [expr int(0xFFFF*($seed/double(233280)))]
}

# Create a seed value. Generate a random number based on the seed.
# Test the generated number for odd or even status.
# Fail on odd. Pass on even. Report results to the screen.
proc test_main { } {
set_seed [clock clicks]
set randval [rand]

if { [expr $randval % 2] == 0 } {
set result 0
print_pass "Even number generated"
} else {
set result 1
print_fail "Odd number generated"
}
ftf_msg "Random value: $randval" v

return $result
}
```

2.5 Sending/Getting IPMI Messages

This section describes the use of the message library and the Firmware Test Library to send IMPI messages to the target machine and to receive replies from the target.

2.5.1 The Message Library

You can find the source for the message library at `C:\FTF\packages\msglib.tcl`. Message library IPMI messages are either "cooked" or "raw." This section deals with simple tests, as defined in the Test Developer's Tutorial section of this chapter. Simple tests do not send raw messages. The following discussion assumes all messages will be cooked.

2.5.2 A Note on Cooked Commands

Tests not intended to survive from one version of the IPMI specification to the next can use message library procedures. When using the message library directly, it is up to the test to understand the request and response data associated with the messages. Tests that are intended to survive changes to the IPMI specification should use the Firmware Test Library procedures instead. This higher-level library contains knowledge of the IPMI request and response messages, and packages them correctly for the current version of IPMI.

2.5.2.1 Message Library Example: gdidsms1.tcl

The following example sends a `GetDeviceId` command for IPMI 1.0 to the SMS interface using the message library:

```
1               # gdidsms1.tcl
2
3               # This test requires the SMS interface.
4               variable Test_Interfaces [list SMS]
5
6               proc test_main { } {
7                   # Cooked IPMI command to get the device ID.
8                   set gdidcmd [list 0x20 6 0 1]
9
10                  # Variable for reporting pass/fail. Assume pass to start.
11                  set result 0
12
13                  # Send command using "msend" proc of the message library.
14                  set sendList [msend SMS $gdidcmd]
15
16                  # Parse result: An error code and a request descriptor.
17                  set ecode [lindex $sendList 0]
18                  set rd [lindex $sendList 1]
19
20                  if { $ecode != 0 } {
21                      # Send failed. Can't finish the test.
```

```
22                  print_fail "msend: [ftf_error $ecode SMS]"
23                  set result -1
24               } else {
25                  # Use request descriptor to get the reply.
26                  set getList [mget $rd]
27
28                  # Parse result: An error code and an IPMI reply.
29                  set ecode [lindex $getList 0]
30                  set reply [lindex $getList 1]
31
32                  if { $ecode != 0 } {
33                     # Get failed. Can't finish the test.
34                  print_fail "mget: [ftf_error $ecode SMS]"
35                  set result -2
36                  } else {
37                     # Test completed. Now determine pass/fail.
38                     # Extract completion code from the reply.
39                     set ccode [lindex $reply 0]
40
41                  if { $ccode != 0 } {
42                     # Test did not pass.
43                     print_fail "completion code = $ccode"
44                     set result $ccode
45                  } else {
46                     # Test passed.
47                     print_pass "GetDeviceId reply received."
48                     set did [lindex $reply 1]
49                     set drev [lindex $reply 2]
50                     set fwid [lindex $reply 3]
51                     set fwrev [lindex $reply 4]
52                     ftf_msg "Device ID: $did" v
53                     ftf_msg "Device Rev: $drev" v
54                     ftf_msg "FW ID: $fwid" v
55                     ftf_msg "FW Rev: $fwrev" v
56                  }
57               }
58            }
59
60            return $result
61            }
```

2.5.2.2 Highlights of gdidsms1.tcl:

Line 4: Target interface.

If a test module requires a specific target interface, it must define `Test_Interfaces` as a list of the required interfaces. This test requires the SMS interface.

Line 7: Syntax of a "cooked" command.

The general form is a list of numbers as follows:

```
<dest> <netfn> <lun> <cmd> <data1> <data2> <data3> ... <dataN>
```

The `GetDeviceId` command has no data, so in the example the last byte is the command number (1).

Line 14: Sending a message with `msend`.

The first parameter to `msend` is the interface. Optionally, you can also specify the transport by using "SMS/LOCTRANS" to send the message through the local transport. In this case, you would use an `mroute` procedure to determine if the desired route is valid. The second parameter to `msend` is the command constructed in line 7.

Lines 17 and 18: Break apart the return value of `msend`.

A two-item list is returned. The first item is an error code. A non-zero value indicates an error in the message library or the transport module. The `ftf_error` statement at line 22 converts the error code to a string for display. The second item is a request descriptor that is needed in order to read the reply.

Line 26: Get the reply with `mget`.

The parameter is the request descriptor we extracted from the `msend` result in line 18.

Lines 29 and 30: Break apart the return value of `mget`.

A two-item list is returned. The first item is an error code. The second item is another list containing a "cooked" IPMI of the form:

```
<completion code> <data1> <data2> <data3> ... <dataN>
```

Line 39: Extract the completion code.

Lines 41 – 44: Unable to get device ID. Test did not pass.

Uses the completion code as the test failure code.

Lines 46 – 55: Got device ID. Test passed.

Displays details only if verbose level is one or higher.

2.5.2.3 Message Library Example: gdidsms2.tcl

The following example uses the `msendget` procedure to simplify `gdidsms1.` While simpler, this example has less visibility into the send/get sequence in the event that something goes wrong. The difference appears in line 14, and the subsequent parsing of the results. In `gdidsm2`, the returned value of `reply` is not a list, but a single value.

```
1     # gdidsms2.tcl
2
3     # This test requires the SMS interface.
4     variable Test_Interfaces [list SMS]
```

```
5
6     proc test_main { } {
7          # Cooked IPMI command to get the device ID.
8          set gdidcmd [list 0x20 6 0 1]
9
10         # Variable for reporting pass/fail. Assume pass to start.
11         set result 0
12
13         # Send command and get reply using "msendget" proc.
14         set sendgetList [msendget SMS $gdidcmd]
15
16         # Parse result: An error code and an IPMI reply.
17         set ecode [lindex $sendgetList 0]
18         set reply [lindex $sendgetList 1]
19
20         if { $ecode != 0 } {
21              # Send/Get failed. Can't finish the test.
22              print_fail "msendget: [ftf_error $ecode SMS]"
23              set result -2
24         } else {
25              # Test completed. Now determine pass/fail.
26              # Extract completion code from the reply.
27              set ccode [lindex $reply 0]
28
29              if { $ccode != 0 } {
30                   # Test did not pass.
31                   print_fail "completion code = $ccode"
32                   set result $ccode
33              } else {
34                   # Test passed.
35                   print_pass "GetDeviceId reply received."
36                   set did [lindex $reply 1]
37                   set drev [lindex $reply 2]
38                   set fwid [lindex $reply 3]
39                   set fwrev [lindex $reply 4]
40                   ftf_msg "Device ID: $did" v
41                   ftf_msg "Device Rev: $drev" v
```

```
42                  ftf_msg "FW ID: $fwid" v
43                  ftf_msg "FW Rev: $fwrev" v
44                }
45        }
46
47        return $result
48    }
```

2.5.3 Messages with the Firmware Test Library

The Firmware Test Library allows messages and replies without requiring packaging and parsing. It also protects test modules from IPMI specification revisions that might invalidate tests using the message library.

2.5.3.1 Message Library Example: gdidsms3.tcl

The following example uses the Firmware Test Library for messaging and replying in the `SMS/GetDeviceId` test:

```
1     # gdidsms3.tcl
2
3     # This test requires the SMS interface.
4     variable Test_Interfaces [list SMS]
5
6     proc test_main { } {
7         # Create control info for sending messages.
8         set pctl [set_packet_controls 0x20 "" SMS]
9
10        # Variable for reporting pass/fail. Assume pass to start.
11        set result 0
12
13        # Send command and get reply.
14        set rspData [req_rsp GetDeviceId "" $pctl]
15
16        # Convert reply to an array.
17        array_set rspArray $rspData
18
19        if { $rspArray(merr) != 0 } {
20             # Send/Get failed. Can't finish the test.
21             print_fail "req_rsp: [ftf_error $rspArray(merr) SMS]"
22             set result -2
```

```
23        } else {
24             # Test completed. Now determine pass/fail.
25             # Extract completion code from the reply.
26             if { $rspArray(CompCode) != 0 } {
27                  # Test did not pass.
28                  print_fail "completion code = $rspArray(ccode)"
29                  set result $rspArray(CompCode)
30             } else {
31                  # Test passed.
32                  print_pass "GetDeviceId reply received."
33                  if { [ftf_getlevel v] } {
34                       print_rsp rspArray
35                  }
36             }
37        }
38
39        return $result
40    }
```

2.5.3.2 Highlights of gdidsms3.tcl:

Line 8: Construct a control variable for sending messages.

The `set_packet_controls` procedure takes a destination, `LUN`, interface (or route) and, optionally, a timeout value and combines them into a single variable that can be reused for sending a series of messages to the same destination. Because only one message is sent, the benefit is not obvious.

Line 14: Send message and get the reply with `req_rsp`.

The second parameter is an empty string that supplies the data bytes following the command number. The `fmt_req` procedure of the Firmware Test Library constructs this parameter for commands that need it. `req_rsp` returns a list that's not very useful without additional processing.

Line 17: Convert awkward list to a less-awkward array.

Line 19: Check result of send/get.

This is the same error code returned by `msendget` of the message library.

Lines 26-29: Unable to get device ID. Test did not pass.

Uses the completion code as the test failure code.

Lines 31-35: Got device ID. Test passed.

Displays details if the verbose level is one or higher. Uses `print_rsp` to display the contents of the array. The array contains the response data and the command number used to get the data. Therefore `print_rsp` knows the meaning of the data and displays it accordingly. However, `print_rsp` has no verbose option, so `ftf_getlevel` queries the effective level and the call to `print_rsp` is conditional.

2.6 Other Test Procedures

Aside from `test_main` and procedures required directly by `test_main`, the framework recognizes additional `test_` procedures. This section describes recognized procedures that are optional in simple tests. The procedures discussion includes: `test_setup`, `test_init`, and `test_help`. The section also contains examples of their use in the context of the `even2.tcl` program, for testing whether a randomly generated number is even or odd.

2.6.1 Initialization and Cleanup Procedures

The framework supports two test initialization procedures: `test_setup` and `test_init`. In addition, the framework supports one clean up procedure and one help procedure. They are `test_cleanup` and `test_help`, respectively.

2.6.1.1 test_setup

Normally, the `test_setup` procedure is called before `test_main` and just after a test module or a setup file for the test is sourced. It may be called again under the direction of a parent test, or if the user selects a new setup file for the test.

Some `test_setup` procedure responsibilities can be handled when sourcing the test. However, initialization at source time means the test's setup file (if any) has not been sourced, and the entire API of the framework and its libraries are not yet available to the test.

The `test_setup` return value must be zero if initialization is successful or a non-zero number if initialization fails.

2.6.1.2 test_init

The `test_init` procedure is also called before `test_main` each time the test is run. The `test_init` procedure causes the test to abort if it returns a non-zero number indicating failure.

2.6.1.3 test_cleanup

The `test_cleanup` procedure is called after `test_main`. A zero return value indicates success. A non-zero return value indicates failure.

2.6.1.4 test_help

The `test_help` procedure uses the `ftf_msg` procedure to display a help message, if requested through the Help menu of the framework user interface. This procedure takes no arguments and has no return value.

2.6.1.5 Example: even2.tcl

The following example shows the even-number test modified so the random number seed is set once instead of every time `test_main` is called. The assignment is inside the `test_setup` procedure. The example also shows the use of `help_test` and the sequence of operations for `test_init` and `test_cleanup` procedures. However, calls to `test_help`, `test_init` and `test_cleanup` print messages without performing other significant action.

```
# Even2.tcl
variable seed 0

proc set_seed { newseed } {
    variable seed
    set seed $newseed
}

proc rand { } {
    variable seed
    set seed [expr ($seed*9301+49297)%233280]
    return [expr int(0xFFFF*($seed/double(233280)))]
}

proc test_setup { } {
    ftf_msg "Proc: test_setup" d
    set_seed [clock clicks]
    return 0
}

proc test_init { } {
    ftf_msg "Proc: test_init" d
    return 0
}

proc test_main { } {
    ftf_msg "Proc: test_main" d

    set randval [rand]

    if { [expr $randval % 2] == 0 } {
        set result 0
        print_pass "Even number generated"
    } else {
        set result 1
        print_fail "Odd number generated"
    }
```

```
    ftf_msg "Random value: $randval" v

    return $result
}

proc test_cleanup { } {
    ftf_msg "Proc: test_cleanup" d
    return 0
}

proc test_help { } {
    ftf_msg "Even2 - Demonstrates the test_setup procedure."
}
```

2.6.2 Saving and Restoring States

This section contains information for saving the current state of tests and later restoring that state to allow further testing based on the stored state. The procedure used to save the current state is `test_state`. The procedure used for restoring a state is `test_setup`. If a test contains the `test_state` procedure, it must also contain the `test_setup` procedure.

2.6.2.1 test_state

The `test_state` procedure queries the framework for the current state of the test module from which it is called. The returned list contains the current test state. The framework can save the current state to a file. After starting the framework, the same tests can be loaded and restored to the saved state.

2.6.2.2 test_setup

The `test_setup` procedure restores a test's current state by taking the list returned by `test_state` as a parameter. The `test_setup` procedure often appears without parameters, but if a test implements `test_state`, then the test must also implement `test_setup` and pass it an optional parameter.

2.6.2.3 Example: even3.tcl

The following version of the `even*.tcl` test example uses `test_state` to record the current state and the random number seed. The `test_setup` procedure appears in the example with the optional parameter for restoring the recorded random number seed.

```
# Even3.tcl
variable seed 0

proc set_seed { newseed } {
```

```
    variable seed
    set seed $newseed
}

proc rand { } {
    variable seed
    set seed [expr ($seed*9301+49297)%233280]
    return [expr int(0xFFFF*($seed/double(233280)))]
}

proc test_state { } {
    variable seed
    return [list seed $seed]
}

proc test_setup { {stateList ""} } {
    set err 0
    if { $stateList == "" } {
        set_seed [clock clicks]
    } else {
        array set stateArray $stateList
        if { [info exists stateArray(seed)] } {
            set_seed $stateArray(seed)
        } else {
            ftf_msg "Bad state list"
            set err 1
        }
    }
    return $err
}

proc test_main { } {
    set randval [rand]

    if { [expr $randval % 2] == 0 } {
        set result 0
        print_pass "Even number generated"
```

```
    } else {
        set result 1
        print_fail "Odd number generated"
    }
    ftf_msg "Random value: $randval" v

    return $result
}
```

2.7 Parent and Child Tests

This section contains a description of the relationships between parent and child tests. It also contains examples of their use.

The term "parent test" describes the relationship between a test and tests it is responsible for loading. A parent test is a Firmware Test Framework-compliant test module that loads and executes other FTF-compliant test modules. Tests loaded and executed by a parent are called child tests. The child of one parent test may be the parent of its own child tests. A parent may have multiple children. A child may have at most one parent.

The framework's cascading test menu provides an exact map of the parent-child hierarchy. However, a parent can load a child and request that the child be a "hidden child" and not appear in the menu hierarchy. Tests loaded by a hidden child are also hidden.

Figure 2-5 shows the test menu containing two tests. The automatically loaded tests suite and a custom test:

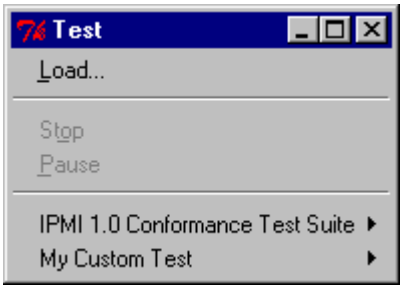

Figure 2-5. Test Menu Showing Custom Test

Figure 2-6 shows the children of the automatically loaded test suite. Similarly, selecting the custom test would reveal any children invoked by the custom test.

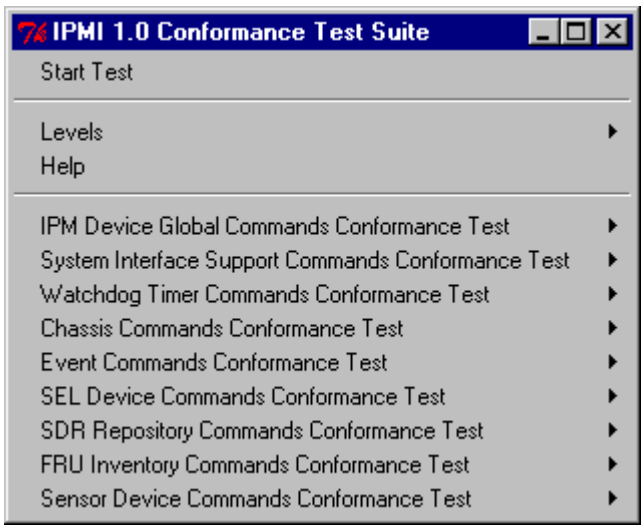

Figure 2-6. Test Menu Showing Children of IPMI 1.0 Conformance Test Suite

NOTE

Only resource constraints imposed by Tcl, the host operating system, and the host hardware limit the depth of the parent-child stack.

2.7.1 Loading Children

Parent tests load children through either static loading or dynamic loading. A single test may use both methods.

2.7.1.1 Static Loading

Use static loading if the name of a child test is fixed and known during implementation of the parent test. Static loading does not allow hidden children. Loading a hidden child requires dynamic loading.

When implementing a static load, use the `Test_Children` variable containing a list of children identified by file name, with or without their `.tcl` extension, which is assumed. The file names may include their directory paths. To resolve relative paths, the framework uses the `Host_TestDirs` variable from the host configuration file. For additional information about host configuration files, refer to the Installation and Configuration chapter of the *Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User Guide.*

NOTE

Parents cannot use their own `test_setup` procedure to set up a child test because the framework cannot load a child prior to calling the parent's `test_setup` procedure.

WARNING

Do not use static loading for recursive tests. The framework will attempt to load an infinite stack of tests. All system resources will be consumed, and the operating system will crash. Use dynamic loading for recursive tests. Even then, take great care to avoid an infinite-stack scenario.

2.7.1.2 Example: Static Load

The following example shows the simplest form of a static loading parent. This parent test, `HelloP1.tcl` contains no procedures. It does not contain test code or code to invoke its child tests. The example code loads only a fixed-name group of children.

```
# HelloP1.tcl
# Parent test to statically load all the Hello examples as children.


variable Test_Children [list Hello1 Hello2 Hello3 Hello4]
```

2.7.1.3 Dynamic Loading

Use dynamic loading of child tests for recursive tests, when the names of the children are not known, and to prevent the child test from appearing in the framework's menu hierarchy. Dynamic child test loading requires the `ftf_loadtest` procedure.

2.7.1.4 Example: Dynamic Load

The following example shows dynamic loading using the `ftf_loadtest` within a `foreach` control structure. The control structure allows iteration through a list of test names created or gathered at run time.

```
# HelloP2.tcl
# Parent test to dynamically load all the Hello examples as children.

proc test_setup { } {
    set err 0
    foreach child [list Hello1 Hello2 Hello3 Hello4] {
        set err [ftf_loadtest $child]
        if { $err } {
            break
        }
    }
    return $err
}
```

Adding an "h" (hidden) as the second parameter to `ftf_loadtest` keeps the children from appearing in the user interface menu hierarchy. In the example, the parent provides no means to invoke the children, so they can not appear in the hierarchy and the hidden parameter is not needed.

2.8 Child Initialization Invocation and Cleanup

This section describes initialization, invocation, and clean up for child tests.

Before invoking a child test, the parent must initialize the child. After a child test invocation, the parent must clean up. Each child initialization requires a separate clean up procedure. The following three procedures initialize and clean up after a child test:

- `ftf_setuptest` — assigns a test setup file to a child, then invokes the child's `test_setup` procedure, if one exists.
- `ftf_inittest` — invokes the child's `test_init` procedure, if one exists.
- `ftf_cleanuptest` — invokes the child's `test_cleanup` procedure, if one exists.

For additional information on `test_setup` and `test_cleanup`, refer to the Initialization and Cleanup Procedures section of this chapter.

Both `ftf_setuptest` and `ftf_inittest` may be used to define Tcl variables in the child's namespace. For additional information on API routines, refer to the Libraries chapter of this manual.

2.8.1 Invoking Children

The API routine for invoking children maps to the `test_main` procedure of the child. The `ftf_runtest` procedure invokes `test_main`. For additional information on this routine, refer to the Libraries chapter of this manual.

Several variations on child initialization, invocation, and clean up are valid. Several variations on parent-child interaction are valid. Within the limits of the API, the parent has few restrictions on how it interacts with its children. The examples in the following subsections show two methods of interaction.

2.8.2 Example: HelloP3 Parent-Child

The following parent-child interaction example, `HelloP3.tcl`, initializes all its children at once, invokes them all in succession, and then cleans up afterwards:

```
# HelloP3.tcl

# Parent test to process children in bulk.

variable My_Children [list Hello1 Hello2 Hello3 Hello4]


proc test_setup { } {
    set err 0
    variable My_Children


    foreach child $My_Children {
    set err [ftf_loadtest $child]
    if { $err } {
        ftf_msg "$child load error: err=$err"
```

```
            break
        }

        set eList [ftf_setuptest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
            if { $lerr || $cerr } {
                ftf_msg "$child setup failed: lerr=$lerr, cerr=$cerr"
                set err 1
                break
            }
        }
        return $err
    }

    proc test_init { } {
        set err 0
        variable My_Children

        foreach child $My_Children {
            set eList [ftf_inittest $child]
            set lerr [lindex $eList 0]
            set cerr [lindex $eList 1]
            if { $lerr || $cerr } {
                ftf_msg "$child init failed: lerr=$lerr, cerr=$cerr"
                set err 1
                break
            }
        }
        return $err
    }
    proc test_main { } {
        set err 0
        variable My_Children

        foreach child $My_Children {
            set eList [ftf_runtest $child]
```

```
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            print_fail "$child failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }
    }
    if { !$lerr && !$cerr } {
        print_pass "All children passed."
    }
    return $err
}
proc test_main { } {
    set err 0
    variable My_Children

    foreach child $My_Children {
        set eList [ftf_runtest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            print_fail "$child failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }
    }
    if { !$lerr && !$cerr } {
        print_pass "All children passed."
    }
    return $err
}

proc test_cleanup { } {
    set err 0
    variable My_Children
```

```
    foreach child $My_Children {
        set eList [ftf_cleanuptest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            ftf_msg "$child cleanup failed: lerr=$lerr, cerr=$cerr"
            set err 1
        }
    }
    return $err
}
```

The following parent-child interaction example, HelloP4.tcl, operates on one child at a time, first initializing it, then invoking it, then cleaning up before initializing the next child:

```
# HelloP4.tcl
# Parent test to process children one at a time.
variable My_Children [list Hello1 Hello2 Hello3 Hello4]

proc test_main { } {
    set err 0
    variable My_Children

    foreach child $My_Children {
        set err [ftf_loadtest $child]
        if { $err } {
            print_fail "$child load error: err=$err"
            break
        }

        set eList [ftf_setuptest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            print_fail "$child setup failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }
```

```
        set eList [ftf_inittest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            print_fail "$child init failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }

        set eList [ftf_runtest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            print_fail "$child failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }

        set eList [ftf_cleanuptest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            ftf_msg "$child cleanup failed: lerr=$lerr, cerr=$cerr"
            break
        }
    }
    if { !$lerr && !$cerr } {
        print_pass "All children passed."
    }
    return $err
}
```

2.9 Variable Description Tables

This section contains a brief description of Variable Description Tables (VDT) and an example of their use.

The VDT feature allows definition of a test module that specifies the characteristics for a set of variables. At runtime, the framework generates values for those variables on behalf of the test. The test itself is isolated from generating VDT values. The test only defines the variable's properties.

The framework generates the VDT variable values in a number of ways. When the framework is run in interactive mode, the framework may choose to prompt the user for the value. In batch mode, it may accept the default value specified by the test module in its VDT, acquire the value from some other source, or generate an error and abort the test.

NOTE

At present the batch mode feature of the framework is not implemented. This is a concept reserved for future use.

2.9.1 Setting Up a VDT Table

A VDT is an array indexed by the names of other variables. The index variable names are framework-generated on behalf of the test. The default name for the array is `Test_VDT`. The test may use another name or define multiple arrays of this type.

Each element of a VDT array is a list containing from one to five items. Each position in the list has a specific meaning. This section describes only the first three of the five items.

The following is an example definition of a VDT:

```
variable Test_VDT

set Test_VDT(theAnswer) [list 42 "The answer to everything." d]
```

In the above example, the following components appear:

`theAnswer` — the name of a variable

`42` — the default value for `theAnswer`.

`"The answer to everything."` — a description of `theAnswer`.

`d` — a type specifier for signed integer. The type specifier may be either a single letter flag or a list of two or more items.

The following list provides the valid VDT variable types and their meaning:

`d`, `u`, `x`, `X`, `s`, `f`, `e`, `E`, `g`, and `G`

The framework supports the following additional types:

`b` Binary (a series of ones and zeros)

`y` Boolean (yes or no)

`F` File Name

`T` Test module file name (for selecting child tests)

`S` Setup file name (for selecting setup files for child tests)

If the type specifier is not a letter but instead a list of two or more items, it indicates that the framework should choose a value from the list of two or more items in order to generate the variable.

The following example shows a VDT entry using a list in place of the type specifier:

```
variable Test_VDT

set Test_VDT(order) [list “forward” “Order” [list “forward” “reverse”]]
```

2.9.2 Generating Variables

To generate a VDT variable, a test uses the `ftf_getvar` procedure. The first, and only required parameter is the name of the variable to be generated. The variable name is usually an index of the `Test_VDT` array. The variable name may also refer to a variable not listed in `Test_VDT`, in which case the framework assumes that it is a string variable as though it were defined with the following VDT entry:

```
variable Test_VDT

set Test_VDT(varname) [list “” “varname” s]
```

To generate our “`theAnswer`” variable from the example in the previous section, call `ftf_getvar` as follows:

```
ftf_getvar theAnswer
```

If the variable already exists, `ftf_getvar` takes no action; otherwise, `ftf_getvar` creates the variable in the test’s namespace. The test can then use the `variable` command to access the variable from inside its procedures.

To force regeneration, even if the variable already existed, call `ftf_getvar` with an extra, non-zero parameter as shown in the following line:

```
ftf_getvar theAnswer 1
```

An optional third parameter allows the test to specify an alternate variable description table. For a more detailed description of `ftf_getvar`, refer to the Libraries chapter of his manual.

2.9.3 Example: VDT

The following example, `HelloP5.tcl`, shows a parent invoking only one child each time it is run. The selection of the child is made using a VDT entry:

```
# HelloP5.tcl

# Parent test to invoke one of several children using a VDT entry.

variable My_Children [list Hello1 Hello2 Hello3 Hello4]

variable Test_VDT

set Test_VDT(child2run) [list "Hello1" "Child to run." $My_Children]


proc test_setup { } {
```

```
    set err 0
    variable My_Children

    foreach child $My_Children {
        set err [ftf_loadtest $child]
        if { $err } {
            ftf_msg "$child load error: err=$err"
            break
        }

        set eList [ftf_setuptest $child]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
            ftf_msg "$child setup failed: lerr=$lerr, cerr=$cerr"
            set err 1
            break
        }
    }
    return $err
}

proc test_init { } {
    set err 0
    variable child2run

    ftf_getvar child2run 1

    if { $child2run == "" } {
        ftf_msg "Test cancelled"
        set err 1
    } else {
        set eList [ftf_inittest $child2run]
        set lerr [lindex $eList 0]
        set cerr [lindex $eList 1]
        if { $lerr || $cerr } {
```

```
            ftf_msg "$child init failed: lerr=$lerr, cerr=$cerr"
            set err 1
        }
    }
    return $err
}
proc test_main { } {
    set err 0
    variable child2run

    set eList [ftf_runtest $child2run]
    set lerr [lindex $eList 0]
    set cerr [lindex $eList 1]
    if { $lerr || $cerr } {
        print_fail "$child2run failed: lerr=$lerr, cerr=$cerr"
        set err 1
    } else {
        print_pass "$child2run passed."
    }
    return $err
}
proc test_cleanup { } {
    set err 0
    variable child2run

    set eList [ftf_cleanuptest $child2run]
    set lerr [lindex $eList 0]
    set cerr [lindex $eList 1]
    if { $lerr || $cerr } {
        ftf_msg "$child2run cleanup failed: lerr=$lerr, cerr=$cerr"
        set err 1
    }
    return $err
}
```

3 Developing Transport Modules

This chapter contains information to support development of and customization of transport modules. Transport modules provide cross-level architecture messaging between the host or target Messaging Library and the communications ports. The examples provided in this chapter rely on a local transport module that implements the SMS interface. Salieg implements the I2C interface via the FWH-I2C transport.

All transport modules share the same API; however, some modules may not implement every standard routine. Some modules may also contain non-standard routines.

NOTE

Use the Message Library Test module to call transport modules. Avoid calling transport module routines directly.

3.1 Cooked and Raw Messages

Test modules support two types of messages: cooked and raw. The normal form of transport messages is cooked. Cooked messages allow maximum coverage of possible interfaces. Raw messages are interface-specific. This section contains the specifications for cooked and raw messages.

3.1.1 Cooked Message Specification

The cooked message support feature of transport modules allows standardization of interface formats. Cooked messages are interface-independent and transport-dependent. The cooked message formats appear in this section.

3.1.1.1 Sending Cooked Messages

Specifications:

```
proc tsend { pd iface[:node] data {options ""} }
```

Parameters:

`pd`	Port descriptor, as returned by `topen`.
`iface`	The target interface.
`node`	The target bus node.
`data`	Command data in the form of a Tcl list of numeric values.
`options`	An empty string selects the default options. The letter "`n`" is the no-reply option indicating that no reply is to be expected.

Return Value:

A Tcl list of the form {ecode md}:

ecode An error code: Zero on success; non-zero on an error.

md A message descriptor that can be used to get the message reply (meaningless for a non-zero ecode and the no-reply option).

Usage Notes:

The message descriptor (md) is not necessarily numeric.

Implementation Notes:

The message descriptor (md) doesn't have to be numeric, although numeric is the most convenient. Any combination of letters, digits, and underscore characters is allowed. Message descriptors are case-sensitive.

The procedure must assume the data parameter is a cooked command message. For IPMI interfaces, this is defined as the following series of numeric values:

```
<dest> <netfn> <lun> <cmd> <data1> <data2> ... <dataN>
```

These items are described in more detail in the IPMI specification.

3.1.1.2 Getting Replies

Specification:

```
proc tget { md {timeout ""} }
```

Parameters:

md Message descriptor as returned by tsend.

timeout A timeout value in units of milliseconds. Use a negative one to block. Use zero for a non-blocking call. Use a positive number for blocking with a timeout. An empty string selects the default value as specified with a call to ttimeout.

Return Value:

An empty Tcl list on a timeout, or a list of the form {ecode data}:

ecode An error code: Zero on success; non-zero on an error.

data Command reply data in the form of a Tcl list of numeric values. (meaningless for a non-zero ecode).

Implementation Notes:

The data item in the returned list must be formatted as a cooked reply, which is defined as a list of numeric values:

