Arm® TBSAv8-M Architecture Test

Revision: r0p0

Validation Methodology and User Guide

arm

101308_0000-0.6dev_02_en

Arm® TBSAv8-M Architecture Test

Validation Methodology and User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
PJDOC-2042731200-3327	March 2018	Confidential	Pre alpha release
PJDOC-2042731200-3327	March 2018	Confidential	Alpha release
0000-01	15 June 2018	Non-Confidential	Dev0.5 release. Note: The document now follows a new numbering format.
0000-02	29 June 2018	Non-Confidential	Dev0.6 release.

Non-Confidential Proprietary Notice

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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 for an Alpha product, that is a product under development.

Web Address

http://www.arm.com

Contents

Arm® TBSAv8-M Architecture Test Validation Methodology and User Guide

Preface

This preface introduces the *Arm® TBSAv8-M Architecture Test Validation Methodology and User Guide*.

It contains the following:

About this book

This book describes the TBSAv8-M Architecture Test Validation Methodology.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This user guide is written for engineers who are validating an implementation of the Trusted Base System Architecture Test Suites for Armv8-M.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Read this chapter for an introduction to the features and components of the Trusted Base System Architecture Test Suites for Armv8-M.

Chapter 2 Porting steps
Read this chapter for information on configuring the test suite.

Chapter 3 Architecture test suite
Read this chapter for information on the tests that are provided with the test suite.

Chapter 4 Abstraction layer APIs
Read this chapter for information on Abstraction layer APIs.

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.

`mono`space
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.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

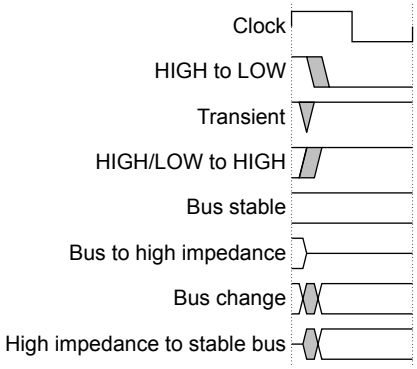

Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level
The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm publications

- *Arm® Platform Security Architecture Trusted Base System Architecture for Armv8-M* version Beta-1 () DEN 0062A.
- *Arm®v8-M Architecture Reference Manual* (ARM DDI 00553A.b).

Other publications
None.

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 *errata@arm.com*. Give:

- The title *Arm TBSAv8-M Architecture Test Validation Methodology and User Guide*.
- The number 101308_0000-0.6dev_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

Read this chapter for an introduction to the features and components of the Trusted Base System Architecture Test Suites for Armv8-M.

It contains the following sections:

1.1 Abbreviations

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

Table 1-1 Abbreviations and expansions

Abbreviation	Expansion
AES	Advanced Encryption Standard
DPM	Debug Protection Mechanism
I2C	Inter-Integrated Circuit
IDAU	Implementation Defined Attribution Unit
MPC	Memory Protection Controller
MPU	Memory Protection Unit
NSC	Non-secure Callable
NVIC	Nested Vector Interrupt Controller
NVM	Non-Volatile Memory
OTP	One-time Programmable
PAL	Platform Abstraction Layer
PE	Processing Element
PPC	Peripheral Protection Controller
SAU	Security Attribution Unit
SPI	Serial Peripheral Interface
TBSA	Trusted Base System Architecture
VAL	Validation Abstraction Layer

1.2 TBSA-v8M test suite

The test suite checks whether an implementation conforms to the behaviors described in the TBSA specifications.

The TBSA-v8M that is described in the TBSA-v8M specification defines the behavior of an abstract machine, referred to as a TBSA-v8M system. Implementations compliant with TBSA-v8M architecture must conform to the described behavior of the TBSA-v8M System.

The test suite includes the following examples:

1. Invariant behaviors that are provided by the TBSA-v8M Architecture Specification. You can use these examples to verify if these behaviors are interpreted correctly.
2. Areas of the architecture that are fundamental, known pitfalls, and common misinterpretations. The tests are not exhaustive and implementers are responsible for device verification.

The TBSA-v8M test suite contains basic and assisted architecture tests. The system designers building a basic implementation of TBSA-v8M architecture must show compliance using the basic architecture test suites. System designers building an assisted implementation of TBSA-v8M must show compliance using both the basic and assisted architecture compliance test suites.

The test suites contain self-checking tests. These tests have checks that are embedded within the test code. Tests are coded in assembly and C.

To facilitate test reporting and management of observing aspects, the TBSA-v8M system must contain at least one UART for printing the status of tests.

This section contains the following subsection:

- *1.2.1 Scope of the document* on page 1-12.

1.2.1 Scope of the document

This document describes the layers of TBSA-v8M test suite and its usage. This document is intended to solicit feedback from partners so that TBSA-v8M test suite can be used agnostic to various system implementations.

Since TBSA-v8M test suite is at Alpha stage, only a subset of tests released with this suite are validated in the Arm internal platform. There are possibilities to have tests where the test code is complete for a particular scenario but cannot be validated in the Arm internal platform.

1.3 Components of the test suite

The test suite consists of the components that are described in the following table.

Components	Description
Suites	The suites are organized to align with the features of the TBSA-v8M architecture. These suites contain self-checking tests that are written in C language.
Substructure	Test supporting layers consist of a framework and libraries setup as: • Scripts to build the test suites • VAL library • PAL library
Documentation	Kit-specific documents.

1.4 Compliance sign-off process

More details on the compliance sign-off or certification process and expectations from partner on this process will be published in the upcoming releases of this document.

1.5 Getting started

This section provides an overview about getting started with the test suite.

This section contains the following subsections:

1.5.1 Directory structure

Validation tests require that the components of the test suite are in a specific hierarchy.

When the test suite release package is downloaded from GitHub, the top-level directory contains the files that are shown in the following figure.

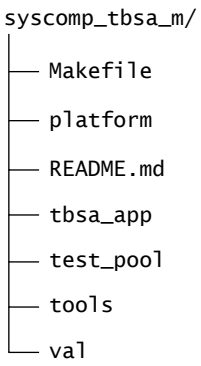

Figure 1-1 Test suite directory structure

platform
This directory contains files to form PAL. PAL is the closest to hardware and is aware of underlying hardware details. Since this layer interacts with hardware, it is ported or tailored to specific hardware required for system components present in a platform. This layer is also responsible for presenting a consistent interface to the validation abstraction layer required for the tests.

tbsa_app
This directory contains the entry point for TBSA-v8M test suites. It is expected from partner that the System under test – boot software would give control to the TBSA-v8M test suite application entry point in Secure privileged mode.

test_pool
This directory contains the test suites. This test suite is a set of C-based directed tests, each of which verifies the implementation against a test scenario that is described by the TBSA-v8M specification. These tests are abstracted from the underlying hardware platform by the VAL.

tools
This directory contains subdirectories for the tools and scripts that are used in the test suite.

val
This directory 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 pool. The VAL makes appropriate calls to the PAL to achieve this functionality. This layer is not ported when the underlying hardware changes.

1.5.2 Software, tools, and licensing requirements

The test suite requires specific versions of software to operate. The following list provides the details of the platform and tools that are needed for the test suite.

- Host Operating System: Ubuntu
- Scripting tools: Perl 5.12.3
- Other open-source tools: GCC 6.3.1

TBSA-v8M test suite is distributed under Apache v2.0 license.

1.5.3 Environment requirements

The following are the minimum memory requirements for TBSA-v8M test suite to run the compliance.

- 64KB of Secure memory and 16KB of Non-secure memory to store VAL or PAL functions and related data
- 16KB of Secure Contiguous Memory for secure tests
- 16KB of Non-secure Contiguous Memory for Non-secure tests
- 4KB of NSC memory

This memory must be usable for both code execution and data access.

Note

- There is no specific requirement on the number of MPU and SAU regions for TBSA-v8M test suites.
- For Non-secure and NSC memory, it is assumed that SAU or IDAU regions are programmed correctly with Non-secure and Non-secure Callable attributes respectively before getting into `tbsa_entry` point.

To learn the details of your hardware environment, the validation tests read the `tbsa_tgt.cfg` file. Refer to *Chapter 2 Porting steps* on page 2-18.

1.6 Feedback, contributions, and support

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

For support, send an e-mail to *support-psa-arch-tests@arm.com* with the details.

Arm licensees can contact Arm directly through their partner managers. Arm welcomes code contributions through GitHub pull requests. See GitHub documentation on how to raise pull requests.

Chapter 2
Porting steps

Read this chapter for information on configuring the test suite.

It contains the following section:

2.1 Target configuration elements

You must populate your system configuration and provide it as an input to test suite.

This is captured in a single static input configuration file that is named as `tbsa_tgt.cfg`. This file is available at `syscomp_tbsa_m/platform/board/<target>.`

An example of the input configuration file is as shown.

```
//PERIPHERALS
timer.num = 2;
timer.0.vendor_id = 0x0;
timer.0.device_id = 0x0;
timer.0.base = 0x50000000;
timer.0.intr_id = 0x8;
timer.0.attribute = SECURE_PROGRAMMABLE;
//MEMORY
sram.num = 2;
sram.1.start = 0x30000000;
sram.1.end = 0x303FFFFF;
sram.1.attribute = MEM_SECURE;
sram.1.mem_type = TYPE_NORMAL_READ_WRITE;
sram.1.dpm_index = 0;
```

Note

If there is a need to add a new type of entity other than the ones listed in `tbsa_tgt.cfg` file, you must contact Arm for further updates.

More details on the structure of the input can be obtained from `val/include/val_target.h`.

This section contains the following subsections:

- *2.1.1 Porting steps to create a new target* on page 2-19.
- *2.1.2 Prerequisites for running DPM related tests in TBSA-v8M test suite* on page 2-20.

2.1.1 Porting steps to create a new target

Since TBSA-v8M test suite is agnostic to various system targets, before building the tests, you must port the files mentioned in the following steps.

Procedure

1. Create a new directory in `platform/board/<platform_name>`. For reference, see the existing platform fvp.
2. The peripheral code exists inside `platform/peripherals/<peripheral_name>`. If `<platform_name>` is using the peripherals that already exist in `platform/peripherals/<peripheral_name>`, then this code can be reused. Otherwise, the code must be ported for platform-specific peripherals.
3. Update `platform/board/<platform_name>/Makefile` with the appropriate path of the peripherals used.
4. Update `platform/board/<platform_name>/src/pal_baremetal_intf.c` with the correct instance of the peripherals used.
5. Update the primary input for the TBSA-v8M tests, that is, target configuration file in `platform/board/<platform_name>/tbsa_tgt.cfg`. Use `platform/boards/fvp/tbsa_tgt.cfg` as reference. Refer `val/include/val_target.h` for structure details.

   Note

   `pal_nvram_read` and `pal_nvram_write` of the reference FVP platform code simulate non-volatility of the data across resets by ensuring that the memory range is not initialized across warm boots. A partner board may choose to simulate the same or provide NVRAM using external storage or Internal Flash.

2.1.2 Prerequisites for running DPM related tests in TBSA-v8M test suite

Since DPM scenarios in TBSA-v8M involve verifying whether debug access is allowed in a target or not based on DPM settings, an external debugger is required.

The handshaking and data transfer protocol between the external debugger and Processing Element is handled by two memory mapped registers named as data and flag register. The description of these registers is given in the following table.

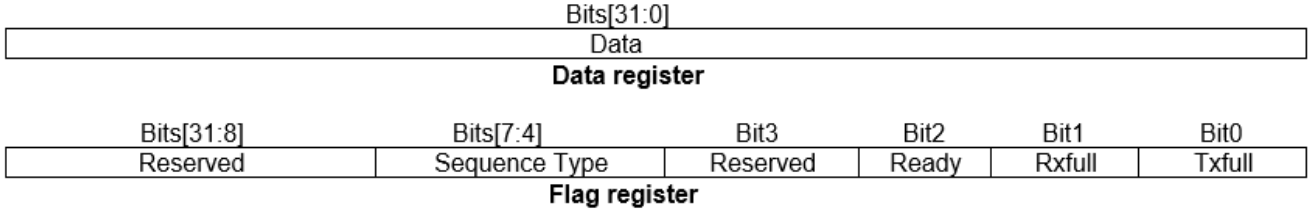

Figure 2-1 Data and flag registers

Table 2-1 Bit field and description

Bit field	Description
Txfull	This bit is set when CPU writes to Data register. This bit is cleared when debugger reads the data from Data register.
Rxfull	This bit is set when debugger writes to the data register and cleared when CPU reads the Data register
Ready	When this bit is set, debugger is ready to receive commands (Data).
Sequence Type	This field indicates debugger to choose appropriate sequence, which the test expects.

Note

- The memory mapped address for these two registers must be chosen in such a way that debug access to these addresses are allowed even when Invasive Debug is not allowed. The address for these two registers should be populated in `syscomp_tbsa_m/boards/<platform_name>/tbsa_tgt.cfg`.
- A reference program for DS-5 external debugger is available at `syscomp_tbsa_m/tools/debug/debugger_script.py`. If you use your specific debugger, then you must port this program to use in your debugger-specific commands.

Chapter 3
Architecture test suite

Read this chapter for information on the tests that are provided with the test suite.

It contains the following sections:

3.1 Test layering details

TBSA-v8M tests are self-checking, portable C-based tests with directed stimulus.

The tests use the layered software-stack approach to enable porting across different test platforms. The constituents of the layered stack are:

1. Test suite
2. VAL
3. PAL

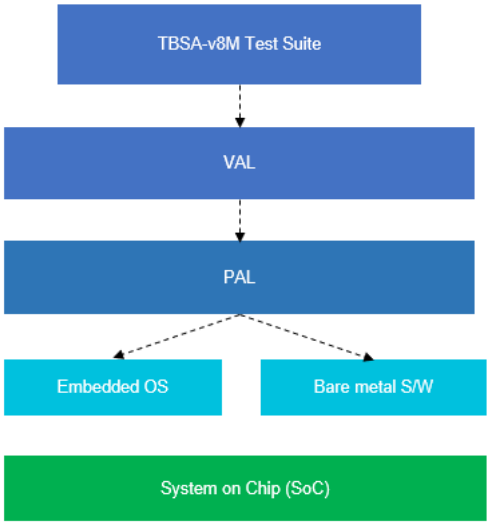

Figure 3-1 Layered software stack

The following table describes the different layers of the test.

Table 3-1 Test layers and descriptions

Layer	Description
Test suite	Collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL.
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 pool. The VAL makes appropriate calls to the PAL to achieve this functionality. This layer is not ported when the underlying hardware changes.
PAL	This layer is the closest to hardware and is aware of underlying hardware details. Since it interacts with hardware, it must be ported or tailored to specific hardware required for system components present in a platform. It is also responsible for presenting a consistent interface to the validation abstraction layer required for the tests.

3.2 Build flow

Each test file must have a metadata section that describes the high-level details of the test such as Test ID, test name, and secure specification requirement. The metadata is filled in the form of macros.

As a precompile step, the metadata in each file is extracted, parsed, and saved in a separate file. The metadata file is used during the package creation phase to fill the `tbsa_acs_hdr`. VAL and PAL files are not required to have a metadata section.

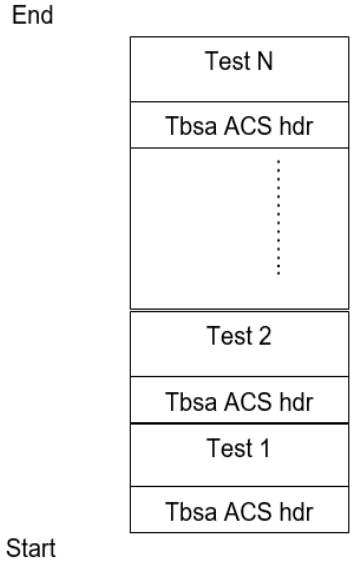

Figure 3-2 Build flow

Build steps

To compile and build the test images for the complete ACK, use the following command:

```
make TARGET=<target_directory_name>
```

For example,

```
make TARGET=fvp
```

Build outputs

The output of the build process is a collection of tests along with metadata of each test packaged as a single binary file and an ELF file. These files are:

- `tbsa_test_combined.bin (./test_pool/out/)`
- `tbsa.elf (./out/)`

The following figure shows the directory structure after building the test ELF images.

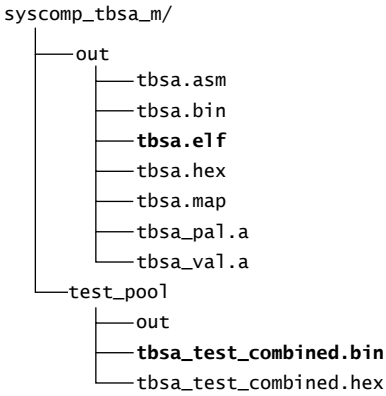

Figure 3-3 TBSA_ACKHOME directory structure

3.3 Test execution flow

The *System Under Test* (SUT) boots to an environment that enables the PAL functionality.

At this point, the SUT boot software gives control to the TBSA-v8M ACS application entry point `tbsa_entry` that comes from ELF header of `tbsa.elf` image in Secure privileged mode.

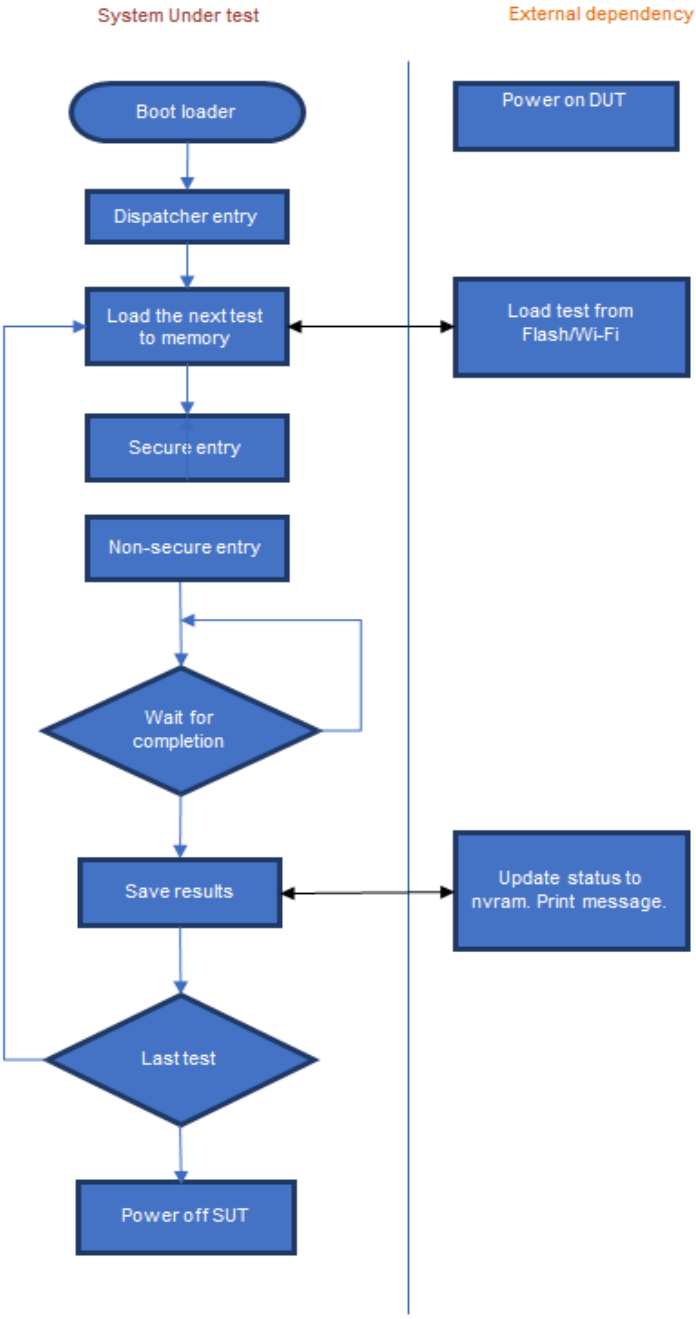

Figure 3-4 Test execution flow

The TBSA-v8M test suite application queries the VAL layer to get the necessary information to run the tests. This information includes memory maps, interrupt maps, and hardware controller maps.

Due to RAM and flash size constraints, all the tests may not be available at the same time. The dispatcher in the VAL queries the PAL to load the next test on the completion of the present test. The

PAL may optionally communicate with the external world to load the next test. Also, the dispatcher makes VAL (and in turn PAL) calls to save and report each of the test results.

Each test must present Secure and Non-secure entry points. If a scenario does not warrant either Secure or Non-secure functionality, then the entry functions will be empty.

To achieve some of the above-mentioned functions, the PAL may optionally make calls that are handled outside the system under test.

The following are some of the responsibilities that require external support:

- Feed the tests to the design under test.
- Collate and print the test status and messages.

Power ON/OFF the system as required by the test sequence. The environment in which a host test harness is running is beyond the scope of this document. But, it is envisioned 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.

3.4 Test dispatcher

Each test must present the following test entry points to the dispatcher.

1. Secure
   a. Entry hook
   b. Payload
   c. Exit hook
2. Non-secure
   a. Entry hook
   b. Payload
   c. Exit hook

To each of the entry points, the dispatcher passes a pointer to a structure containing the function pointers to all the available VAL functions. The Secure entry points receive the function pointers to the Secure VAL APIs. The Non-secure entry points receive function points to the Non-secure wrapper functions in the NSC region. These functions make the appropriate secure VAL function call.

The dispatcher first makes the Secure function calls and on success, calls the Non-secure functions.

The flow of the dispatcher is described in the following steps.

1. Request VAL to load the metadata of the next test into the main memory.
2. Parse the metadata that is associated with the test.
3. Verify that test is compatible with the system under test.
4. Load the test code and data sections to the appropriate locations in the main memory.
5. