Arm® Firmware Framework for Armv8-A Architecture Compliance Suite

Version 1.0

Validation Methodology

arm

102411_0100_01_en

Arm® Firmware Framework for Arm®v8-A Architecture Compliance Suite

Validation Methodology

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-01	06 April 2021	Non-Confidential	Alpha release

Non-Confidential Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

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® Firmware Framework for Armv8-A Architecture Compliance Suite Validation Methodology

Preface

This preface introduces the *Arm® Firmware Framework for Armv8-A Architecture Compliance Suite Validation Methodology*.

It contains the following:

About this book

This book describes the Architecture Compliance Suite for Arm® Firmware Framework for Armv8-A.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This chapter introduces the features and components of the Architecture Compliance Suite (ACS) for Arm Firmware Framework for Armv8-A.

Chapter 2 Validation methodology

This chapter describes the validation methodology that is used for the Architecture Compliance 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.

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-ff-a-acs@arm.com*. Give:

- The title *Arm Firmware Framework for Armv8-A Architecture Compliance Suite Validation Methodology*.
- The number 102411_0100_01_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.

———————————

Other information

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

Chapter 1
Introduction

This chapter introduces the features and components of the Architecture Compliance Suite (ACS) for Arm Firmware Framework for Armv8-A.

It contains the following sections:

1.1 Scope of the document

The goal of this document is to describe the validation methodology for Arm Firmware Framework for Armv8-A Architecture Compliance Suite. It focuses on describing the framework and the methodology that is used to run the tests.

1.2 Abbreviations

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

Table 1-1 Abbreviations and expansions

Abbreviation	Expansion
ABI	Application Binary Interface
ACS	Architecture Compliance Suite
API	Application Programming Interface
CPU	Central Processing Unit
EL	Exception Level
EP	Endpoint
FF-A	Arm Firmware Framework for A class
FF	Firmware Framework
MP	Multi-Processor
OSPM	Operating System Power Management
PAL	Platform Abstraction Layer
PE	Processing Element
PM	Partition Manager (Represents both SPM and Hypervisor)
PSCI	Power State Coordination Interface
SP	Secure Partition
SPM	Secure Partition Manager
SPMC	SPM Core
SPMD	SPM Dispatcher
SUT	System Under Test
UP	Uni-Processor
VAL	Validation Abstraction Layer
VM	Virtual Machine

1.3 Arm Firmware Framework for Armv8-A

Arm Firmware Framework for Armv8-A (Arm FF-A) describes a software architecture that achieves the following goals:

- Apply the Virtualization Extension to isolate software images provided by different vendors from each other.
- Describe interfaces that standardize communication between the various software images. This includes communication between images in the Secure world and Normal world.

Arm FF-A also goes beyond the mentioned goals to ensure that the interfaces are used to standardize communication:

- In the absence of Virtualization Extensions in the Secure world, this aspect provides a migration path for existing Secure world software images to a system that implements Virtualization Extension in the Secure state.
- Between Virtual Machines (VMs) managed by a hypervisor in the Normal world. The Virtualization Extensions in the Secure state mirror its counterpart in the Non-secure state. A hypervisor could use the FF interfaces to enable communication between the VMs it manages.

Components of Arm FF-A
The main components of Arm FF-A are:

- A Partition Manager (PM), which manages partitions is the hypervisor in Normal world and the Secure Partition Manager (SPM) in Secure world.
- One or more partitions that are sandboxes, created by the PM could be VMs in Normal world or Secure world. The VMs in Secure world are called Secure Partitions (SP).

  ———— **Note** ————

  In this document, the terms endpoint and partition are used interchangeably.

  ————————————

- Application Binary Interfaces (ABIs) that partitions can invoke to communicate with other partitions.
- A partition manifest that describes system resources, requirements, implemented services, and attributes related to governing the runtime behavior of a partition.

———— **Note** ————

For more information, download the Arm FF-A specification from *https://developer.arm.com/docs/den0077/latest*.

————————————

1.4 Architecture Compliance Suite

Architecture Compliance Suite contains the architecture tests that are a set of examples of invariant behaviors that are specified by the Arm FF-A specifications. Use these tests to check that these behaviors are interpreted correctly in your system.

These tests cover checks for the following categories of features, with each suite covering a different area of the architecture.

Table 1-2 Test suite categories and their descriptions

Suite name	Covered features	Description
setup_discovery	FF-A set up and Discovery Interfaces, status reporting interfaces and Partition Initialization.	The directed test cases verifying the implementation of Arm FF-A for the FF-A set up and discovery interfaces, and endpoint setup requirements such as initialization of partition, setting up the Multi-Processor (MP) execution contexts for partition, and Uni-Processor (UP) migrate capability.
direct_messaging	FF-A Direct Messaging Interfaces, status reporting interfaces, and FF-A CPU Cycle Management Interfaces.	The directed test cases verifying the implementation of Arm FF-A for the FF-A Direct Messaging interfaces and FF-A CPU Cycle Management Interfaces.
indirect_messaging	FFA Indirect Messaging Interfaces, status reporting interfaces, and FFA CPU Cycle Management Interfaces.	The directed test cases verifying the implementation of Arm FF-A for the FF-A indirect Messaging interfaces and FF-A CPU Cycle Management Interfaces.
memory_manage	FFA Memory Management Interfaces and status reporting interfaces.	The directed test cases verifying the implementation of Arm FF-A for the FF-A Memory Management interfaces.