```
<completion code> <data1> <data2> ... <dataN>
```

The complete definition of a command reply can be found in the IPMI specification.

Transport modules do not actually have to support millisecond timeout values, but at a minimum they must not block indefinitely if the timeout value is zero or positive.

3.1.1.3 Query Available Commands

Specification:

```
proc commands { }
proc commands { {ltran “”} }
```

Parameters:

`ltran` Logical transport name.

Return Value:

A Tcl list containing the names of every Tcl command available in the transport module, or in the second case, for the specified logical transport.

Implementation Notes:

The first form of this procedure (without any parameters) must be implemented in transport modules that do not implement the `ltrans` procedure.

The second form of this procedure (with the `ltran` parameter) must be implemented for transport modules that do implement the `ltrans` procedure.

3.1.1.4 Query Interfaces

Specification:

```
proc interfaces { }
proc interfaces { {ltran “”} }
```

Parameters:

`ltran` Logical transport name.

Return Value:

A Tcl list containing the names of every supported IPMI interface for the module, or in the second case, for the specified logical transport. Table 3-1 lists the possible interfaces. A transport module is permitted have unreported non-IPMI interfaces.

Implementation Notes:

The first form of this procedure (without any parameters) must be implemented in transport modules that do not implement the `ltrans` procedure.

The second form of this procedure (with the `ltran` parameter) must be implemented for transport modules that do implement the `ltrans` procedure.

Table 3-1. Possible IPMI Target Interfaces

Name	Description
`I2C`	I^2C
`SMS`	System Mangement Software Interface (KCS, SMIC, or Block Transfer)

3.1.1.5 Opening a Transport

Specification:

```
proc topen { {portid ""} {options ""} }
proc topen { {portid ""} {options ""} {ltran ""} }
```

Parameters:

`portid` A port identifier. The meaning is transport-dependent. For example a serial transport module (RS-232) might accept "COM1" or "COM2" as the port ID. A LAN transport module might accept a TCP/IP address or a host name. An empty string selects the default port. A transport module is not required to support a default port. It may return an error instead.

`options` A list of opening options, entirely transport-dependent. For example a serial transport module might accept baud rates and such. An empty list selects the default options.

`ltran` Logical transport name.

Return Value:

A Tcl list of the form {`ecode pd`} where:

`ecode` An error code. Zero on success, non-zero on an error.

`pd` A port descriptor.

Usage Notes:

Don't assume that the port descriptor is a numeric value.

Implementation Notes:

It is up to the implementer of the transport module whether to support more than one opening at a time. Generally, if the module supports only an empty string for the `portid`, there's no need to support multiple openings. Port ID's must be case-insensitive.

The port descriptor (`pd`) doesn't have to be numeric although you may find this to be the most convenient form. Any combination of letters, digits, and underscore characters is allowed, however a transport module must never generate a port descriptor that is also a valid port ID for the same transport. Port descriptors should not be case-sensitive.

The first form of this procedure (without any parameters) must be implemented in transport modules that do not implement the `ltrans` procedure.

The second form of this procedure (with the `ltran` parameter) must be implemented for transport modules that do implement the `ltrans` procedure.

3.1.1.6 Closing a Transport

Specification:

```
proc tclose { pd }
```

Parameters:

`pd` A port descriptor as returned by `topen`.

Return Value:

Zero if the operation is successful. Non-zero on an error.

3.1.1.7 Sending Messages

Specifications:

```
proc tsend { pd iface[:node] data {options ""} }
```

Parameters:

`pd`	Port descriptor as returned by `topen`.
`iface`	The target interface.
`node`	The target bus node.
`data`	Command data in the form of a Tcl list of numeric values.
`options`	An empty string selects the default options. The letter "`n`" is the no-reply option indicating that no reply is to be expected.

Return Value:

A Tcl list of the form {`ecode md`} where:

`ecode`	An error code: Zero on success; non-zero on an error.
`md`	A message descriptor that can be used to get the message reply (meaningless for a non-zero `ecode` and the no-reply option).

Usage Notes:

Don't assume that the message descriptor (`md`) is numeric.

Implementation Notes:

The message descriptor (`md`) doesn't have to be numeric although you may find this to be the most convenient form. Any combination of letters, digits, and underscore characters is allowed. Message descriptors are case-sensitive.

The procedure must assume that the `data` parameter is a cooked command message. For IPMI interfaces this defined as the following series of numeric values:

```
<dest> <netfn> <lun> <cmd> <data1> <data2> ... <dataN>
```

These items are described in more detail in the IPMI specification.

3.1.1.8 Getting Replies

Specification:

```
proc tget { md {timeout ""} }
```

Parameters:

`md` Message descriptor as returned by `tsend`.

`timeout` A timeout value in units of milliseconds. Use a negative one to block. Use zero for a non-blocking call. Use a positive number for blocking with a timeout. An empty string selects the default value as specified with a call to `ttimeout`.

Return Value:

An empty Tcl list on a timeout, or a list of the form `{ecode data}` where:

`ecode` An error code: Zero on success; non-zero on an error.

`data` Command reply data in the form of a Tcl list of numeric values. (Meaningless for a non-zero `ecode`).

Implementation Notes:

The `data` item in the returned list must be formatted as a cooked reply, which is defined as a list of numeric values:

```
<completion code> <data1> <data2> ... <dataN>
```

The complete definition of a command reply can be found in the IPMI specification.

Transport modules do not actually have to support millisecond timeout values, but at a minimum they must not block indefinitely, if the timeout value is zero or positive.

3.1.1.9 Set/Get Timeout

Specification:

```
proc ttimeout { pd {timeout ""} }
```

Parameters:

`pd` A port descriptor as returned by `topen`.

`timeout` A value in milliseconds specifying the default value for subsequent calls to `tget` and `trawget`. An empty string means that no change should be made in the default timeout value (used to query the current value).

Return Value:

The procedure returns the previous default timeout value, or an empty string if the new value is invalid.

Implementation Notes:

Transport modules do not actually have to support millisecond resolution timeout values, but at a minimum they must not block indefinitely, if the timeout value is zero or positive.

3.1.1.10 Flushing Messages

Specification:

```
proc tflush { {pd ""} }
```

Parameters:

`pd` A port descriptor as returned by `topen`. An empty string indicates that the procedure should act on all its open ports.

Return Value:

Zero on success. Non-zero if there's an error.

Implementation Notes:

This procedure should release any resources associated with open message descriptors, free those descriptors, and flush any unread cooked replies (for `tget`) and raw messages (for `trawget`).

3.1.2 Raw Message Specification

Raw message support is an optional feature of transport modules. A given module may support raw messages for any subset of the interfaces it supports for cooked messages. For unsupported interfaces, the `trawsend` and `trawget` procedures must return a non-zero error code.

Raw message formats are interface-specific and transport-independent. Any transport modules providing raw message support for a specific interface must use the same message format as all other transport modules supporting the same interface.

3.1.2.1 SMS Raw Message Format

The raw message formats for SMS and 12C appear in this section.

An SMS raw command message is similar to the cooked message defined in the *Cooked and Raw Messages* section of this chapter. There are two differences. First, a raw SMS message does not include a destination address. SMS messages are always directed to the BMC. Second, the `NetFn` and `LUN` values are combined into a single byte.

The complete message format is as follows:

```
<NetFn/LUN> <Cmd> <Data1> ... <DataN>
```

where:

`NetFn` Network function, six most-significant bits.

`LUN` Logical unit number, two least-significant bits.

`Cmd` Command number.

`Data1 ...` Command data.

An SMS raw response message is similar to the raw command message.

```
<NetFn/LUN> <Cmd> <CCode> <Data1> ... <DataN>
```

where:

`NetFn`	Original network function incremented by one, six most-significant bits.
`LUN`	An echo of the logical unit number, two least-significant bits.
`Cmd`	An echo of the command number.
`CCode`	The command completion code.
`Data1 ...`	Command response data.

I2C Raw Message Format

A raw I2C command message contains two sets of bytes as follows:

```
<rsSA> <NetFn/rsLUN> <Check1>
<rqSA> <rqSeq/rqLUN> <Cmd> <Data1> <Data2> ... <DataN> <Check2>
```

where:

`rsSA`	Receiver's I2C slave address.
`NetFn`	Network function, six most-significant bits.
`rsLUN`	Logical Unit Number on the receiver, two least-significant bits.
`Check1`	One's compliment of the sum of the preceding bytes.
`rqSA`	Requester's I2C slave address (typically the BMC, which is 0x20).
`rqSeq`	Requester's message sequence number, six most-significant bits.
`rqLUN`	Requester's logical unit number, two least-significant bits (typically 0x2 for SMS).
`Cmd`	Command number.
`Data1 ...`	Command data.
`Check2`	One's compliment of the sum of the preceding bytes.

A raw I2C response message has a similar format:

```
<rqSA> <NetFn/rqLUN> <Check1>
<rsSA> <rqSeq/rsLUN> <Cmd> <CCode> <Data1> ... <DataN> <Check2>
```

where:

`rqSA`	Requester's I2C slave address.
`NetFn`	An echo of the network function from the original command, but incremented by one.
`rqLUN`	An echo of the requester LUN from the original command.
`Check1`	One's compliment of the sum of the preceding bytes.
`rsSA`	Receiver's I2C slave address.
`rqSeq`	An echo of the sequence number from the original request.

`rsLUN` Receiver's logical unit number.

`Cmd` An echo of the command number from the original request.

`CCode` Command completion code.

`Data1 ...` Command response data.

`Check2` One's compliment of the sum of the preceding bytes.

3.1.2.2 Sending Raw Messages

Specifications:

```
proc trawsend { pd iface[:node] data }
```

Parameters:

`pd` Port descriptor as returned by `topen`.

`iface` The target interface.

`node` The target bus node.

`data` Raw data in the form of a Tcl list of numeric values.

Return Value:

An error code: Zero on success; non-zero on an error.

Usage Notes:

The `trawsend` procedure is similar to `tsend` except that it does not perform any interface-dependent packaging of the data before being sent, and it does not allocate a message descriptor.

Implementation Notes:

Raw message format is defined above.

This procedure must not make any assumptions about the meaning of the data.

It is not required that a transport module support raw messages for all interfaces for which it supports cooked messages (`tsend`/`tget`). For unsupported interfaces `trawsend` should return a non-zero error code.

3.1.2.3 Getting Raw Messages

Specification:

```
proc trawget { pd iface[:iq] {timeout ""} }
```

Parameters:

`pd`	Port descriptor as returned by `topen`.
`iface[:iq]`	The target interface with optional interface qualifier.
`timeout`	A timeout value in units of milliseconds. Use a negative one to block. Use zero for a non-blocking call. Use a positive number for blocking with a timeout. An empty string selects the default value as specified with a call to `ttimeout`.

Return Value:

An empty Tcl list on a timeout, or a list of the form {`ecode data`} where:

`ecode`	An error code: Zero on success; non-zero on an error.
`data`	Raw data in the form of a Tcl list of numeric values where each item in the list represents a single byte (meaningless for a non-zero `ecode`).

Usage Notes:

The `trawget` procedure is similar to `tget` except that it does not perform any interface-dependent unpackaging of the data before returning it to the caller and it does not use message descriptors.

Implementation Notes:

Raw message format.

The transport module is permitted to queue messages as they arrive from the target machine, in which case this procedure should return the oldest unread message.

Transport modules do not actually have to support millisecond timeout values, but at a minimum they must not block indefinitely, if the timeout value is zero or positive.

It is not required that a transport module support raw messages for all interfaces for which it supports cooked messages (`tsend`/`tget`). For unsupported interfaces `trawget` should return a non-zero error code.

3.1.2.4 Logical Transports

Specification:

```
proc ltrans { }
```

Return Value:

This optional procedure returns a list of logical transport names that are in addition to the default behavior of the transport module.

Implementation Notes:

A single transport module may support more that one logical transport. For example, the FWHTRANS transport module, when loaded, virtualizes itself into a second transport module called FWH-I2C. This second "logical" transport modules supports the I2C interface (instead of SMS) when the FWHOST program on the target system is started with a different option.

If this procedure is implemented, then the `commands` procedure (described in the *Query Available Commands* section of this manual) and the interfaces procedure (described in the *Query Interfaces* section of this manual) must accept a logical transport name as a parameter.

Logical transport names must be case-insensitive.

3.1.2.5 Error Strings

Specification:

```
proc terror { ecode }
```

Parameters:

`ecode` An error code returned by a transport module routine.

Return Value:

A string describing the error or an empty string if the error code is not recognized.

3.1.2.6 Debug Levels

Specification:

```
proc tdebug { {level ""} {cmd ""} }
```

Parameters:

`level` The new debug level. An empty string indicates that you do not want to change the current level.

`cmd` The name of print command to use for printing debug messages. An empty string indicates that you do not want to change the current command. This command should be able to accept a string as its first and only parameter, and it should provide automatic new-line termination.

Return Value:

The procedure returns the previous debug level.

4 Libraries

This chapter contains procedure and test specifications for the libraries used by the framework or by a developer creating tests for use in the framework. This chapter contains sections listing the procedures available in the FRU, SDR, Micro controller, Wake on LAN, and SMS libraries. Each library section consists of a set of procedure and test specifications that describe the use and characteristics of each test or procedure.

4.1 FRU Library

4.1.1 ReadFRUData Read FRU Data

Specification:

```
proc ReadFRUData { offset length result {dlrt ""} }
```

Parameters:

`offset` A value. States the offset value of the FRU data to read. The offset value may be more than a byte.

`length` A value. The length of the FRU data to read, starting from the offset location. The length value may be more than a byte.

`result` A list. A name for the list containing the data after a successful read.

`dlrt` An optional list. Contains destination, LUN, route, and timeout information. The default destination is BMC (0x20), LUN 0, first available transport in the transport list, and the transport's default timeout.

Return Value:

`ecode` An error code. Zero on success and completion code as defined in *"Response Data Array"* on error.

Usage Note:

The device is accessed as bytes or words based on *"Get FRU Inventory Area Info"* response.

4.1.2 WriteFRUData Write FRU Data

Specification:

```
proc WriteFRUData { offset data {dlrt ""} }
```

Parameters:

`offset` A value. The offset of the FRU data to write. The offset value may be more than a byte.

`data` A list of data bytes to write. The length of the FRU data to write depends on the data list length.

`dlrt` An optional list. Contains destination, LUN, route and timeout information. The default destination is BMC(0x20), LUN 0, first available transport in the transport list, and the transport's default timeout.

Return Value:

`ecode` An error code. Zero on success and completion code as defined in *"Response Data Array"* on error.

Usage Note:

The device is accessed as bytes or words based on *"Get FRU Inventory Area Info"* response.

4.1.3 ReadFruNVRam Reading FRU Area

Specification:

```
proc ReadFruNVRam { areatype {dlrt ""} }
```

Parameters:

`areatype` A string. Identifies the FRU area. Valid strings are header, internal, chassis, `board, product and multirec.`

`dlrt` An optional list. Contains destination, LUN, route, and timeout information. The default destination is BMC(0x20), LUN 0, first available transport in the transport list, and the transport's default timeout.

Return Value:

`result` A list. Construct a result array by executing the `array_set` command. The result array contains `err` and `data` elements. The `err` element contains zero on success and non-zero on failure. The `data` element contains a list of data bytes on success.

Usage Notes:

This function reads a specified area of FRU from NVRAM.

4.2 SDR Library

4.2.1 ReadFullSdrRecord Reading a Complete SDR Record

Specification:

```
proc ReadFullSdrRecord { recid{dlrt ""} }
```

Parameters:

`recid` Record ID of the SDR to be read.

`dlrt` An optional list. Contains destination, LUN, route and timeout information. The default destination is BMC(0x20), LUN 0, first available transport in the transport list, and the transport's default timeout.

Return Value:

`rsplist` A response array list. Similar to the array returned by the `req_rsp` function. The `CompCode` element contains the error code. The `RecData` element contains the list of bytes on success.

4.2.2 ReadFullSdr Reading the Entire SDR

Specification:

