Arm® System Control and Management Interface Test Suite

Version 2.0

Validation Methodology

Arm® System Control and Management Interface Test Suite

Validation Methodology

Release Information

Document History

Issue	Date	Confidentiality	Change
0200-01	30 September 2019	Non-Confidential	New document for v2.0 alpha
0200-02	16 December 2020	Non-Confidential	REL 2.0

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

This document includes terms that can be offensive. We will replace these terms in a future issue of this document.

If you find offensive terms in this document, please contact *terms@arm.com*.

Contents

Arm® System Control and Management Interface Test Suite Validation Methodology

Preface

This preface introduces the *Arm® System Control and Management Interface Test Suite Validation Methodology*.

It contains the following:

- *About this book* on page 6.
- *Feedback* on page 8.

About this book

This book describes the framework and methodology used to run the tests in the Arm System Control and Management Interface (SCMI) test suite.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter introduces the features and components of the Arm System Control and Management Interface (SCMI) test suite.

Chapter 2 Validation Methodology
This chapter describes the validation methodology that is used for the test suite.

Appendix A Revisions
This appendix describes the technical changes between released issues of this book.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Other information

- *Arm® Developer*.
- *Arm® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *support-scmi-acs@arm.com*. Give:

- The title *Arm System Control and Management Interface Test Suite Validation Methodology*.
- The number 101871_0200_02_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the features and components of the Arm System Control and Management Interface (SCMI) test suite.

It contains the following sections:

1.1 Abbreviations

This section lists the abbreviations that are used in this document.

Table 1-1 Abbreviations and expansions

Abbreviation	Expansion
ACS	Architecture Compliance Suite
OSPM	Operating System-directed configuration and Power Management
PAL	Platform Abstraction Layer
SCMI	System Control and Management Interface
SCP	System Control Processor
VAL	Validation Abstraction Layer

1.2 System Control and Management Interface

System Control and Management Interface (SCMI) is a set of operating system-independent software interfaces that are used in system management. It is extensible and provides interfaces for:

- Discovery and self-description of the interfaces it supports.
- Power domain management, which is the ability to place a given device or domain into various supported power states.
- Performance management, which is the ability to control the performance of a domain.
- Clock management, which is the ability to set and inquire rates on platform-managed clocks.
- Sensor management, which is the ability to read sensor data, and be notified of sensor value changes.
- Reset domain management, which is the ability to place a given device or domain into various reset states.

For more information about SCMI, see the *SCMI specification*.

The Architecture Compliance Suite (ACS) is a set of examples of the specified invariant behaviors. Use this suite to verify that these behaviors are implemented correctly in a given platform.

1.3 Test suite components

The compliance suite contains self-checking and portable C-based tests. These tests are divided into various categories based on the protocols supported by the SCMI.

The following table describes the test suite components.

Table 1-2 SCMI test components

Components	Description
Base	Tests to verify base protocol compliance
Clock	Tests to verify clock protocol compliance
Performance	Tests to verify performance protocol compliance
Power domain	Tests to verify power domain protocol compliance
Reset domain	Tests to verify reset domain protocol compliance
Sensor	Tests to verify sensor protocol compliance
System power	Tests to verify system power protocol compliance

1.4 Layered software stack

The compliance tests use the layered software stack approach to enable porting across different test platforms.

The constituents of the layered stack are:
- SCMI tests
- Validation Abstraction Layer (VAL)
- Platform Abstraction Layer (PAL)

The following figure shows the constituents of the layered software stack.

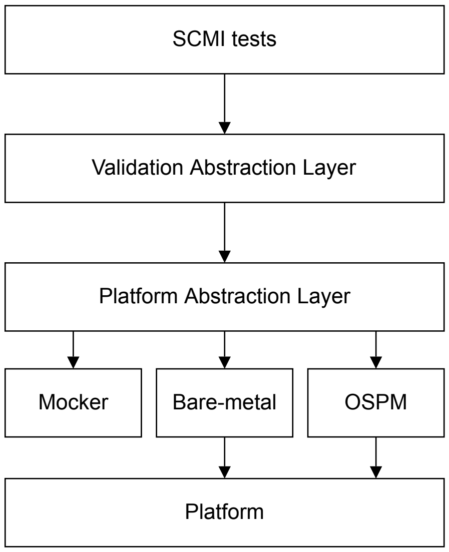

Figure 1-1 Compliance test layers

The following table describes the different layers of a compliance test.

Table 1-3 Compliance test layers

Layer	Description
SCMI tests	Is a collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL.
VAL	Provides a uniform view of all the underlying hardware and test infrastructure to the test suite.
PAL	Is a C-based, Arm-defined API that you can implement. It abstracts features whose implementation varies from one target system to another. Each test platform requires a PAL implementation of its own. PAL APIs are meant for the compliance test to reach or use other abstractions in the test platform such as OS infrastructure and bare-metal abstraction.
Mocker	Provides unit test framework for test flow verification.
Bare-metal	Provides the environment to run the tests as part of SCP firmware.
OSPM	Provides enviroment to run the tests as linux application.

1.5 Deployment scenarios

The SCP firmware can be deployed in two ways:

- As a library in the trusted OS or EL3 firmware running in the Secure world
- On a separate microcontroller

The SCMI ACS is built as an Operating System-directed configuration and Power Management (OSPM) application running in the Normal world.

Scenario 1

The compliance tests run as an OSPM agent using SMC-based mailbox as the transport mechanism when the SCP firmware is running as a library in Trusted OS or in EL3 (TF-A). The following figure shows the SCP firmware running as a library in Trusted OS or EL3 firmware.

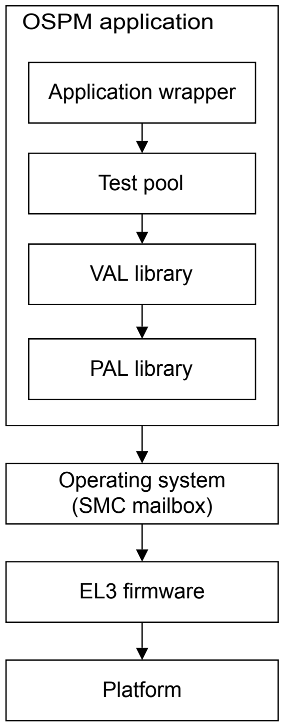

Figure 1-2 SCP firmware running as a library in Trusted OS or EL3 firmware

Scenario 2

The compliance tests run as an OSPM agent using hardware-based mailbox as the transport mechanism when the SCP firmware is running on a microcontroller. The following figure shows the SCP firmware running on a microcontroller.

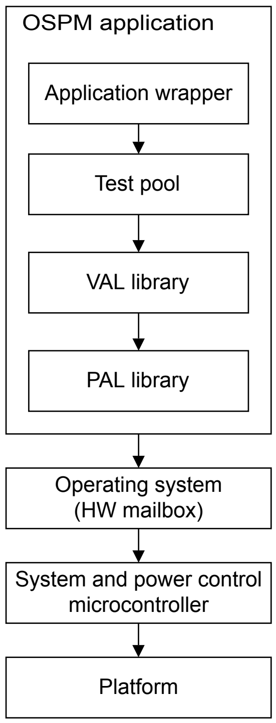

Figure 1-3 SCP firmware running on a microcontroller

1.6 Test suite directory structure

The test components must be in a specific hierarchy for the test suite. When the release package is downloaded from GitHub, the top-level directory contains the components shown in the following figure.

```
scmi/
  |
  |-------Makefile
  |-------README.md
  |-------docs/
  |-------linux_app/
  |-------mocker_app
  |-------baremetal_app
  |-------platform/
  |-------test_pool/
  |                |
  |                |-------base/
  |                |-------clock/
  |                |-------performance/
  |                |-------power_domain/
  |                |-------reset/
  |                |-------sensor/
  |                |-------system_power/
  |
  |-------val/
```

Figure 1-4 Test suite directory structure

The following table describes all the components.

Table 1-4 SCMI ACS directory components and descriptions

Component	Description
`README.md`	Contains the release details of the SCMI test suite.
`docs/`	Contains the suite documentation.
`linux_app/`	Contains wrapper application code to execute the tests on Linux-based platforms.
`platform/`	Contains code for the supported platforms. For example, the mocker platform code for unit testing on the host machine.
`test_pool/`	Contains the test source files for each protocol.
`val/`	Contains common code that is used by the tests. Makes calls to PAL as needed.
`mocker_app/`	Contains application code to execute the tests as host application.
`baremetal_app/`	Contains wrapper code to execute tests for baremetal scenario.

Chapter 2
Validation Methodology

This chapter describes the validation methodology that is used for the test suite.

It contains the following sections:

2.1 Test platform abstraction

The compliance suite defines and uses the test platform abstraction that is illustrated in the following figure.

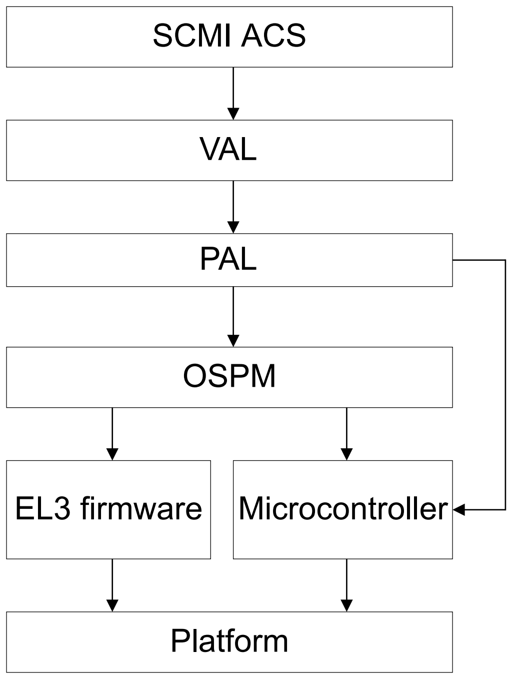

Figure 2-1 Test platform abstraction

The following table describes the SCMI abstraction terms.

Table 2-1 Abstraction terms and their description

Abstraction	Description
EL3 firmware	SCP running as library in TF.
Microcontroller	SCP running on Cortex-M.
Operating system	Operating system providing transport resources.

2.2 Overview of test suite layers

This section describes the test suite layers: SCMI ACS, VAL, and PAL.

SCMI ACS

The test suite contains a set of tests for every supported protocol. These tests are grouped based on protocol and are independent of other protocols. Tests that depend on multiple protocols are present in the integration test directory. For every protocol, the self-discovery tests are run first. The protocol and domain attributes are saved in the VAL layer and used in the execution of subsequent tests.

Every protocol has a test list array which contains the test entry function for that protocol. The test list contains tests for both SCMI version 1.0 and 2.0. The build flag must be passed to build the tests for a specific version. By default, version 2.0 tests are built. The build option can be used to select building the tests for a specific protocol.

VAL

The VAL layer is a generic framework component which prepares an execution context and executes the test suites. This component has all the generic test execution logic that is used by the rest of the test suite components. It provides functions to access platform resources, test dispatcher functions, and database for every protocol to maintain protocol and domain attributes.

PAL

The PAL is a C-based, arm-defined API that must be implemented for different platforms. This has the platform-specific source code which implements the defined interfaces that are needed by the test suite. You can specify the expected values and the agent characteristics here.

The following table lists the common set of PAL files and APIs that must be ported for communicating with the platform.

Table 2-2 PAL APIs and descriptions

File name	API name	Description
`pal_base_expected.h` `pal_base.c`	-	Contains BASE protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_clock_expected.h` `pal_clock.c`	-	Contains CLOCK protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_performance_expected.h` `pal_performance.c`	-	Contains PERFORMANCE protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.

Table 2-2 PAL APIs and descriptions (continued)

File name	API name	Description
`pal_power_domain_expected.h` `pal_power_domain.c`	-	Contains POWER_DOMAIN protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_reset_expected.h` `pal_rest_domain.c`	-	Contains RESET protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_sensor_expected.h` `pal_sensor_domain.c`	-	Contains SENSOR protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_system_power_expected.h` `pal_system_power_domain.c`	-	Contains SYSTEM_POWER protocol which is related to platform-specific information that is needed during the test execution for validating the response returned by SCMI commands. This information must be provided by a given platform.
`pal_platform.c`	`pal_send_message`	Test agent uses this API to send an SCMI command to the platform and receive the response.
	`pal_receive_delayed_response`	Test agent uses this API to receive delayed response.
	`pal_receive_notification`	Test agent uses this API to receive notifications.
	`pal_initialize_system`	Contains steps to set system in required state before test execution.
	`pal_print`	Test agent uses this API to dump the test execution output.