Note

The test suite contains tests that have checks embedded within the test code. To view the list of test suites and how these different categories of features are checked for compliance, see `testcase_checklist.md` document in the `doc/` directory.

1.5 ACS components

The components of the ACS are described in the following table.

Table 1-3 Test suite components

Component	Description
Test suites	Contains self-checking tests that are written in C.
Substructure	Test supporting layers consist of a framework and libraries set up as: • Tools to build the compliance tests • Validation Abstraction Layer (VAL) library • Platform Abstraction Layer (PAL) library
Documentation	Suite-specific documents such as test lists, porting guide, and API specification.

1.6 Directory structure

In the 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 files that are shown in the following figure.

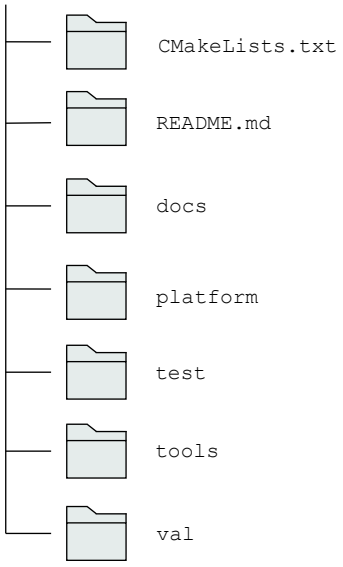

Figure 1-1 Test suite directory structure

The components of the test suite directory structure are described in the following table.

Table 1-4 Test suite directory

Component	Description
CMakeLists.txt	Top-level CMake file contains information about the CMake build support.
docs	Contains the test suite documentation.
platform	Contains files to form the PAL. PAL is the closest to hardware and is aware of the underlying hardware details. Since this layer interacts with hardware, it must be ported or tailored to specific hardware required for system components present in a platform. For more information on porting setups see, `doc/porting_guide.md` document. This layer is also responsible for presenting a consistent interface to the VAL required for the tests.
test	Contains the Arm FF-A tests.
tools	Contains CMake files and scripts that are used to generate test binaries.
val	Contains subdirectories for the VAL libraries. This layer provides a uniform and consistent view of the available test infrastructure to the tests in the test suite. The VAL makes appropriate calls to the PAL to achieve this functionality. This layer is not required to be ported when the underlying hardware changes.

1.7 Feedback, contributions, and support

For feedback, use the GitHub Issue Tracker that is associated with this repository.

For support, send an email to *support-ff-a-acs@arm.com* with the details.

Arm licensees can contact Arm directly through their partner managers.

Arm also welcomes code contributions through GitHub pull requests. See, GitHub documentation on how to raise pull requests.

Chapter 2
Validation methodology

This chapter describes the validation methodology that is used for the Architecture Compliance Suite.

It contains the following sections:

2.1 Test layering details

Arm FF-A tests are self-checking and portable C-based tests with directed stimulus. These tests use the layered software stack approach to enable porting across different test platforms.

The constituents of the layered stack are:

- Tests
- VAL
- PAL

The following figure illustrates System Under Test (SUT) when Non-secure hypervisor component is present in the system.