```
proc ReadFullSdr { {dlrt ""} }
```

Parameters:

`dlrt` An optional list. Contains destination, LUN, route and timeout information. The default destination is BMC(0x20), LUN 0, first available transport in the transport list, and the transport's default timeout.

Return Value:

`sdrlist` A list suitable for conversion to an array. The array may be converted to a more useful form using `sdrDecode`.

4.3 SDR Utilities

4.3.1 sdrDecode Decoding SDR Data

Specification:

```
proc sdrDecode { encodedArray {ipmiVer ""} }
```

Parameters:

`encodedArray` Encoded array. SDR data in the form returned by `ReadFullSdr`.

`impiVer` IPMI version number. An empty string causes the value to be acquired from the `Platform_IPMI_Ver` configuration variable.

Return Value:

`decodedList` The list of decoded SDR data that may be converted to an array with the array set command. If the `ERR` element of this array is zero, the operation completed successfully. Non-zero indicates an error occurred.

Usage Note:

Normally test modules do not use this procedure directly. If the Checking SDR Utility Data Status variable, `Target_SDR_Source`, is set to `BMC_SDR`, library initialization automatically causes reading and decoding of the SDR information from the BMC. The read data is stored in an internal data array.

4.3.2 sdrGetMicrocontrollers Getting SDR Microcontrollers

Specification:

```
proc sdrGetMicrocontrollers { {decodedArray ""} {translate 1} }
```

Parameters:

`decodedArray` An array. Decoded SDR data in the form returned by `sdrDecode`. An empty string indicates the procedure should use internally-stored data created during library initialization.

`translate` A value. Non-zero number causes the conversion of long string names to pre-defined short-named aliases before returning. For example, the string name "`Basbrd Mngt Ctlr`" will convert to "`BMC`". Uses the defined aliases in the `Target_SDR_uC_Info` array.

Return Value:

`uCList` A list. Microcontroller names found in the SDR data.

Usage Note:

The preferred method for tests to get the list of microcontrollers is to use `ucDeviceList`.

4.3.3 sdrGetMicrocontrollerSlaveAddress Getting SDR Micro Addresses

Specification:

```
proc sdrGetMicrocontrollerSlaveAddress { deviceName {decodeArray ""} {translate 1} }
```

Parameters:

`deviceName` A string. The microcontroller name.

`decodedArray` An array. Decoded SDR data in the form returned by `sdrDecode`. An empty string indicates the procedure should use internally-stored data created during library initialization.

`translate` A value. Non-zero indicates that if the initial search fails, the procedure should search through the SDR data again. The first search uses the exact name specified by the `deviceName` parameter. The second search assumes that `deviceName` is a short-named alias for a long string name. The search uses the equivalent long name. For example, if `deviceName` is "BMC", the procedure will search on "BMC". If the first search fails, the procedure will search again on "Basbrd Mngt Ctlr".

Return Value:

`slaveAddr` An address. The slave address of the microcontroller. An empty string indicates the specified microcontroller was not found in the SDR data.

Usage Note:

The preferred method for tests to get microcontroller addresses is to use `ucSlaveAddress`.

4.3.4 sdrFlushCache Flush the SDR Decoded Array's Cache

Specification:

```
proc sdrFlushCache { {decodedArray ""} }
```

Parameters:

`decodedArray` An array. Decoded SDR data in the form returned by `sdrDecode`. An empty string indicates the procedure should use internally-stored data created during library initialization.

4.4 Microcontroller Library

4.4.1 ucDeviceList Getting a List of Microcontrollers

Specification:

```
proc ucDeviceList { }
```

Return Value:

`uCList` A list of microcontroller names

Usage Note:

The `Target_SDR_Source` variable determines the `Target_SDR_Source` variable. If set to "`Target_Config`", the information is obtained from the `Target_SDR_uC` configuration variable. If set to "`BMC_SDR`" it acquires information from the SDR records in the BMC. If set to an empty string or left unset, it gets information from the default source, which depends on the particular installation of FTF.

4.4.2 ucSlaveAddress Getting Microcontroller Addresses

Specification:

```
proc ucSlaveAddress { deviceName }
```

Parameters:

`deviceName` A string. A microcontroller name.

Return Value:

`slaveAddr` The device slave address. An empty string if an address for the named controller is not found.

Usage Note:

See the discussion under `ucDeviceList` in the previous section.

4.4.3 ucDeviceName Getting Microcontroller Names

Specification:

```
proc ucDeviceName { slaveAddr }
```

Parameters:

`slaveAddr` The device slave address.

Return Value:

`deviceName` A string. The name for a microcontroller. An empty string indicates a matching device name was not found.

Usage Note:

This is a convenience routine that uses a combination of `ucDeviceList` (described in the *`ucDeviceList` Getting* a List of Microcontrollers section of this chapter) and `ucSlaveAddress` (described in the *`ucSlaveAddress`* Getting Microcontroller Addresses section of this chapter) to perform a reverse lookup on the slave address to find the corresponding name.

4.4.4 ucDefaultMicro Getting the Default Microcontroller

Specification:

```
proc ucDefaultMicro { }
```

Return Value:

`deviceName` A string. The device name for the default microcontroller, as selected by the user via the framework's user interface.

Usage Note:

This feature of the framework is not utilized by the ICTS test modules. Instead of operating on the default micro controller, they operate on all microcontrollers identified in the SDR.

4.5 Wake On LAN† Library

4.5.1 Loading the Library

Unlike most of the other firmware libraries, the framework does not automatically load the Wake On LAN library during initialization. A test must request the library with the following procedure call:

```
ftf_requirelib lib_wol
```

This call returns a zero if the library loads successfully; otherwise, it returns a non-zero. wolSendMagicPacket Sending a "Magic Packet†"

Specification:

```
proc wolSendMagicPacket { ieeeAddr ipAddr }
```

Return Value:

`ieeeAddr` A value. The IEEE address (a.k.a. MAC address, a.k.a. Ethernet address) of a network adapter supporting "Magic Packet" technology. The address notation is "xx:xx:xx:xx:xx:xx". Each "xx" represents a two-digit hexadecimal number.

`ipAddr` A value. The broadcast IP address in “dot” notation for the network where the network adapter resides. This may also be a host name that resolves to a broadcast address.

Usage Note:

This procedure sends a Wake-On-LAN “Magic Packet” to the specified network adapter on the specified network.

Legal Note:

Magic Packet is a trademark of Advanced Micro Devices.

4.6 SMS Library

4.6.1 Loading the Library

The framework does not automatically load the SMS library during initialization. A test must request the library with the following procedure call:

```
ftf_requirelib lib_sms
```

This call returns a zero if the library loads successfully; otherwise it returns a non-zero.

4.6.2 smsWrapForNonBmcMicro Wrapping Non-BMC Messages

Specification:

```
proc smsWrapForNonBmcMicro { message {seq ""} {broadcast 0} }
```

Parameters:

`message` A "cooked" message. The message form must be suitable for either the `msend` procedure of the framework's message library or the `tsend` procedure of a transport module.

`seq` A sequence number. Only the six least significant bits are used. If an empty string is passed, the procedure will generate a sequence number on its own.

`broadcast` A value. Non-zero causes a request that the message be wrapped for broadcasting.

Return Value:

`wmessage` An encapsulated message. The message is encapsulated in a `WriteI2C` command (IPMI 0.9 or less) or a `SendMessage` command (IPMI 1.0 or greater) suitable for sending to an SMS interface through the `msend` or `tsend` procedures.

Usage Note:

The functionality contained in this procedure is built in to most transport modules that support an SMS interface. However the broadcast option is not typically built in to transport modules, so this is the only mechanism currently provided for constructing broadcast messages.

4.6.3 smsUnwrapNonBmcResponse Unwrapping Non-BMC Responses

Specification:

```
proc smsUnwrapNonBmcResponse { wresponse }
```

Parameters:

`wresponse` A wrapped response. The response is acquired by either the `ReadSMSBuffer` command (IPMI 0.9 or less) or the `GetMessage` command (IPMI 1.0 or greater) as returned by either `mget` of the framework message library or `tget` of a transport module.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`response` The unwrapped response, in “cooked” form.

Usage Note:

The functionality contained in this procedure is built in to most transport modules supporting an SMS interface. However, you will need this procedure to unwrap broadcast responses since most transport modules do not support broadcast messages.

4.6.4 smsSendNonBmcMessage Sending Non-BMC Messages

Specification:

```
proc smsSendNonBmcMessage { message {seq ""} {broadcast 0} }
```

Parameters:

`message` A “cooked” message suitable for either the `msend` procedure of the framework’s message library or the `tsend` procedure of a transport module.

`seq` A sequence number. Only the six least significant bits are used. If an empty string is passed, the procedure generates a sequence number on its own.

`broadcast` If non-zero, the procedure requests that the message be wrapped for broadcasting.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`rd` A request descriptor, which may be used with `smsGetNonBmcMessage`.

Usage Note:

This is a convenience procedure that provides an `msend`-style interface for sending non-BMC messages.

4.6.5 smsGetNonBmcMessage Getting Non-BMC Messages

Specification:

```
proc smsGetNonBmcMessage { rd }
```

Parameters:

`rd` A request descriptor. The descriptor takes the same form as one returned from `smsSendNonBmcMessage`.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`response` The message response in “cooked” form.

Usage Note:

This is a convenience procedure that provides an `mget`-style interface for getting the responses to non-BMC messages. It performs either a `ReadSMSBuffer` command (IPMI 0.9 or less) or a `GetMessage` command (IPMI 1.0 or greater) in order to get the response.

4.6.6 smsSendMessage Sending SMS Messages

Specification:

```
proc smsSendMessage { message }
proc smsBroadcastMessage { message }
```

Parameters:

`message` A "cooked" message. The message is in a form suitable for either the `msend` procedure of the framework's message library or the `tsend` procedure of a transport module.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`rd` A request descriptor that may be used with `smsGetMessage`.

Usage Note:

The `smsSendMessage` procedure calls `smsSendNonBmcMessage` or `msend`, depending on the target address. Both BMC and non-BMC messages are supported.

The `smsBroadcastMessage` procedure is for broadcast messages only. It does not support BMC commands.

4.6.7 smsGetMessage Getting SMS Messages

Specification:

```
proc smsGetMessage { rd }
```

Parameters:

`rd` A request descriptor. The descriptor takes the same form as one returned from `smsSendMessage` or `smsBroadcastMessage`.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`response` A message response in "cooked" form.

Usage Note:

The calling procedure should wait at least 60 milliseconds after sending a message before calling this procedure to read the response. Use the Tcl after command to create the delay.

4.6.8 smsSendGetMessage Send/Get SMS Message

Specification:

```
proc smsSendGetMessage { message }
```

Parameters:

`message` A "cooked" message. The message is in a form suitable for the either the `msend` procedure of the framework's message library or the `tsend` procedure of a transport module.

Return Value:

A list containing the following items:

`ecode` An error code. Zero indicates success.

`response` A message response in "cooked" form.

Usage Note:

This convenience procedure does `smsSendMessage` and `smsGetMessage` in sequence, including a 60ms delay between them.

4.6.9 Generic Library Functions

This section contains functions frequently used by test modules.

4.6.10 array_set Creating a New Array

Specification:

```
proc array_set { arrname datalist }
```

Parameters:

`arrname` Name of the array to be created.

`datalist` A format list to determine the form of the new array. The list is similar to the list `req_rsp` returns.

Return Values:

A value returned by the `"array_set"` command.

Usage Notes:

The Tcl command array set creates an array from a list. If an array is already created and this command is used to create another array using the previously created array name, the new elements are added to the existing array. Old elements are not deleted. The `array_set` library procedure unsets the previously created array and creates a new array from the list.

Use the `array_set` procedure for `req_rsp` rather than the array set command.

Soliciting input for returning completion code or message library error code by this function, if code can take advantage by testing for return code while making this call.

4.6.11 print_pass Test Pass Message

Specification:

```
proc print_pass { message }
```

Parameters:

`message` A string to display.

Return Values:

None

Usage Notes:

Use this function to display the pass message and increment the counter associated with the number of passes in the test.

Example: `print_pass "Sel Get Info executed"` call displays the following pass message and increments the pass counter.

```
PASS: Sel Get Info executed
```

4.6.12 print_fail Test Fail Message

Specification:

```
proc print_fail { message {rsp_arr_name "" } }
```

Parameters:

`message` A string to display.

`rsp_arr_name` Response array returned by the `req_rsp` function to display response-specific error messages.

Return Values:

None

Usage Notes:

The function displays the fail message and increments the counter associated with the number of failures in the test.

Example: print_fail "Unable to clear SEL entries" call displays the following fail message and increments the fail counter.

```
FAIL: Unable to clear SEL entries
```

Example: `print_fail "Unable to clear SEL entries" selrsp` call displays the following fail message and increments the fail counter.

```
FAIL: Unable to clear SEL entries
SelClear returned 0xc1, Invalid command
```

4.6.13 print_warn Test Warning Message

Use this function to display a warning message.

Specification:

```
proc print_warn { message }
```

Parameters:

`message` A string to display.

Return Values:

None

Usage Notes:

Use this function to display a warning message.

Example: `print_warn "Check the firmware mode, auto detection not implemented"` call displays the following fail message.

```
WARN: Check the firmware mode, auto detection not implemented
```

4.6.14 print_na Test Not Applicable Message

Use this function to display a non-applicability message.

Specification:

```
proc print_na { message }
```

Parameters:

`message` A string to display.

Return Values:

None

Usage Notes:

Use this function to display a warning message.

Example: `print_na "Warm Reset not implemented. Test skipped."` Call displays the following fail message and increments the not-applicable counter.

```
N/A: Warm Reset not implemented. Test skipped.
```

4.6.15 get_test_fail_count Get Test Fail Count

Specification:

```
proc get_test_fail_count { }
```

Return Values:

`fail_count` Number of failures in the test module in which this function is called.

Usage Notes:

Use this function to get a test failure count from the counter for the current test.

A test may return this number to the parent module to indicate a test is passed or failed.

4.6.16 print_in_hex Print in Hexadecimal Format

Specification:

```
proc print_in_hex { data {mode "n"} {options ""} }
```

Parameters:

`data` A list on bytes.

`mode` The letter `n, d,` or `v`. The mode flags stand for normal, debug and verbose, respectively. The default mode is d0 and v0. Valid values are `d0, d1, d2, v0, v1` and `v2`.

`options` Tags to cause `nonewline` or color. Refer to the `ftf_msg` function for detailed values.

Return Values:

None

Usage Notes:

Use this function to display the given bytes in a format similar to the debug program.

Example:

```
0000  01 01 0E 12 1A 00 00 C4-01 00 00 00 00 00 00 00 ................
0010  00 A9 02 00 00 54 65 68-00 00 00 00 00 00 00 00 .....Teh........
0020  0E 00 0E 00 45 69 5B 36-41 77 5B 36 0F 02 0A 0B ....Ei[6Aw[6....
```

4.6.17 print_line Displaying a Line

Specification:

```
proc print_line { {char "-"} {width "80"} {mode "n"} {options ""} }
```

Parameters:

`char` A character to be used to display a line.

`width` A value that specifies the line width.

`mode` A string specifies the debug or verbose level. Refer to the `ftf_msg` function for details.

`options` A string specifies the color and `nonewline` options. Refer to the `ftf_msg` function for details.

Return Value:

None

4.6.18 DateTime Date and Time Formatting

Specification:

```
proc DateTime { {time “”} {gmt 0} }
```

Parameters:

`time` An optional time value in seconds since January 1, 1970. Default is to return the current date and time.

`gmt` Flag, if non-zero, suppresses any conversion to local time. Useful when you want the formatted string to represent GMT time, or when the input `time` value is already in local time.

Return Value:

`str` Date and Time of the format `MM/DD/YY HH:MM:SS` is returned.

4.6.19 LocalSeconds Local Time in Seconds

Specification:

```
proc LocalSeconds { }
```

Return Value:

`seconds` The local time in seconds since January 1, 1970.

Usage Notes:

The value returned by the procedure is suitable for use as an SEL time. The return value of the Tcl `clock seconds` command should not be used for SEL time because its value is GMT.

4.6.20 formatx Formatting a Value in Hexadecimal

Specification:

```
proc formatx { value {fm “”} }
```

Parameters:

`value` A value to be formatted.

`fm` Number of digits after formatting.

Return Value:

`str` A numeric string in hexadecimal notation.

Example:

```
value 0x10     output 0x10
value 10       output 0x0A
value 0x102 output 0x0102
value 256      output 0x0100
```

Usage Notes:

The prefix string ("0x") may be overridden with the user configuration variable `User_Numeric_Prefix(16)`.

4.6.21 formatb Formatting a Value in Binary

Specification:

```
proc formatb { value {fm ""} }
```

Parameters:

`value` A value to be formatted.

`fm` Number of digits after formatting.

Return Value:

`str` A numeric string in binary notation.

Example:

```
value 0x10    output 00010000
value 10      output 00001010
value 0x102   output 0000000100000010
value 256     output 0000000100000000
```

Discussion:

A prefix string may be added with the user configuration variable `User_Numeric_Prefix(2)`.

4.6.22 heart_beat Progress Indicator – Heart Beat

Specification:

```
proc heart_beat { {mode "n"} {abort_check "no"} }
```

Parameters:

`mode` An optional verbose or debug state.

`abort_check` An optional indication for `ftf_stopcheck`

Return Value:

`abort` A non-zero value is returned if an abort is requested by the user. Value 0 otherwise.

4.6.23 print_arr Displaying Array Elements

Specification:

```
proc print_arr { array_name }
```

Parameters:

`array_name` A string. The name of the array to display.

Return Value:

None

Usage Note:

This is a convenience function for debugging.

The function prints an element name, a tab character, and its value in a single line.

4.6.24 Converting Byte List to String

Specification:

```
proc bytes_to_string { datalist }
```

Parameters:

`datalist` A list of byte values.

Return Value:

`str` A string.

4.6.25 get_hex_list Converting Bytes to Hexadecimal List

Specification:

```
proc get_hex_list { data }
```

Parameters:

`data` A list of byte values.

Return Value:

`hex_list` A list of hexadecimal bytes.

4.6.26 compare_byte_lists Compare List of Bytes

Specification:

```
proc compare_byte_lists { list1 list2 {exclude ""} }
```

Parameters:

`list1` A list of byte values.

`list2` A list of byte values.

`exclude` Optional list. Contains byte locations that should be excluded while doing the compare. Default is to compare all elements.

Return Value:

`result` Zero indicates that both lists are matched. Non-zero indicates a mismatch.

4.6.27 concat_chars Padding Characters to a String

Specification:

```
proc concat_chars { msg length {char "."} }
```

Parameters:

`msg` A character string.

`length` Number of bytes to pad the characters.

`char` Optional. Represents the characters to pad..

Return Value:

`msg` Padded character string.

4.6.28 set_child_options Setting Child Test Options

Specification:

```
proc set_child_options { options }
```

Parameters:

`options` A list of option keywords. The following list contains the possible option keywords and their meanings:

- `summary` Default. Enable pass/fail summary at the end of each child test.
- `nosummary` Disable pass/fail summary at the end of each child test.

Return Value:

None

Usage Notes:

Called this procedure from a parent test `test_main` before starting the child test. Options set by this procedure are retained until completion of the parent test. On completion, the options return to the system defaults.

4.6.29 get_checksum Get Checksum

Specification:

```
proc get_checksum { bytelist {size "byte"} }
```

Parameters:

`bytelist` A list of byte values.

`size` A string. The string "`byte`" indicates the return value is a byte checksum. The string "`word`" indicates the return value is a word checksum.

Return Value:

`checksum` A byte or word check sum based on the size indicator. A null value is returned if the input list is empty.

5 Tcl Namespace Considerations

When creating tests and making use of existing Tcl modules, Tcl namespace becomes an issue. Tcl assumes certain characteristics for the development of tests using Tcl commands. This section contains information about the use of Tcl modules and the development of procedures and tests that may influence or be influenced by Tcl namespace issues.

The framework loads test modules and transport modules into Tcl namespaces. Test module implementation requirements arise from the characteristics of the framework. This section contains a list of the namespace considerations for creating new FTF test modules.

Generally, use the `global` command to access variables in top-level namespace, including Tcl built-in variables (`env`, `tcl_platform`). However, this does not include the host, platform, and target variables. Use `ftf_getglobal` to access configuration file variables. For additional information on configuration file variables, refer to the Installation and Configuration chapter of the *Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User Guide*.

Avoid creating a circumstance where a transport module or test module makes assumptions about being loaded to explicitly named namespace. If you cannot avoid such circumstances, the `namespace current` command provides access to load and naming information.

The following is a list of namespace considerations discussed in this section:

- Variables initialized outside test module procedures may cause conflicts with global framework variables if correct precautions are not taken.
- Arrays initialized outside test module procedures may cause conflicts with global framework array variables if correct precautions are not taken.
- Accessing externally-initialized variables from within a test module can create conflicts if precautions are not taken.
- Test modules should not access framework global variables directly. Use the `ftf_getconfig` procedure provided by the framework (described in the *ftf_getconfig Checking for Configuration Variables* section of this manual).
- Test modules can access Tcl built-in global variables directly. The `env` array containing the environment variable settings is an example.
- Test modules must never modify variables outside their own namespaces, except as provided by the framework's API.

5.1 Setting Variables Outside a Procedure

For Tcl modules, avoid using the `set` command outside of procedures. If, and only if a global variable already exists, the `set` command references the global variable instead of a namespace variable. To avoid this resolution of ambiguous naming, use the `variable` command to initialize variables outside of procedures.

To define arrays outside procedures, use the `variable` command first, then set individual array elements with the `set` command or the `array set` command.

5.1.1 Example: Initializing Arrays with the variable Command

```
variable Test_VDT
set Test_VDT(x) [list "10.0" "X Coordinate" f]
set Test_VDT(y) [list "5.0" "Y Coordinate" f]
```

5.2 Global Variables

Use the `variable` command to reference a namespace module's "global" variables. Using the `global` command inside procedures allows access only to the variable within the namespace. Variables defined as global within a namespaced module are not available outside that module.

5.2.1 Example: Accessing Globals with the variable Command

```
variable bar 1
proc foo { } {
      variable bar
      puts $bar
}
```

WARNING

The following example provides undefined results.

```
variable bar 1
proc foo { } {
     global bar
     puts $bar
}
```

5.3 Non-array Variable Initialization Outside Test Module Procedures

Use the `variable` command instead of the `set` command to initialize non-array variables outside of test module procedures. Using the `variable` command prevents potential conflict with framework global variables.

If a test module variable matches the name of a framework global variable, using the `set` command modifies the framework's global variable rather than creating the variable within the test's own namespace.

5.3.1 Example: Variable Initialization

```
variable Test_Interfaces [list SMS I2C]
proc test_main { } {
# Test body
}
```

5.4 Array Initializing Outside Procedures

Use the `variable` command to create the array variable; then use the `set` or `array set` commands to set the individual elements.

5.4.1 Example: Array Initialization

```
variable Test_Interfaces [list SMS I2C]
variable Test_VDT
set Test_VDT(x) [list 1.0 "X Coordinate" f]
proc test_main { } {
# Test body
}
```

5.5 Accessing Externally-defined Variables Within Test Module Procedures

Use the `variable` command instead of the `global` command. The `global` command creates a reference to a framework global variable instead of a test module namespace variable.

5.5.1 Example: Variable Access Within a Module

```
variable Test_Interfaces [list SMS I2C]
variable Test_VDT
set Test_VDT(x) [list 1.0 "X Coordinate" f]
proc test_main { } {
variable Test_Interfaces
variable Test_VDT
# Test body
}
```

5.6 Namespace and Loadable C Modules

By default, the functions `Tcl_CreateCommand` and `Tcl_CreateObjCommand` install commands in the global namespace, not the current namespace. To install commands in the current namespace, as required for transport modules, use the `Tcl_Eval` function and the `namespace current` command to determine the current namespace, then use the returned information to prefix your newly created procedure names.

6 Procedures and APIs

This chapter provides detailed information on pre-existing procedures and application program interfaces. The chapter lists procedure specifications by group. Each group has unique characteristics that set it apart from the others.

6.1 Test Module Procedure Specifications

This section contains *Test Module* procedure specifications. Each specification appears under a header describing its purpose. Syntax, parameter lists, characteristics, and descriptions appear in each specification.

6.1.1 test_setup Post-Sourcing Initialization

Specification:

```
proc test_setup { }
proc test_setup { {statelist ""} }
```

Parameters:

`statelist` A list created by a prior call to the test's `test_state` procedure.

Return Value:

This procedure must return zero if initialization succeeds and non-zero if it fails.

Usage Notes:

Optional. Completes initialization needed once-per-test module loading. Do repeated initializations in `test_init`.

The second form of the procedure is required only if `test_state` is implemented.

6.1.2 test_help Test Help

Specification:

```
proc test_help { }
```

Return Value:

None

Usage Notes:

Optional. Uses `ftf_msg` or other output procedures to display help information. The `test_help could even spawn an external program to display help for the test, such as the` Windows† Help utility.

6.1.3 test_init Pre-Execution Initialization

Specification:

```
proc test_init { }
```

Return Value:

This procedure returns zero if initialization succeeds and non-zero if it fails.

Usage Notes:

Optional. Completes initialization needed each time the test is run. Use the `test_setup` procedure for initialization needed once-per-load.

6.1.4 test_main Running the Test

Specification:

```
proc test_main { }
```

Return Value:

This procedure must return zero if the test passes and a non-zero if the test fails. By convention, a positive number signifies that the test carried through to completion but produced an unexpected result. A negative number indicates the test could not complete or is not applicable. To be clear, the test should use the `ftf_msg, print_pass, print_fail, and print_na` procedures to provide result details.

Usage Notes:

Optional. All executable test modules have a `test_main`. It is possible to create a non-executable test module that contains only a `test_setup` procedure and/or the `Test_Children` variable.

6.1.5 test_state Saving the Test State

Specification:

```
proc test_state { }
```

Return Value:

This procedure must return a list, the contents of which are determined entirely by the test. Normally, the list contains state information that includes the test's configuration options and settings. Restore settings by passing the returned list to `test_setup`.

Usage Notes:

Optional. As part of the returned list, the test may also include state information for child tests as obtained using `ftf_stateoftest`.

6.2 Base Framework API

This section contains specifications for functions provided to the *Test Module* by the *Base Framework*. The Base Framework functions manage message logging, process control, internal maintenance tests, and coordination of child-parent relationships.

6.2.1 Message Logging

The message logging system allows a test module to write text messages to the screen and to a test log file. The system automatically time-stamps log file messages.

The following list contains the three messaging modes and descriptions of their characteristics:

- **Normal** —written to output destination without conditions
- **Verbose** —written if effective verbose level is sufficiently high
- **Debug** —written if effective debug level is a sufficiently high

For verbose and debug, the effective level is set either locally or globally through the interface or through the test during loading. The system recognizes only the greater value of the global and local level.

Normal and verbose messages display test progress, status, or results for the benefit of the test user.

Debug messages display a test module's or test framework's internal workings for the benefit of the test developer.

6.2.1.1 ftf_setlevel Local Message Level Control

Specification:

```
proc ftf_setlevel { mode {level ""} }
```

Parameters:

`mode` The letter `d` or `v` for debug and verbose modes, respectively.

`level` A digit, zero through three. An empty string indicates that no change is desired.

Return Value:

The procedure returns the previous level for the mode in order to allow restoration of the previous level at a later time.

Usage Notes:

The level set by this procedure is a local value and applies only to the calling test. The effective level is the greater value of the local level and the global level. The effective level can be queried using `ftf_getlevel` procedure described in the ftf_getlevel Effective Message Level Query section of this manual.

Well-behaved test modules record the beginning message level and restore the level to that original value before completion.

6.2.1.2 ftf_getlevel Effective Message Level Query

Specification:

```
proc ftf_getlevel { mode }
```

Parameters:

`mode` The letter `d` or `v` for debug and verbose modes, respectively.

Return Value:

The effective debug or verbose level, which is the greater value of the global level and the test's local level.

Usage Notes:

You can query the level at any time. Default values are built into the framework if they are otherwise not set.

6.2.1.3 TEST_STATE Writing Test Steps

Specification:

```
proc TEST_STATE { stateDesc warnings }
```

Parameters:

`stateDesc` A description to a user that declares the current activity of a test.

`warnings` A description indicating any limitations or other critical information in the given test state. The warning message is shown in red.

Return Value:

none

Usage Notes:

A message of "No Warnings" will not produce a warning message.

6.2.1.4 ftf_msg Family for Writing Messages

Specification:

```
proc ftf_msg { {message ""} {mode "n"} {options ""} }
proc ftf_log { {message ""} {mode "n"} {options ""} }
proc ftf_out { {message ""} {mode "n"} {options ""} }
proc ftf_status { {message ""} {mode "n"} }
proc ftf_clear { {mode "n"} }
proc ftf_bell { {mode "n"} }
```

Parameters:

`message` A text message. Only writes to the screen. Automatically appends a trailing new-line character (\n) unless suppressed in the `options` parameter. Allows embedded new-line characters.

`mode` The letter `n`, `d`, or `v` for normal, debug, and verbose modes, respectively. In the case of `d` and `v`, an optional digit, zero through three, represents the effective level number. The following equivalencies apply:

```
d0 = v0 = n
d1 = d
v1 = v
```

`options`	A Tcl list containing one or both of the following options:	
	`nonewline`	Suppress the trailing new-line character.
	`tag=attribute`	Text display attribute (red, green, blue, reverse, normal, underline).

Depending on the display device, the framework reserves the right to ignore all text attribute options. Text attributes have no effect on the log file.

Return Value:

Zero for success and non-zero for failure. Failure can result if a log file is not open or if a problem occurs while attempting to write to the log file.

Usage Notes:

The `ftf_msg` procedure writes only to the scrolling window on the screen and to the host log file if the screen capture option is enabled.

The `ftf_log` procedure writes only to the target log file.

The `ftf_out` procedure writes to both the scrolling display and the target log file, irrespective of the state of the screen capture option.

The `ftf_status` procedure writes to the single-line status bar on the screen.

The `ftf_clear` procedure clears the scrolling display area.

The `ftf_bell` procedure generates an audible beep or ring, depending on the host's sound system.

WARNING

Do not use `ftf_log` or `ftf_out` until the test module's `test_setup` procedure is called.

6.2.1.5 ftf_flush Flushing Output

Specification:

```
proc ftf_flush { }
```

Parameters:

None

Return Value:

None

Usage Notes:

Flushes all framework text output streams, including log files and standard output.

6.2.1.6 ftf_error Error Strings

Specification:

```
proc ftf_error { ecode {route ""} }
```

Parameters:

`ecode` An error code returned by a framework routine.

`route` An associated message route, if known.

Return Value:

A descriptive string for the error code and a message describing the error event sent to the destination provided by the route parameter.

6.2.2 Process Control

This section provides procedure specifications for the framework process control routines. The process control procedures allow querying of the user interface, the framework's internals state, and current configuration values to allow a test in progress to make control flow decisions.

6.2.3 ftf_stopcheck Checking for a Stop Event

Specification:

```
proc ftf_stopcheck { }
```

Return Value:

A non-zero value if a stop has been requested.

Usage Notes:

During long tests, call this procedure periodically to allow user-requested test interruption.

This procedure interprets CAD close requests as stop requests. Use the `ftf_eventcheck` procedure to avoid this characteristic.

6.2.3.1 ftf_eventcheck Checking for a User Interface Event

Specification:

```
proc ftf_eventcheck { }
```

Return Value:

A list suitable for `array set` conversion. The resulting array may contain the following elements:

`STOPREQ`	A non-zero number if the user made an explicit stop request.
`CADCLOSEREQ`	A list of handles to cursor-addressable displays for which the user has made a close request.

Usage Notes:

After the test stops, the framework automatically closes all displays listed in the `CADCLOSEREQ` list.

Unlike `ftf_stopcheck,` the test is not expected to stop itself with a CAD close request. It may stop itself, or it may close the specified CADs and continue.

6.2.3.2 ftf_requirelib Library Requirements

Specification:

```
proc ftf_requirelib { libNames }
```

Parameters:

`libNames`	A list of FTF library names. The elements of `libNames` do not include extensions, but they do include paths relative to the FTF library installation directory.

Return Value:

Zero, if all of the requested libraries have been successfully loaded; otherwise, a non-zero number.

Usage Notes:

Most libraries load at framework startup; however, a few infrequently-used libraries do not automatically load. Tests use this procedure to declare the need for a particular library. It is conceptually similar to the Tcl `package require` command, except there is no deferred loading.

6.2.4 Managing Other Tests

The Firmware Test Framework allows FTF-conformant tests to load and execute other FTF-conformant tests. This section lists the API routines for managing these relationships.

Table 6-1 defines the terms applied to tests spawned by other tests.

Table 6-1. Types of Tests

Type of Test	Description
Standard	A test that does not execute other tests
Parent	A test that executes other tests

Child	A test executed by another test. Both standard tests and parent tests may be the children of other parents. Children are not necessarily aware that they are children.

6.2.4.1 Parent-Child API Usage

The framework provides an API for parent tests to manage their children. Table 6-2 lists the procedures and their uses:

Table 6-2. Parent to Child Relationship

In parent:	Use these API calls for the child tests:
`test_setup`	`ftf_loadtest`[1], `ftf_setuptest`
`test_init`	`ftf_inittest`[2]
`test_main`	`ftf_inittest`[2], `ftf_runtest`,`ftf_cleanuptest`[2]
`test_cleanup`	`ftf_cleanuptest`[2]
`test_state`	`ftf_stateoftest`

6.2.4.2 Parent-Child Requirements

When managing parent-child relationships, the considerations in the following list must be observed:

1. For parent tests with a fixed set of child tests, as opposed to a dynamic set determined by a setup file or other means, the `Test_Children` variable may be used to indicate which children should be auto-loaded by the framework.
2. Parent tests may use, but do not require, `ftf_inittest` and `ftf_cleanuptest`. The parent must set up properly for its child before calling `ftf_runtest`, but it need not use `ftf_inittest`. Similarly, a parent must clean up after a child, but it need not use `ftf_cleanuptest` immediately after `ftf_runtest` if an undisturbed system state is desired. The parent must properly clean up everything if its own `test_cleanup` procedure is called.

6.2.4.3 tf_loadtest Loading a Child Test

Specification:

```
proc ftf_loadtest { module {options ""} {statelist ""} }
```

Parameters:

`module`	A test module name, with or without the file name extension. The name may include an absolute or relative path. The framework resolves relative paths using the `Host_TestDirs` variable.
`options`	Any combination of the following letters:
`h`	Hide this module from the framework's user interface.
`statelist`	A list created by a prior call to `ftf_stateoftest` for the same module. Restoring a child's state using `statelist` unless the child was loaded via the `Test_Children` variable. If `Test_Children` was used, restore the state via the `ftf_testsetup` procedure.

NOTE

When using `statelist`, a follow up call to `ftf_testsetup` for the same child is not generally required because restoring the state includes restoring the setup information.

Return Value:

Zero if the load succeeds. Non-zero for failure.

Usage Notes:

This procedure first seeks the test module in the parent's load directory. If the test module is not in the parent load directory, the procedure traverses the directories listed in `Host_TestDirs`. In each directory, the procedure looks first for the exact name test module, then for the test name with a `.tcl` or `.tcb` extension.

Tests normally use this procedure inside their own `test_setup` procedure. Follow this with a call to `ftf_setuptest`.

If the specified test has been previously loaded, the loaded version is discarded and replaced.

When a parent loads a child test, the parent gets its own personal instantiation of the child, independent of other instantiations created by other parents.

The `Test_Children` variable may be used instead of this procedure to load a set of child tests. Using `Test_Children` is preferred when the set of child test modules is fixed. Using `ftf_loadtest` is preferred when the set of child tests is dynamic.

6.2.4.4 ftf_setuptest Setting Up a Child Test

Specification

```
proc ftf_setuptest { module {setupname ""} {varlist ""} {statelist ""} }
```

Parameters:

`module` The loaded test module name. The name does not allow a path or an extension.

`setupfile` The name of a test setup file. The test setup file allows optional extensions and absolute or relative paths. If `setupfile` is an empty string, previous setup files are not re-sourced. The framework resolves relative paths using the `Host_SetupDirs` variable.

`varlist` A list of Tcl variables of the form `var=value`. These are defined after the new setup file is sourced and before the module's `test_setup` procedure call.

`statelist` A list returned by `ftf_stateoftest` for the same test module. Generally, it is preferable to pass `statelist` as a parameter to `ftf_loadtest` and avoid a call to `ftf_setuptest`, but for tests loaded via the `Test_Children` variable that method is not possible.

Return Value:

A list of two values: { `err ierr` }

`err` Zero if re-initialization was performed; otherwise, non-zero.

`ierr` The return value of the module's `test_setup` procedure, if any.

Usage Notes:

This procedure is normally called after a call to `ftf_loadtest` for the same test.

The setup process applies only to the parent's personal instantiation of the child, not to children of the same name instantiated by other parents.

6.2.4.5 Initializing a Child Test

Specification:

```
proc ftf_inittest { module {varlist ""} }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

`varlist` A list of Tcl variables of the form `var=value`. These are defined before the module's `test_init` procedure is called.

Return Value:

A list of two values: { `err perr` }

`err` Zero if re-initialization was performed; otherwise, non-zero.

`perr` The return value of the module's `test_init` procedure, if any.

Usage Notes:

Normally used in the parent's `test_main` procedure in combination with `ftf_runtest` and with `ftf_cleanuptest`. An alternative scheme is to call `ftf_inittest` in the parent's `test_init` procedure.

6.2.4.6 ftf_runtest Running a Child Test

Specification:

```
proc ftf_runtest { module {options ""} }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

`options` A series of one or more letters as follows:

`m` Mute the test. Suppress output.

Return Value:

A list of two values: { `err merr` }

`err` Zero if the test was executed. Non-zero otherwise.

`merr` The return value of the module's `test_main` procedure.

Usage Notes:

Use this procedure in the `test_main` procedure of the parent test.

This procedure does not call the child's `test_init` procedure prior to calling `test_main`.

6.2.4.7 ftf_cleanup Cleaning Up After a Child Test

Specification:

```
proc ftf_cleanuptest { module {varnames ""} }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

`varnames` A list of variable names to unset in the test's namespace.

Return Value:

A list of two values: { `err cerr` }

`err` Zero if clean up was performed; otherwise, non-zero.

`cerr` The return value of the module's `test_cleanup` procedure, if any.

Usage Notes:

Use `ftf_cleanup` in the parent's `test_main` procedure in combination with `ftf_inittest`, `ftf_runtest` and `ftf_cleanuptest`. To avoid `ftf_cleanuptest`, call `ftf_inittest` in the parent's `test_cleanup` procedure.

Well-behaved parent tests do not unset any child's variable that the parent did not set using `ftf_setuptest` or `ftf_iniittest`.

6.2.4.8 ftf_getvartest Querying Child Test Variables

Specification:

```
proc ftf_getvartest { module varname }
```

Parameters:

`module` A loaded test module name without a leading path or file name extension.

`varname` The name of one of the child's variables.

Return Value:

A list of two items: { `err value` }

`err` Zero if the value was retrieved; otherwise, non-zero.

`value` The value of the requested variable.

6.2.4.9 ftf_istest Querying Child Test Executable Types

Specification:

```
proc ftf_istest { module }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

Return Value:

The `ftf_istest` procedure returns a non-zero number if the specified module is loaded and contains a `test_main` procedure; otherwise, it returns a zero.

6.2.4.10 ftf_isloadedtest Checking for a Loaded Child

Specification:

```
proc ftf_isloadedtest { module }
```

Parameters:

`module` A (thought to be) loaded test module name without a leading path or a file name extension.

Return Value:

A non-zero number, if the specified module is loaded; otherwise, it returns a zero.

6.2.4.11 ftf_unloadtest Unloading a Child

Specification:

```
proc ftf_unloadtest { module }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

Return Value:

Zero, if the module was initially loaded and is now unloaded; otherwise, it returns a non-zero number.

6.2.4.12 ftf_stateoftest Saving a Child's State

Specification:

```
proc ftf_stateoftest { module }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension.

Return Value:

The procedure returns the list from a call to the child's `test_state` procedure. If the child has no `test_state` procedure, an empty list is returned.

Usage Notes:

Use `ftf_stateoftest` in a parent test's own `test_state` procedure to record the state of its children as part of the parent's state. The parent can then restore its own state along with the state of its children by sending the list as a parameter to `test_setup`.

6.2.4.13 ftf_testdir Determining a Test's Load Directory

Specification:

```
proc ftf_testdir { {module ""} }
```

Parameters:

`module` A loaded test module name without a leading path or a file name extension. An empty string refers to the calling test module.

Return Value:

The absolute directory path from which the specified test module was loaded. On an error, an empty string is returned.

Usage Notes:

Use `ftf_testdir` in a test to determine the test's own installation directory. The results allow manipulation of directories relative to the directory specified by `ftf_testdir`.

The Tcl `file join` command combines an absolute path with a relative path.

6.2.5 Variables Management

This section contains procedure and variable specifications for variable definition and management. For additional information about VDT tables and their use, refer to the *Variable Description Tables* section in this chapter.

6.2.5.1 The Variable Description Table

Specification:

```
variable Test_VDT

set Test_VDT(varname) vdtentry
```

Where:

`Test_VDT` The name of a variable description table. You may create more than one VDT, but the one named "`Test_VDT`" is the default table.

`varname` The name of a variable within the test's own namespace.

`vdtentry` A Tcl list containing one to five items, as described below.

A VDT Entry List:

1. Default value when the variable is first created. Required item if you want to create the variable in batch mode.
2. Description. The default is the variable name. The description is used only in interactive mode if no acquisition callback is defined.
3. Data type or selection list: (A) One of the letters `d,u,x,X,o,s,f,e,E,g,` or `G` as with the Tcl `format` and `scan` commands, or one of the following types unique to the framework:
   - `b` Binary value
   - `y` Yes or no
   - `F` File name for reading
   - `W` File name for writing
   - `T` Test module file name for reading
   - `S` Setup file name for reading
   - `I` Interface
   - `M` Transport module
   - `D` Directory
   - `B` Byte list (a series of numbers and quoted strings representing byte values)

   (B) This field can alternatively be a Tcl list, in which case the value of the variable is restricted to those items found in the list.

   The default type/list is "`s`". The type/list field is used only in interactive mode, and only if no acquisition callback is defined.

4. Acquisition information: The name of a callback procedure to acquire the value. The specification is:

   ```
   proc acquire_value { varname currentval }
   ```

   or:

   ```
   proc acquire_value { varname currentval vdtentry }
   ```

   The procedure normally returns the acquired or computed value. In interactive mode, the default is a procedure acquiring a value from the user. In batch mode, the procedure "acquires" the default value. The second form is required only if in order to override the default VDT.

5. Validation information: (A) The name of a callback procedure to validate the value after acquisition. The specification is:

   ```
   proc check_var { varname proposedval }
   ```

   or:

   ```
   proc check_var { varname proposedval vdtentry }
   ```

   The procedure normally returns a list of two items: {ecode approvedvalue}. The first item is an error code that is zero if the value is approved and non-zero otherwise. The second item is the approved value, perhaps after rounding or some other tweaking. The default is no validation, always good and with no tweaks. The second form is required only in order to override the default VDT.

   (B) This field can alternatively be a two-item Tcl list representing the minimum and maximum values for the variable. This option is supported only for numeric data types.

6.2.5.2 ftf_getvar Checking for Test Variables

Specification:

```
proc ftf_getvar { varname {reget 0} {vdt Test_VDT} }
```

Parameters:

`varname`	The name of a variable in the test's own namespace.
`reget`	A flag indicating that the procedure should re-acquire the value even if the variable is already defined.
`vdt`	An alternate variable description table.

Return Value:

The value of the specified variable.

Usage Notes:

Use this procedure to acquire or create any variable not guaranteed to pre-exist, particularly variables from various external sources. Once created, the test may access the variables directly but may not directly modify variables outside its own namespace.

The method for generating the variable is determined by its VDT entry. If the variable does not have a VDT entry and the framework is in interactive mode, the framework prompts the user for a value. In batch mode, if the variable does not exist and it has no VDT entry, the test aborts without an error.

6.2.5.3 ftf_getallvars Checking for All Test Variables

Specification:

```
proc ftf_getallvars { {reget 0} {vdt Test_VDT} {order “”} }
```

Parameters:

`reget` A flag indicating that the procedure should re-acquire the values even if the variables are already defined.

`vdt` An alternate variable description table.

`order` A list of variable names (subscripts of the `vdt` array) indicating the order in which the variables should be processed. This may also be used to limit processing to a subset of the variables defined in the `vdt` array. The default is to process all variables in the `vdt` array in a non-deterministic order.

Return Value:

There is no return value.

Usage Notes:

This convenience procedure invokes `ftf_getvar` for every element in the default or specified variable description table and generates an error if the array does not exist.

Use `ftf_getvar` in a test module’s `test_setup` and/or `test_init` procedure.

6.2.5.4 ftf_getconfig Checking for Configuration Variables

Specification:

```
proc ftf_getconfig { varName {raiseError 0} }
```

Parameters:

`varName` A configuration variable name from the tables in Installation and Configuration chapter of the Intelligent Platform Management Interface (IPMI) Conformance Test Suite (ICTS) User Guide.

`raiseError` If non-zero, the procedure raises an error. If zero, the procedure returns an empty string for non-existent variables. Trap the error with the Tcl `catch` command.

Return Value:

The value of the specified variable. If the variable does not exist, the framework does not create it, and the `raiseError` flag is set. An empty string is returned.

Usage Notes:

This procedure is the approved method for tests to retrieve the value of configuration variables. This procedure gives the framework an opportunity to create the variable if it does not exist. It does not have built-in knowledge of all configuration variables, so it may not know how to create a given variable.

6.2.5.5 ftf_checkarray Checking Array Contents

Specification:

```
proc ftf_checkarray { arrayName idxList {options "e"} }
```

Parameters:

`arrayName` The name of the array to check.

`idxList` A list of index values (subscripts) that are expected to be found in the array.

`options` A combination of any of the following characters:

- `e` Check for expected elements.
- `u` Check for unexpected elements.
- `g` Allow for "generic" array elements. For example, if `idxList` contains "CBC1" an array element indexed by "CBC" is a match.
- `v` Be verbose. Display warning messages for mismatches.

Return Value:

Zero, if the array check passes; otherwise, a non-zero value.

6.2.5.6 ftf_interface Querying for the Default Interface

Specification:

```
proc ftf_interface { }
```

Return Value:

The currently selected default interface, a variable maintained by the framework on the behalf of tests and the user.

Usage Notes:

It is up to tests (and perhaps libraries) to use the value returned by this procedure. The currently selected default interface has no effect on the functionality or behavior of the Base Framework itself.

The default interface is initialized whenever a target configuration file is loaded. Its value comes from the first item in the `Target_Interfaces` list.

6.2.6 SMTP Electronic Mail

The framework allows routing of messages across electronic mail routes. This section contains procedure specifications for capturing messages and routing them through electronic mail.

6.2.6.1 ftf_mail Sending E-mail

Specification:

```
proc ftf_mail { toList ccList from subject body }
```

Parameters:

`toList`	A list of e-mail addresses to which to send a message. If empty, the procedure gets the value from the `User_Email` variable.
`ccList`	A list of e-mail addresses to which to send copies of a message.
`from`	The e-mail address of the sender. If empty, the procedure gets this value from the `User_Email` variable.
`subject`	Subject of the message.
`body`	Body of the message. The body text allows embedded new-line characters.

Return Value:

A non-zero value if an error occurred.

Usage Notes:

Use this procedure to send a message using Simple Mail Transfer Protocol (SMTP) via the mail server specified by the `Host_MailHost` variable. SMTP is defined in RFC 821.

This procedure records the results of the transaction in the mail log file named "`mail.log`" in the directory specified by the `Host_LogDir` variable.

6.2.6.2 ftf_mailing_okay And ftf_paging_okay Permission to Send E-Mail

Specification:

```
proc ftf_mailing_okay { }
```

```
proc ftf_paging_okay { }
```

Return Value:

A non-zero number if automated sending of e-mail or automated paging is currently allowed; otherwise, returns zero.

Usage Notes:

Use these procedures to determine permission for sending e-mail with libraries with built-in automated e-mailing of test results.

The `ftf_mailing_okay` procedure is for the `User_Email` variable and other e-mail addresses to which you would send conventional e-mail messages.

The `ftf_paging_okay` procedure is for the `User_Pager_Email` variable and other e-mail addresses associated with pager devices. For pager devices, the body of the message should be kept very short.

This procedure is not required if the user has direct control over whether a message is sent or not, via a confirmation dialog or other means.

6.2.7 FTP File Transfer

This section contains specifications for procedures that provide access to the File Transfer Protocol (FTP). The procedures in this section require an installed, conventional text-based FTP client program on the host machine in a directory listed in the PATH environment variable.

6.2.7.1 ftf_ftpget Getting a File

Specification:

```
proc ftf_ftpget { host remoteFile localFile {mode "bin"} {user "anonymous"}
{password ""} }
```

Parameters:

`host`	The name or IP address of the remote host.
`remoteFile`	The name of the remote file to get, including a leading path.
`localFile`	The name of the local copy of the remote file, including a leading path. This file will be overwritten if it already exists.
`mode`	Transfer mode: "bin" or "ascii".
`user`	User's login name on the remote host.
`password`	User's password on the remote host. If empty, the value of the `User_Email` variable is used.

Return Value:

A non-zero value if an error occurred.

6.2.7.2 ftf_ftpput Putting a File

Specification:

```
proc ftf_ftpput { host localFile remoteFile {mode "bin"} {user "anonymous"}
{password ""} }
```

Parameters:

`host`	The name or IP address of the remote host.
`localFile`	The name of the local copy to put, including a leading path.
`remoteFile`	The name of the remote file to create, including leading path. This file will be overwritten if it already exists.
`mode`	Transfer mode: "bin" or "ascii".
`user`	User's login name on the remote host.
`password`	User's password on the remote host. If empty, the value of the `User_Email` variable is used.

Return Value:

A non-zero value if an error occurred.

6.2.7.3 ftf_ftpls Getting a Directory Listing

Specification:

```
proc ftf_ftpls { host remoteDir {user "anonymous"} {password ""} }
```

Parameters:

`host`	The name or IP address of the remote host.
`remoteDir`	The name of the remote directory from which a listing is desired.
`user`	User's login name on the remote host.
`password`	User's password on the remote host. If empty, the value of the `User_Email` variable is used.

Return Value:

A list of file names from the remote directory. If there is an error, an empty list is returned.

6.2.7.4 ftf_ftpdelete Deleting a File

Specification:

```
proc ftf_ftpdelete { host remoteFile {user "anonymous"} {password ""} }
```

Parameters:

`host`	The name or IP address of the remote host.
`remoteFile`	The name of the remote file to delete.
`user`	User's login name on the remote host.
`password`	User's password on the remote host. If empty, the value of the `User_Email` variable is used.

Return Value:

A non-zero value if an error occurred.

6.2.8 Cursor-Addressable Display Areas

This section contains procedure specifications for manipulation and access to cursor-addressable display (CAD) areas. These procedures allow creation of accessible user interface objects and manipulation of those objects in response to test results and procedures.

6.2.8.1 ui_opencad Open a Cursor-Addressable Display

Specification:

```
proc ui_opencad { {title "FTF-CAD"} {rows 25} {cols 80} }
```

Parameters:

`title`	Window title
`rows`	Number of rows
`cols`	Number of columns

Return Value:

A handle that may be used for subsequent operations. If there is an error, an empty string is returned.

Usage Notes:

Test modules should check the return value of an empty string and attempt to proceed with the test even if the open request fails. This is because if a text-based framework is developed, this facility may not exist or may exist only in limited form.

Data written to these areas is never logged to a file.

6.2.8.2 ui_closedcad Close a Cursor-Addressable Display

Specification:

```
proc ui_closecad { hcad }
proc ui_freecad { hcad }
```

Parameters:

`hcad`	Handle to a cursor-addressable display area returned from `ui_opencad`

Return Value:

Zero on success; otherwise, non-zero.

Usage Notes:

- The `ui_closecad` procedure closes the display area immediately.
- The `ui_freecad` procedure queues the display for closing at a later time by the user or when `ui_opencad` opens new display areas. Queuing gives a test module a means to indicate to the framework that it has finished writing to the display area, while letting the user view the final contents of the area before it is closed.
- Even if a test does not close or free a display area before stopping, the test should not count on the area still being available if the test is started again.

6.2.8.3 ui_writecad Write to a Cursor-Addressable Display

Specification:

```
proc ui_writecad { hcad row col text {options ""} }
```

Parameters:

`hcad`	Handle to a cursor-addressable display area returned from `ui_opencad`
`row`	Row number at which to begin writing text. Rows are numbered from top to bottom, starting with zero.
`col`	Column number at which to begin writing text. Columns are numbered from left to right, starting with zero.
`text`	Text string to write. It may contain embedded new-line characters, but no other escape characters.
`option`	A Tcl list containing items of the following form:
	`tag=attribute` Text display attribute (red, green, blue, reverse, normal, underline)

NOTE

Depending on the display device, the framework reserves the right to ignore all text attribute options.

Return Value:

Zero if the entire text string fits in the display area. If not, or in the event of any other error, a non-zero value is returned.

6.2.8.4 ui_cleared Clear a Cursor-Addressable Display

Specification:

```
proc ui_clearcad { hcad }
```

Parameters:

`hcad` Handle to a cursor-addressable display area returned from `ui_opencad`

Return Value:

Zero on success, non-zero in case of an error.

6.2.9 System Information

This section contains procedure specifications for accessing host and target system information.

6.2.9.1 ftf_version Version Numbers

Specification:

```
proc ftf_version { }
```

Return Value:

A list suitable for conversion into an array with the Tcl `array set` command. The resulting array has the elements listed in Table 6-3:

Table 6-3. Array Set Command Array Contents

Element	Description	Examples
FTF	FTF version number	0.94
FTF,major	FTF major version number	0
FTF,minor	FTF minor version number	94
FTF,edition	FTF edition	ESG, ICTS
Tcl	Tcl version number	8.3.0 through 8.3.9
Tcl,major	Tcl major version number	8
Tcl,minor	Tcl minor version number	0
Tcl,patch	Tcl patch level	5
Tcl,required	Minimum Tcl version required for FTF	8.0.4
Tcl,recommended	Minimum Tcl version recommended for FTF	8.3.0 through 8.3.9

In addition to the array elements listed in the above table, the entire contents of the pre-defined Tcl global array `tcl_platform` are copied to the list returned by this procedure. For example, the array element `tcl_platform(os)` becomes the `Platform,os` element of the list.

6.2.9.2 ftf_cps Clicks-per-Second

Specification:

```
proc ftf_cps { }
```

Return Value:

A calibrated value for clicks-per-second, where “clicks” are those used by the Tcl `clock clicks` command.

Usage Notes:

Clicks-per-second is calibrated immediately after the host configuration file is loaded.

6.2.10 API Provided by Libraries

This section contains API procedure specifications for sending requests, receiving responses, and formatting responses. Test modules and Firmware Test Library modules use these functions.

6.2.10.1 fmt_req Formatting Request Packet

Specification:

```
proc fmt_req { cmd_name d0 {d1 ""} {d2 ""} {d3 ""}… {d10 ""} }
```

Parameters:

`cmd_name` Name of the firmware function if `cmd_name` is a valid firmware command. If `cmd_name` is valid, it uses the predefined response structure.

`d0 d1 …d10` The data format and number of parameters depend on the request data structure. The first element in the request data structure corresponds to the first parameter; the second element corresponds to the second parameter; the third to the third, and so on.

Return Value:

`reqlist` A list of bytes on success. A null list on failure.

Usage Notes:

Use this function to construct request data for the `req_rsp` function.

The `fmt_req` function requires the developer to know the organization of the request data structure. Mistaken assumptions about the organization of the request structure cause unpredictable results.

If the `RecData` element appears in the request structure, it represents a list.

Example:

This example uses the following request data structure:

```
set reqSelGetEntry {
      { ReserveId   2-1 }
      { RecId    4-3 }
      { Offset      5 }
      { ReadCount   6 }
}
```

The following example uses a `reqlist` call to generate a byte list. The generated list in the `req_rsp` call is used in a `SelGetEntry` call with `ReserverId = 0x1020, RecId = 0x0005, Offset = 0` and `ReadCount = 0xFF,` where `ReserveId` is a two-byte field, `RecId` is a two byte field, `Offset` is a one byte field, and `ReadCount` is one-byte field.

The following call constructs the request data.

```
set reqlist [fmt_req SelGetEntry 0x1020 5 0 0xFF]
```

The reqlist contains: 0x20 0x10 0x05 0x00 00 0xFF.

The req_rsp function call sends the request and gets the response.

```
array set selentry [req_rsp $reqlist ]
```

6.2.10.2 req_rsp Synchronous Send and Get

Specification:

```
proc req_rsp { cmd_name {reqdata ""} {dlrt “”} {options "CR"} }
```

Parameters:

`cmd_name` Name of the firmware functions for which the send request and receive response are made.

`reqdata` Optional. A list of request bytes.

`dlrt` Optional. A destination list containing LUN, route, and timeout information. For additional information about `dlrt,` refer to Chapter 5.

`Options` Any combination of the following letters:

- C Check the response for validity including min/max length checking.
- c Don’t check the response for validity. Note that when this option is used the returned data may not be suitable for use with the `print_rsp` procedure (section 6.2.10.4). A better choice would be `print_arr` (section 4.6.23).
- R Send the request and wait for the response. (Also “y” for backward compatibility.)
- r Send the request but don’t get the response. (Also “n” for backward compatibility.)

Return Value:

`rspdata` A list for constructing the response data array by executing the `array_set` command. For additional information about `array_set`, refer to section `array_set` Creating a New Array section of this manual. Response data array Elements are defined in Table 6-4.

Usage Notes:

For a debug level of three, request and response data bytes print during this call

For a debug level of two, the data array prints during this call

Change debug level behavior with the `req_rsp_msg_control` procedure. For additional information on `req_rsp_msg_control`, refer to Section 6.2.10.5 of this chapter. (Note: The tests modules included with ICTS utilize req_rsp_msg_control to transfer the default debug level behavior to the verbose level.)

Table 6-4. Response Data Array Elements

Array Index	Descriptions
`Cmdname`	The firmware command name (`cmd_name`) for the response data.
`Merr`	The error code (`ecode`) returned by the message library.
`Mreq`	The request data sent to the message library.
`Mrsp`	The response data bytes (`data`) returned by the message library.
`Emsg`	An error message. If the completion code is not zero, it contains the completion code error message. If `merr` is not zero, it contains the error message returned by `merror`. If the response packet length does not match, it contains an appropriate error message. Contains an empty string for zero completion code.
`Dlrt`	Destination, route, LUN, and response information used when the request was made. Chapter 5 describes dlrt in detail.
`Resptime`	The number of milliseconds taken between request send and response received.
`Imode`	The mode of the target firmware according to the test library before sending the request message merged with the interface.
`CompCode`	The Completion Code returned by the response data in the LSB. If the message library returns an error code, if the response data is less than `minbytes` or more than `maxbytes`, or if any other internal error occurs, MSB contains a non-zero value. It is sufficient for the tests to look at `CompCode` before using the response data.
`RecData`	Optional field containing a list of data bytes from `mres`. Begins at an offset specified in the response data structure and ends at the end of the `mres` list. The `RecData` element is present only when the response data structure contains a `RecData` element.
`Element_name…`	The field names are the same as defined in the response data structure except `minbytes` and `maxbytes` elements. Field value depends on the representation of elements in the response data structure.

Example:

The following example call returns the response data array to the `selinfo` variable from `SelGetInfo` firmware function.

```
array_set selinfo [req_rsp SelGetInfo]
```

The response data structure defined in the library for `SelGetInfo` is:

```
set resSelGetInfo {
      { SelVersion 1 }
      { LogCount      3-2 }
      { FreeSpace  5-4 }
      { AddTime    9-6 }
      { EraseTime  13-10 }
      { OpSupport  14 }
      { OverflowFlag  14-7:7 }
      { DelSelSup  14-3:3 }
      { PartAddSup 14-2:2 }
      { ReserveSelSup 14-1:1 }
      { SelAllocSup   14-0:0 }
      { minbytes      15 }
      { maxbytes      15 }
}
```

Response array contents on a particular are:

```
name:       SelGetInfo
merr:       0
mreq:
mres:       00 10 02 00 C0 0C 62 F9-5A 36 45 69 5B 36 02
dlrt:       BMC 0 I2C/FWH-I2C/COM1
imode:      OP_I2C
restime:    43
CompCode    = 0
SelVersion = 10
LogCount    = 02
FreeSpace   = 0cc0
AddTime     = 365af962  11/24/98 10:22:26
EraseTime   = 365b6945  11/24/98 18:19:49
OpSupport   = 02
 OverflowFlag  = 00
```

```
DelSelSup     = 00
PartAddSup    = 00
ReserveSelSup = 01
SelAllocSup       = 00
```

Where:

`SelVersion, LogCount, FreeSpace, AddTime, EraseTime, OpSupport, OverflowFlag, DelSelSup, PartAddSup, ReserveSelSup` and `SelAllocSup` are elements of "`Gel Sel Info`" command response data structure.

6.2.10.3 tlm_CMDex with tlm_CMD to call commands

Specification:

```
proc tlm_CMDex {cmdname {args ""} {argList ""} tlm_CMD {} }
proc tlm_CMD {cmdname {args ""} {route ""} {micro ""} {expectedCompCode "0"}
{errorCompCode "1"} {abortReturnCode "1"} {formatData "y"} {stopCheck "y"}
{silent "n"} {msendget "y"} }
```

Parameters:

`cmdname`		A firmware command name from the command library.
`args`		The request data for the firmware command.
`route`		A route to send command through (e.g. SMS, I2C, ICMB, ...).
`micro`		A device to send commad to (e.g. BMC, HSC, ...).
`expectedCompCodergs`		Any one of the following numbers:
	0	If the above firmware command does not return one of the expectedCompCodes, the function will return: 'return $errorCompCode'.
	-1	Specifies that the function will not fault regardless of the commands return code.
	-2	Specifies that the function should fail on an error does not occur.
	?	A single number specifies for the routine to return a fault if the specific code is not returned.
	[list ? ? ?]	A list of numbers indicates for the routine to return a fault only if the command's return code doesn't match one of the specified values
`errorCompCode`	"1"	Specifies the return code when the executing firmware command does not return one of the expected completion codes.
`AbortCompCode`	"1"	Specifies the return code when a user has stopped the test.

FormatData	"y"	Specifies if the 'args' passed should be formatted with the 'fmt_req' cmd.
StopCheck	"y"	Obviously, this command returns a 'return abortReturnCode' to the caller so that if the user requests stop, the script can stop. Alternatively, one can request this routine to ignore user requests to stop.
Silent	"n"	This switch is used to disable this routine's print.
Msendget	"y"	y = use "msendget", anything else (like "n") uses "msend" (no get).

Return Value:

The return data is either going to be of the form 'return x' where x is a number or someother value, or it will be of the form 'array_set rsp [array get returnData]' where return data is the data being returned from the firmware command. The success, will return essentially equates to 'array_set rsp' to the functions return data. If the command failes, the function returns a 'return 1' so that the current procedure is aborted. This behavior is switchable so the function can continue.

Usage Notes:

```
test_main { } {
    set_packet_controls SMS "" BMC
     #
    # Note, if any of the commands below error, test_main will be exited.
    #
    eval [tlm_CMD SetSenThreshold "0x41 0"]; # using the current micro and interface
    ftf_msg "thresholds are: [array get rsp];
    eval [tlm_CMD SetSenThreshold "0x41 0" "" "" 0 1 1 y y y]; # same as above, except the call is
  silent
    ftf_msg "thresholds are: [array get rsp];
    set silent {"" "" 0 1 1 y y y}
    eval [tlm_CMDex SetSenThreshold "0x41 0" $silent]; # same as above, but demonstarates the
ability to easily define a default behavior without constantly retyping a long list of arguments.
   ftf_msg "thresholds are: [array get rsp];
  }
#   Note, in the above example, silent defines all of the variables for tlm_CMD.  Be aware that it is
only necessary to define the variables up to and including the ones that are of concern.
```

Avdanced Usage Notes:

Since in testing many firmware calls are made with similar characteristics (i.e. options to the tlm_CMDex function), it is convenient to tag these parameters with a variable to make the code

more readable. Within FTF a large number of typical uses have been identified and defined as procedures that return the parameter to pass the tlm_CMDex function. These functions are located in the file util_CMD.tcl and should be referenced directly in the code. An example use is:

eval [tlm_CMD SetSenThreshold "0x41 0" [tlm_DON'T_RETURN_ON_FAILURE]];

Alternatively, this could be called with an optional route or destination micro as with:

eval [tlm_CMD SetSenThreshold "0x41 0" [tlm_DON'T_RETURN_ON_FAILURE "i2c"]];

eval [tlm_CMD SetSenThreshold "0x41 0" [tlm_DON'T_RETURN_ON_FAILURE "sms"]];

For reference, this particular function is coded as:

```
proc tlm_DONT_RETURN_ON_FAILURE
        { {route ""} {micro ""} } { return "\"$route\" \"$micro\" -1 1 1 y y n" }
```

which you can see simply returns a string of parameters for the function tlm_CMDex.

6.2.10.4 print_rsp Displaying Response Data Array

Specification:

```
proc print_rsp { rsparr {mode “”} }
```

Parameters:

`rsparr` Name of the response data array that need to be printed.

`mode` A string "all" prints all the elements in the response array. By default it calls the print function of firmware command.

Return Value:

None

Usage Notes:

Use to display response data array elements or verbose response data.

Example1:

The following output shows an example of `SelGetInfo` response data.

```
array_set selinfo [req_rsp SelGetInfo]
print_rsp selinfo
```

The completion code is zero:

```
======Get SEL Info======
SelVersion = 10
LogCount   = ce
FreeSpace  = 00
AddTime    = 365b7741  11/24/98 19:19:29
EraseTime  = 365b6945  11/24/98 18:19:49
Operation Support = 82
```

```
Overflow Flag          = 01
Delete SEL Support     = 00
Partial Add Support    = 00
Reserve SEL Support    = 01
SEL Alloc Info Support = 00
```

Example2:

```
array_set selinfo [req_rsp SelGetAllocInfo]
print_rsp selinfo
```

The `SelGetAllocInfo` completion code is non-zero:

```
======Get SEL Allocation Info======
CompCode = c1, Invalid Command
```

6.2.10.5 req_rsp_msg_control Send/Get Debug/Verbose Control

Specification:

```
proc req_rsp_msg_control { rr mode fmt {newlevel ""} }
```

Parameters:

`rr` A string, "`req`" (request) or "`res`" (reponse), indicating the type of message to which this change applies.

`mode` A letter, "`d`" (debug) or "`v`" (verbose), indicating the type of message level to which this change applies.

`fmt` A letter, "`h`" (hex) or "`a`" (ASCII formatted, as with `print_rsp` in Section 6.2.10.4), indicating the type of formatting used for output.

`newlevel` A numeric value. Indicates the lowest debug or verbose level at which `req_rsp` automatically generates printed hex or ASCII output for the request or response message. An empty string causes default behavior restoration. Setting this value to a large number (e.g., 99) disables automatic printing by `req_rsp`.

Return Value:

`oldlevel` The previous level for the specified combination of `rr`, `mode`, and `fmt`. An empty string represents the system default.

Usage Notes:

The combination of `rr=req` and `fmt=a` is not supported and has no effect. All other combinations of `rr` and `fmt` are supported.

Settings made by this procedure apply only to the calling test module.

6.2.11 Destination and Route Controls

This section contains procedure specifications for providing destination addresses, LUN numbers, and route information for request messages and receive responses for the `req_rsp` function. The following API provides getting the information to the test modules.

6.2.11.1 set_packet_controls Setting Message Packet Controls

Specification:

```
proc set_packet_controls { dest lun route {timeout ""} }
```

Parameters:

`dest` A value or string. Denotes the destination address. If a string, the address name should be defined in the platform configuration file. An empty string uses the default destination. Initial default is BMC.

`lun` A value or a string. Denotes the destination LUN number. An empty string returns the default LUN number. Initial default is LUN 0.

`route` A string. Denotes the interface or transport route in which a message is sent or received. An empty string denotes the default route. Initial default route is the first interface defined in the target interface list.

`timeout` A value. Identifies the `route` timeout value for getting a message.

Return Value:

`dlrt` A list. Contains the destination address, LUN number, route and timeout values for a `req_rsp` call.

Usage Notes:

Use to set the default control information for message packets.

6.2.11.2 get_packet_controls Getting Message Packet Controls

Specification:

```
proc get_packet_controls { {dest ""} {lun ""} {route ""} {timeout ""} }
```

Parameters:

`dest` A value or a string. A value denotes the destination address. If it is string, the address name should be defined in the platform configuration file. An empty string uses the default destination. Initial default is BMC.

`lun` A value or a string. A value denotes the destination LUN number. An empty string returns the default LUN number. Initial default is LUN 0.

`route` A value or a string. A value denotes the interface or transport route in which a message is sent or received. An empty string denotes the default route. Initial default route is the first interface defined in the target interface list.

`timeout` A value. Identifies the `route` timeout for getting a message.

Return Value:

`dlrt` A list. Contains the destination address, LUN number, route and timeout values for a `req_rsp` call.

6.2.11.3 fmt_data Formatting Data Based Using List Structure

Specification:

```
proc fmt_data { structure_list data_list }
```

Parameters:

`structure_list` A list. Contains structure elements similar to those used in the `fmt_req` function.

`data_list` A list. The bytes requiring formatting.

Return Value:

`result_list` A list. Contains a structure list element name and formatted value from the data list.

Usage Note:

Use the `array_set` command to create an array similar to the one described in the function specification for `req_rsp`. The elements in this array can be indexed by structure list elements.

If an element named `RecData` is present in the structure list, it is treated as a variable length list of bytes starting at the position represented by the `RecData` element.

The structure element names `minbytes` and `maxbytes` are treated as normal structure elements with no special meaning, as in the `req_rsp` call.

6.2.11.4 get_fwcmd_info Firmware Command Information

Specification:

```
proc get_fwcmd_info { cmd_name field_name {microname ""} }
```

Parameters:

`cmd_name` A string. The name of the firmware command for which information is needed.

`field_name` A string. The name identifies a field item of the firmware command. Valid field names are: `CmdNumber, NetFnNumber, Description, ReqList, RspList, OptRspList` and `fw_modes`.

`microname` A string. The name of the micro to which the `cmd_name` belongs. A null string uses BMC as the default micro.

Return Value:

On success, the return type depends on the field name. On failure, the return value is an empty string. Table 6-5 lists possible success return value types.

Table 6-5. Firmware Command Information

field_name	Return Value
CmdNumber	Returns a value. The value represents the command number of a given firmware function `cmd_name`. Example: `get_fwcmd_info SdrGetInfo CmdNumber` The function returns a value `0x20`.
NetFnNumber	Returns a value. The value identifies the net function of the firmware command `cmd_name`. Example: `get_fwcmd_info SdrGetInfo NetFnNumber` The function returns a value `0x0A`.
Description	Returns a string. The string describes the command. Example: `get_fwcmd_info SdrGetInfo Description` The function returns `"Get SDR Repository Info"`
ReqList	Returns a list with a request data structure. An empty list is returned if no request data structure is defined for the `cmd_name.`
RspList	Returns a list with a response data structure. An empty list is returned if no response data structure is defined for the `cmd_name.`
OptRspList	Returns a list with an optional response data structure. An empty list is returned if no response data structure is defined for `cmd_name.`
fw_modes	Returns a value 1 if the `cmd_name` is supported in the firmware mode and the interface and power state are specified by the `fw_modes` parameter. Otherwise, returns an empty string. The function `get_fwmodes` returns a list of available `fw_modes.`

6.2.11.5 get_fwmodes Available List of Firmware Modes

Specification:

```
proc get_fwmodes { power_state {cmdname “”} {microname “”} }
```

Parameters:

`power_state` A string. Identifies the power state for which a list of firmware modes are needed. The string `“PowerOn”` identifies the power ON state. The string `“PowerStandby”` identifies the standby power state.

`cmdname` A string. A command name that causes the return of all modes associated with the power state for the specified command.

`microname` A string. The name of the micro to which the `cmd_name` belongs. The null string uses BMC as the default name.

Return Value:

`fw_modes` A list. Contains a firmware mode, an interface identifier, and a power state identifier. The identifier list format is:

`[power_]fwmode_[RM_]iface.`

Where:

`power` A string. `STB` identifies standby power state; otherwise, `power` is empty.

`fwmode` A string. Identifies the firmware mode. "`OP`" represents normal Operational mode, and "`MTM`" represents Manufacturing Test Mode.

`iface` A string. Identifies the interface. Valid interface strings are `"SMS"` and `"I2C".`

Usage Notes:

Querying a particular mode is supported for firmware commands using the `get_fwcmd_info` function.

6.2.11.6 get_address Bus Address

Specification:

```
proc get_address { micro_name }
```

Parameters:

`bus_name` A string. A name identifying the microcontroller used. A list of microcontrollers names is specified in the platform configuration file. For example, `BMC` and `CBC` are microcontroller names.

Return Value:

`address` A value. Contains the microcontroller address. An empty string if the given microcontroller name is not found.

Usage Notes:

A test can query the availability of a particular microcontroller for a specified platform before constructing the destination address for `dlr` or before executing a particular firmware function.

Example:

The function call `[get_address BMC]` returns address `0x20`.

6.2.11.7 get_fwcmd_name Get Firmware Function Name

Specification:

```
proc get_fwcmd_name { netfn_number cmd_number {micrname “”} }
```

Parameters:

`netfn_number` A number. Represents the net function.

`cmd_number` A number. Represents the firmware command.

`microname` A string. The name of the micro to which the `cmd_name` belongs. The null string uses BMC as the default.

Return Value:

`cmd_name` A string. On success, the name of the firmware function used by library functions such as `fmt_req, req_rsp` and `get_fwcmd_info.` Returns an empty string if no name is associated with the given net function and command number.

Usage Notes:

A test can find the name associated with a command before making requests.

6.2.11.8 get_netfn_number Net Function Number

Specification:

```
proc get_netfn_number { netfn_name {microname “”} }
```

Parameters:

`netfn_name` A string. Name of a net function such as `Application, Storage, Sensor, Bridge` and `Chassis`.

`microname` A string. Name of the micro to which the `cmd_name` belongs. The null string uses BMC as default.

Return Value:

`netfn_number` A number. Returns a net function number on success; otherwise, an empty string.

6.2.11.9 set_fw_mode Firmware Mode

Specification:

```
proc set_fw_mode { mode {value “”} }
```

Parameters:

`mode` Controls the type of input value. The mode can be any one of the following switch fields:

`-pm` Switch `–pm` specifies the mode to operate on power control field. Valid power modes are ON and STB.

`-fm` Switch `–fm` specifies the mode to operate on firmware mode. Valid firmware modes are OP and MTM.

`value` A string. The mode switch determines the string type. An empty string returns current mode value. A valid value sets the global firmware mode and returns the previous mode.

Return Value:

`state` A string. Return value depends on the mode switch.

Usage Notes:

The target system mode determines particular firmware command support information.

Target power state: When the target is OFF (AC OFF), the target is not running firmware, and no firmware calls are supported. When the target is in standby, firmware calls for unsupported sensors are not supported.

Firmware Mode: The target firmware modes are Operational Mode and Manufacturing Test Mode. Firmware command availability varies depending on the firmware mode.

The test software needs the current state of the target firmware and system to determine pass and failure for each firmware call. The state can be managed by monitoring the different request and response messages or by user control.

If a test transitions from one mode to other, it calls this function to set the current mode and allow the library to know which command is executed in which mode.

6.2.11.10 get_cmd_exec_status Command Execution Status

Specification:

```
proc get_cmd_exec_status { cmdname interface {fwmode "OP"}\ {powerstate "ON"}
{microname "BMC"}}
```

Parameters:

`cmdname` Command name for getting execution status

`interface` Interface name, such as SMS or I2C

`fwmode` Firmware mode, such as OP or MTM

`powerstate` Power state indicator: ON or STB

`microname` Name of the micro corresponding to the `cmdname`

Return Value:

`status` A value. Zero indicates the command is not executed in this framework run. Non-zero indicates the command is executed.

6.2.11.11 auto_interface_wakeup Auto Interface Wakeup

Specification:

```
proc auto_interface_wakeup { iface {command “”} {interval 60} }
```

Parameters:

`iface`	Name of the interface.
`command`	A command to execute when the timer interval expires for the specified interface. This command must be able to accept a “dlrt” value as returned by `get_packet_controls` (section 6.2.11.2) and return a list of the same form returned by `req_rsp` (section 6.2.10.2). An empty string may be used to query the current setting.
`interval`	A timer interval in seconds. A value of zero may be used to disable the wakeup feature for the specified interface.

Return Value:

`Oldconfig`	Returns a list containing two items, the previous command and the previous interval for the specified interface.

Usage Notes:

- Some interfaces require periodic submission of commands to keep an active connection. This function enables or disables this automatic feature.

6.2.11.12 get_target_version Set Target Version Number

Specification:

```
proc get_target_version { version_id {component “”} {microname “”} }
```

Parameters:

`version_id`	A keyword. The type of version number desired. Possible values include: `IPMI` IPMI version number `Firmware` Firmware version number
`component`	A string. A keyword that identifies the desired component of the version number. An empty string requests the entire version number. The string “`Major`” requests the major component and the string “`Minor`” requests the minor component.
`microname`	A string. The microcontroller for which version number information is desired. Not applicable to all values of `version_id`. When applicable, an empty string results in an assumption of BMC.

Return Value:

`version`	A value. The requested version information.

Index

G

H

I

L

M

N

P

R

S

T

U

V

W