The reference PAL implementations are available for Mocker platform and OSPM agent.

2.3 Test execution flow

This section describes the test execution flow for SCMI tests.

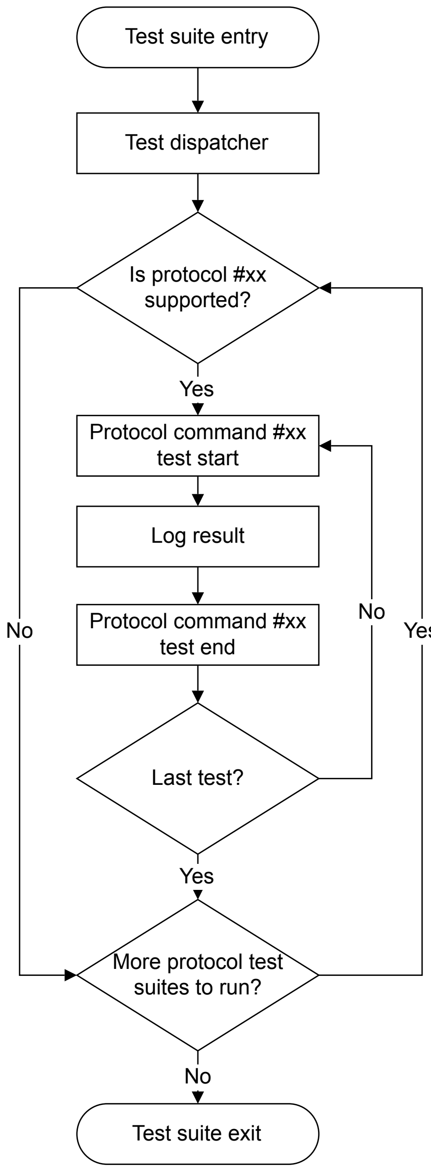
Test suite entry
Test dispatcher
Is protocol #xx supported?
Yes
No
Protocol command #xx test start
Log result
Protocol command #xx test end
No
Yes
Last test?
Yes
More protocol test suites to run?
No
Test suite exit

Figure 2-2 Test execution flow

The following steps are involved in executing the SCMI tests.

1. The tests are built as a library which is linked to the execution environment. The execution environment invokes the test entry point.
2. The test entry point initializes the test environment and calls the test dispatcher function.
3. To discover which protocols are supported by the underlying platform and platform attributes, base protocol tests are run first. If a protocol is not supported by the platform, the protocol tests are skipped.
4. Each test logs its status and when all the protocol tests are executed, a consolidated test report for each protocol is logged.

2.4 Test build and execution

This section provides information on building and executing the SCMI test suite.

Build for self-test mocker platform

A self-test framework is implemented that provides a response to the SCMI commands issued by the test suite. It is used for the purpose of unit-testing. The build and execution steps are detailed in the *User Guide*.

Build for OS-based tests

The test suite can run as an OSPM agent running on Linux. The build and execution steps are detailed in the *User Guide*.

Build for Bare-metal environment

The test suite can run in a bare-metal environment. The build and execution steps are detailed in the *User Guide*.

Appendix A
Revisions

This appendix describes the technical changes between released issues of this book.

It contains the following section:

- *A.1 Revisions* on page Appx-A-24.

A.1 Revisions

Table A-1 Issue 0000-01

Change	Location
First release.	-

Table A-2 Differences between issue 0000-01 and issue 0000-02

Change	Location
Updated the diagram and the table in Test suite directory structure section.	See *1.6 Test suite directory structure* on page 1-16
Updated the diagram and the table in Test platform abstraction section.	See *2.1 Test platform abstraction* on page 2-18
Updated the PAL APIs and the descriptions table.	See *2.2 Overview of test suite layers* on page 2-19
Added another pointer that explains about the build for bare-metal environment.	See *2.4 Test build and execution* on page 2-22