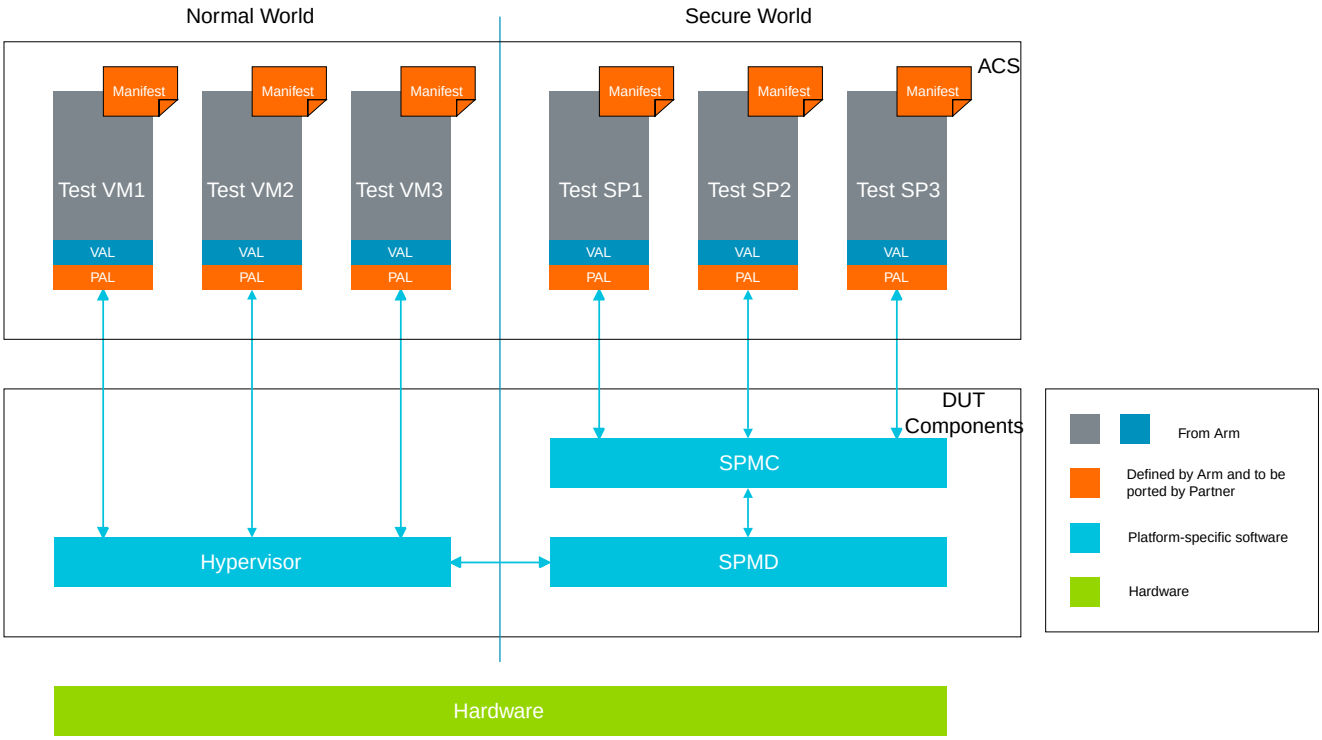

Figure 2-1 SUT when Non-secure Hypervisor is present

The following figure illustrates SUT when a Non-secure hypervisor component is absent in the system.

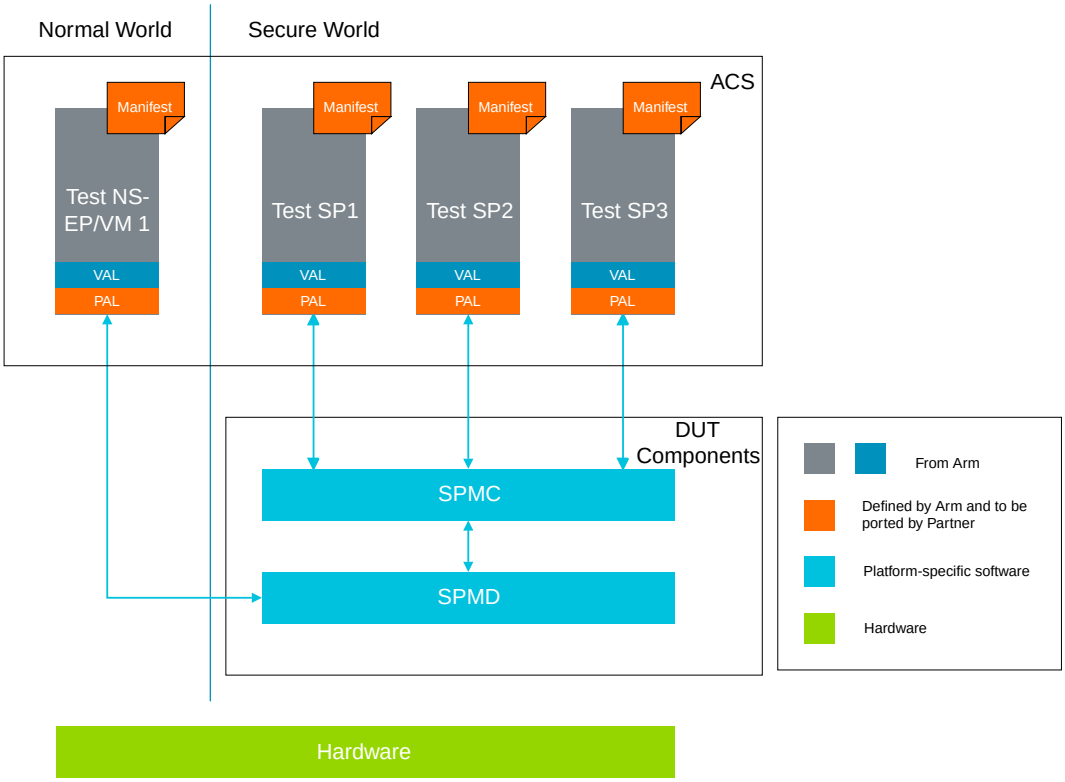

Figure 2-2 Non-secure hypervisor component is absent in SUT

The following table describes the constituents of the layered stack.

Table 2-1 Layered software stack components

Layer	Description
Tests	A set of C and assembly-based directed tests, each of which verifies the implementation against a test scenario that is described by the Arm FF-A specification has Secure and Non-secure test checks. Such tests are launched from Normal world test endpoint. Dedicated instance of client-server test runs on different test VM or SP, depending on the test scenario. Such tests are abstracted from the underlying hardware platform by the VAL. This implies that porting a test for a specific target platform is not required. Each test endpoint is also provided with manifest file describing the resources that it needs for to boot and function.
VAL	This layer provides a uniform and consistent view of the available test infrastructure to the tests in the test pool, by making appropriate calls to the PAL. It is designed in a way that it can be used from both Secure and Non-secure sides. This layer does not require porting when the underlying hardware changes.
PAL	This layer is the closest to the hardware and is aware of the platform details. It is responsible for presenting the hardware through a consistent interface to VAL. This layer must be ported to the specific hardware present in the platform. The PAL is designed in a way that it can be used from both Secure and Normal worlds.

2.2 Test suite organization

The directory structure of Arm FF-A test suite is described in this section.

The following figure shows the contents of the Arm FF-A test suite directory structure.

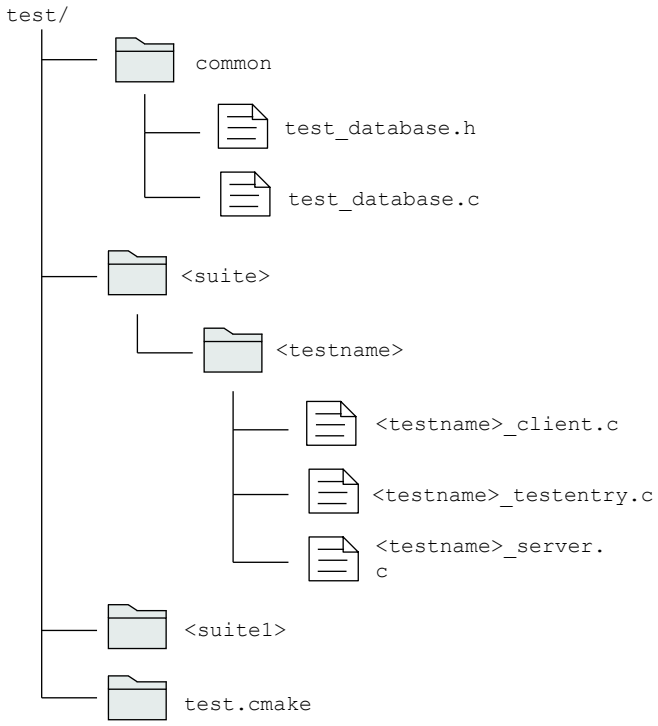

Figure 2-3 Arm FF-A test suite directory structure

The following table shows the contents of the directory files in the Arm FF-A test suite:

Table 2-2 Arm FF-A test suite directory details

Directory file	Description
`Test/`	Top-level test directory.
`Common/`	Contains source that is common for all suites.
`test_database.c`, `test_database.h`	Contains test information such as test name, test client-server functions for all tests from each suite. This database is used by the `val_test_dispatch` API to launch each test one after the other.
`<Suite>`	Contains the tests of a particular test category.
`<testname>`	Test directory containing test sources.
`<testname>_testentry.c`	Holds the test entry point for the test. It is run by the test dispatcher VM in Normal world.
`<testname>_client.c`	Holds client test functions. Source code of this file is duplicated to each of the available test endpoints. This covers different test endpoint interactions using the same client test function code.
`<testname>_server.c`	Holds server test functions. Source code of this file is duplicated to each of the available test endpoints. This covers different test endpoint interactions using the same server test function code.

2.3 Integrating the test suite with the SUT

The test compilation flow creates the following libraries that you must integrate with your SUT software.

Table 2-3 Integrating test binaries

Test binaries	Integration and loading
Secure test endpoint binaries: • `<BUILD_DIR>/output/sp1.bin` • `<BUILD_DIR>/output/sp2.bin` • `<BUILD_DIR>/output/sp3.bin`	These secure test binaries must be integrated and loaded in Secure world using your SPMC component with the help of the information provided in their respective test endpoint manifest files available in `platform/manifest/<platformName>/`. These SPs can be executed at SEL1 or SEL0 depending on whether SPMC is implemented at SEL2 or SEL1.
Test Dispatcher endpoint binary (Normal world): • `<BUILD_DIR>/output/vm1.bin`	This binary can be integrated in two ways: 1. As a bare-metal VM, where NS partition manager performs the integration and loading of the dispatcher binary into Normal world. 2. As a Linux kernel module, to run the VM1 test dispatcher code as part of Linux OS, Linux kernel module files *https://gitlab.arm.com/linux-arm/linux-acs/-/tree/master/ffa-acs-drv* are required. The procedure to build and run FF-A tests on this configuration is described in the README file of the module package. Dispatcher VM must be run at NS-EL1.
Secondary VM binaries: • `<BUILD_DIR>/output/vm2.bin` • `<BUILD_DIR>/output/vm3.bin`	The secondary VM binaries must be integrated and loaded in Normal world using your NS partition manager component with the information provided in respective test endpoint manifest file available in `platform/manifest/<platformName>/`. Secondary VMs must be run at NS-EL1.

Note

To know more about the test binary generation steps, see `README.md`.

2.4 Test execution flow

This section provides details of the test execution flows for Arm FF-A tests.

The sequence of operations for Arm FF-A test execution flow is as follows:

1. SUT boots to an environment that enables the test functionality. This implies that the SPM and hypervisor is initialized, and ACS test endpoints (except for test dispatcher VM) are ready to accept the requests.
2. SUT boot software gives control to the test dispatcher VM in Normal world. The dispatcher VM then invokes the `test_dispatcher` function.
3. Dispatcher VM launches the `<testname>_testentry` function for each test one by one and drives the overall test regression.
4. The dispatcher also makes VAL (and in turn PAL) calls to save and reports each of the test results, then Dispatcher VM sends the message to required test endpoints to release it for executing appropriate test functions as per the test needs.

Based on the test scenario, different test endpoints communicate with each other using FF-A ABIs that are defined in the specification and report the test results using VAL print API (in turn PAL API ported to the specific platform). Each test scenario is driven using dedicated client-server test functions and they are:

1. Available in `<testname>_client.c` and are suffixed with _client label. Based on test needs, client functions are executed in any of the available test endpoints.
2. Available in `<testname>server.c` and are suffixed with _server label. The server functions are invoked on specified test endpoints by the client test function. Test_entry function and information about different requirements for endpoint interactions are specified in `<testname>_testentry.c` file of the test.
3. After completion of client-server test functions, Dispatcher VM collects test status and prints it to console.
4. The tests query the VAL layer to get the necessary information to run the tests. This information can include memory maps, interrupt maps, and hardware controller maps.

Note

To facilitate test reporting and management aspects, the Arm FF-A system contains UART for printing the status of tests. If a display console is not available, PAL can be updated to make the test results available to the external world through other means.

Information about the environment in which a host test harness is running, is beyond the scope of this document. However, it is presumed that the SUT is communicating with the host using serial port, JTAG, Wi-Fi, USB, or any other means that allow for access to the external world.

The following figure illustrates the test execution flow for the Secure and the Non-secure domain.

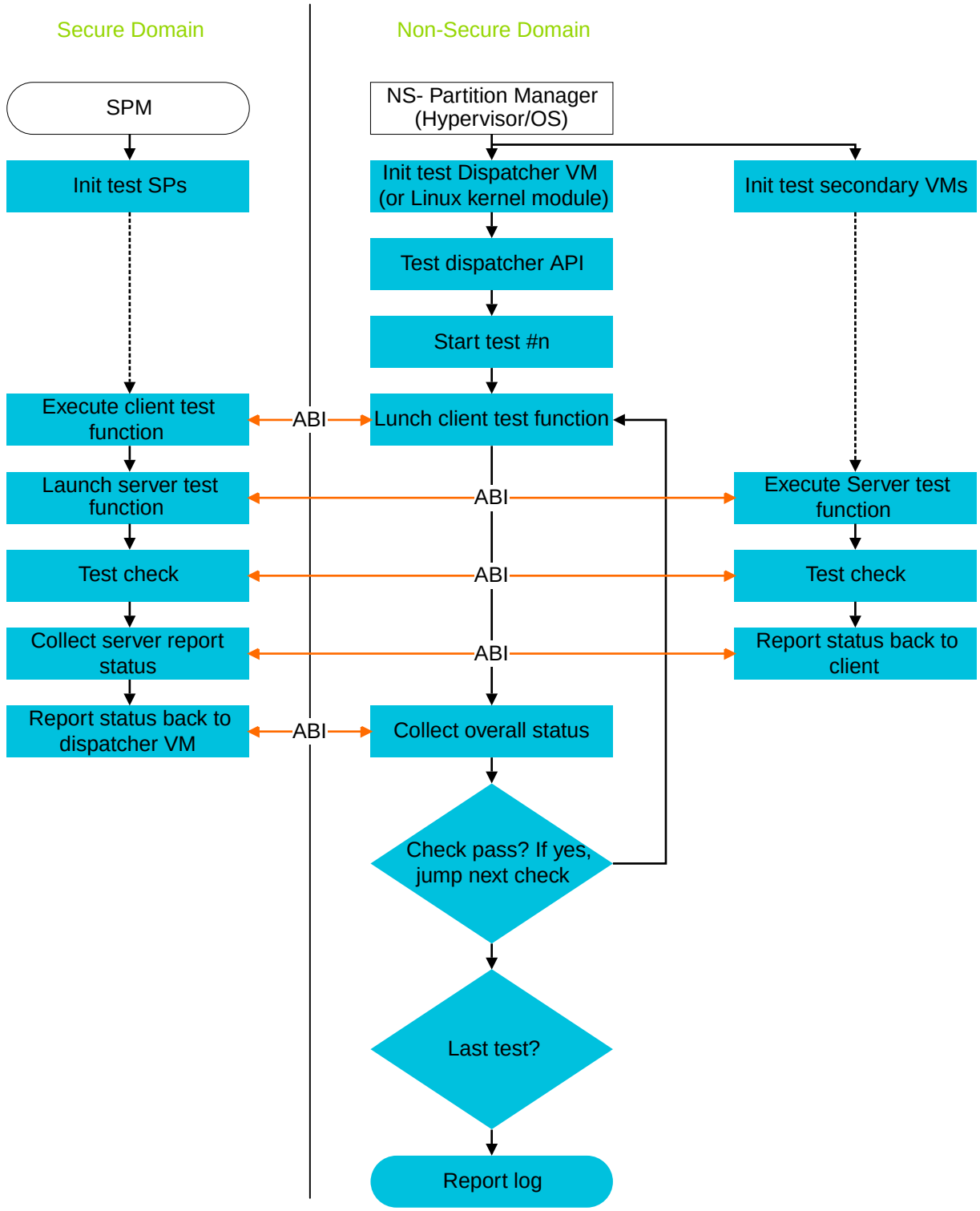

Figure 2-4 Test execution flow for the Arm FF-A

The following figure is an example of a code snippet for test execution flow.

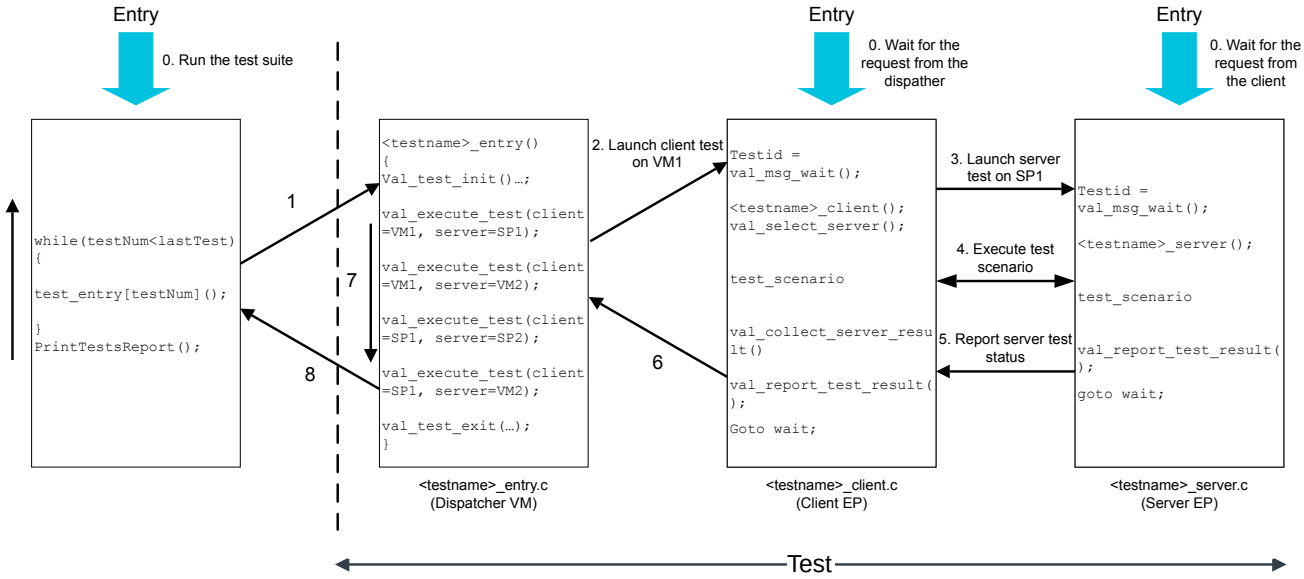

Figure 2-5 Example code snippet for execution flow

The execution steps for the code snippet are described below:

1. Dispatcher VM launches the `<testname>_entry` function for each test one by one and drives the overall test regression.
2. On test entry, Dispatcher VM performs the test initialization as part of `val_test_init` function. Example of such initialization is printing the test name onto the console, enabling the watchdog timer. Afterward, Dispatcher VM selects the client and server pair combination that must be used for executing the test sequence as part of `val_execute_test` function. This function launches client test functions onto the client endpoint specified using logical ID. This function also passes the test-related metadata such as the current test client and server logical ids and test number to the client endpoint.
3. Client launches server test function onto the server endpoint as part of the `val_select_server_fn` function.
4. Client-server endpoint executes the test sequence that is written for the given test scenario.
5. On completion of the test scenario sequence, client endpoint collects the server test function status.
6. Client reports the overall test status for the chosen client-server pair.
7. Dispatcher VM repeats steps 3-6 for the next pair of client and server.
8. Dispatcher VM performs the test exit sequence as part of the `val_test_exit()` and prints the overall test status onto console.
9. Dispatcher VM repeats the preceding steps for all other tests and prints the overall test summary for the suite.

Note

All the handshaking between test dispatcher VM to client EP and client EP to server EP happen using Direct Messaging.

2.5 Test configuration

The majority of Arm FF-A interfaces are common for both the Security state endpoints, the test scenarios for such interfaces must be repeated with different endpoint combinations.

For example, direct messaging test may need to be repeated for VM to SP, SP to SP and VM to VM communication. ACS uses the test configuration mechanism to help in the reuse of test code for multiple endpoint combinations as follows:

1. The test configuration is based on runtime selecting the client-server pair. The `<testname>testentry.c` file of every test enables this selection.
2. The `<testname>testentry.c` file contains repeated calls to `val_execute_test` with different client-id and server-id endpoint ID.
3. `val_execute_test` invokes the client test function on the specified client endpoint which invokes the server test function on the specified server endpoint. This is how the test scenario is executed with the given client and server pair.
4. `val_execute_test` also checks the validity of the endpoint combination for the given target system.
5. It runs the test only when the client and server pair is valid for the target, otherwise it skips the check. For example, for an absent Non-secure hypervisor target, it would skip the VM to VM client-server pair combination.

2.6 Test endpoint booting

As part of initializing a test partition, the PM must program an entry into the first execution context of the test partition. This execution context is hereby called the boot execution context.

For example, the hypervisor is responsible for initializing a VM. It initiates this process by programming an entry into the boot execution context corresponding to a vCPU of the VM. This vCPU is hereby called primary boot CPU.

PM must follow this regardless of a partition has UP or MP execution contexts booting. For setting up the remaining execution contexts for the MP partition, see *MP execution setup* on page 2-27 section.

Using the primary boot CPU, test partition performs the following boot sequence:

1. Clears the BSS region of the test partition image and fix the GOT symbols of the image.
2. Programs the primary boot CPU stack to enable C programs.
3. Programs the VBAR with the default vector table.
4. Programs the GIC for handling partition interrupts.
5. Creates page table and enables the MMU
   - Flat mapping for endpoint regions – text, data, BSS, device (UART, NVM, watchdog).
   - Support for 4k, 16k, 64k TT Granule.
6. Gives control to the Dispatcher function if the current endpoint is Dispatcher or calls ffa_msg_wait ABI to indicate that endpoints are initialized and ready to accept the request.

Note

The EL0 SP would skip step 3-5.

2.7 MP execution context setup

This section describes the MP execution context setup.

PM initializes only the boot execution context. If the partition is an MP endpoint, initialization of other execution contexts (secondary vCPU) must be done through an *IMPLEMENTATION DEFINED* mechanism. For this, ACS relies on PAL APIs to initialize the other partition contexts.

The secondary execution context setup for Secure endpoint is as follows:

1. FF-A components in the Secure world do not perform power management independently from the Normal world. Instead, the SPM Core (SPMC), SPM Dispatcher (SPMD), and SPs are informed about Operating System Power Management (OSPM) operations initiated by the Normal world through Power State Coordination Interface (PSCI) functions.
2. The boot execution context of SP uses `FFA_SECONDARY_EP_REGISTER` interface to register the secondary execution context entry point with SPMC and SPMD for initialization during a secondary cold boot.
3. The secondary execution context of SP uses the registered entry point by SPMC programs when Normal world invokes `PSCI_CPU_ON` interface for the given vCPU.
4. Test SP then sets up the stack for the secondary vCPU. Note that test partition runs with MMU off on secondary vCPU.
5. SP uses `FFA_MSG_WAIT` interface to indicate completion of the secondary execution context to the FF-A framework.

Note

See PSCI documentation, *https://static.docs.arm.com/den0022/d/Power_State_Coordination_Interface_PDD_v1_1_DEN0022D.pdf*

The execution context setup for Test Non-secure Endpoint is as follows:

1. The default implementation of PAL APIs relies on Power State Coordination Interface (PSCI) implementation of PM for the MP execution setups.
2. The primary boot vCPU of the partition uses the `PSCI_CPU_ON` interface to request the PM to initialize another vCPU of the VM or SP. These PAL APIs must be ported if you rely on a framework other than PSCI calls.
3. Upon invoking `PSCI_CPU_ON` interface, PM must release the mentioned secondary vCPU at the partition entry address specified during the `PSCI_CPU_ON` call.
4. Test partition sets up the stack for the secondary vCPU. Note that test partition does not set up the MMU for the secondary vCPU, instead it makes sure that necessary cache maintenance operations are performed for any shared location transactions between vCPUs.
5. Test partition uses the `PSCI_CPU_OFF` interface to exclude calling vCPU from the system for the given partition.

2.8 Error handling

This section defines the test methodology to handle error situations when they occur.

There are two types of error or fault conditions that are possible for tests:

1. In type-1, the generic code encounters an unexpected error situation from which it cannot recover or continue or spin.
2. In type-2, the test performs a sequence to trigger expected error conditions. For example, test endpoint performs unauthorized accesses in which it expects to trigger fault at the PM-level (abort must handle at EL2). Test does this to test the PM behavior for unauthorized accesses.

Test framework handles the type-1 errors as follows:

1. The framework relies on hardware watchdog and non-volatile memory region that must be assigned to Test SP1.
2. The framework waits for watchdog timeout on encounter of error condition and expects watchdog to reset the system so that framework can continue with next available test.
3. The framework uses non-volatile memory to preserve test data over watchdog timer reset.

There are two ways to handle the type-2 errors:

1. If PM supports the injection of faults into originated endpoint (abort injection from higher Exception Level (EL) to lower EL), error handling steps are as follows:
   - Test Endpoint installs handler at the expected vector table location. For example, install synchronous abort handler for stage 2 Data Abort.
   - Test Endpoint performs the authorized access and expects fault-handling at PM.
   - PM injects the fault at lower EL by copying ESR.EC and FAR system registers values.
   - Endpoint receives the abort at the installed handler which fixes the error condition and returns to interrupted code.
2. Use of type-1 error handling, hardware watchdog, and non-volatile memory region-based recovery.
   - Test notifies the expected reboot into NVM just before performing the authorized access.
   - Upon entry into test framework after rebooting, the framework reads the notification flag and checks whether the reboot was intended and marks the test status accordingly.

2.9 Running Test dispatcher VM (VM1) as part of Linux OS

To run VM1 test dispatcher code as part of the Linux OS, Linux kernel module files are required.

These files are available at *https://gitlab.arm.com/linux-arm/linux-acs/-/tree/master/ffa-acs-drv*. The procedure to build and run FF-A tests on this configuration is described in the README file of the module package.

2.10 Analyzing test run results

Each test follows a uniform test structure that is defined by VAL.

1. Performing any test initializations.
2. Dispatching the test functions.
3. Waiting for test completion.
4. Performing the test exit.

The test may pass, fail, skip, or be in an error state. For example, if a test times out or the system hangs, then it means that something went wrong and the test framework was unable to determine the error. In this case, you may have to check the logs. If a test fails or skips, then you may see extra print messages to determine the cause.

The test suite summary is displayed at the end.

```
**** FF-A ACS Version 0.5 ****

Running ‘setup_discovery’ test suite..

Test - ffa_version => START
       Executing test from client=VM1
       Executing test from client=SP1

Test - ffa_version => PASSED

Test – ffa_version => START
       Executing test from client=VM1
       FFA_ERROR_32 -> feature supported
       FFA_SUCCESS_32 -> feature supported
       FFA_SUCCESS_64 -> feature not supported
       FFA_INTERRUPT_32 -> feature supported
       FFA_VERSION_32 -> feature supported
       FFA_FEATURES_32 -> feature supported
       FFA_RX_RELEASE_32 -> feature supported
       FFA_RXTX_UNMAP_32 -> feature not supported
       fid = 0x84000067 must be supported
       (check failed at:test/setup_discovery/ffa_features/ffa_features_client.c,line 58)

Test - ffa_features => FAILED (ERROR CODE = 2)

Test - ffa_id_get => START
       Executing test from client=VM1
       Executing test from client=SP1

Test - ffa_id_get => PASSED

REGRESSION REPORT:
TOTAL TESTS      : 3
TOTAL PASSED     : 2
TOTAL FAILED     : 1
TOTAL SKIPPED    : 0
TOTAL SIM ERROR  : 0

**** END OF ACS ****
Entering Standby..
```

Debugging of a failing test

Since each test is organized with a logical set of self-checking code, if a failure occurs, searching for the relevant self-checking point is a useful point to start debugging.

Consider the above snippet of a failing test on the display console.

Here are some debugging points to consider.

- If the default prints do not give enough information, you can recompile and rerun the test binaries with high print verbosity level. See the test suite build README to understand how the test verbosity can be changed.
- Prints from each of the test endpoints are prefixed with the endpoint name. For example, print from SP1 is prefix with `"SP1:"`
- In case of a test fail,
  - Along with the error message, the test also prints the file and line number from where the error message is printed.
  - Test results contain the error code associated with the error message. The status of the error code is mapped with a structure `val_status_t` that is available at `val/inc/val.h`. Look for the `enum` that is dedicated to this number to see the status in the verbatim form.

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-33.

A.1 Revisions

Table A-1 Issue 0100-01

Change	Location
First release.	-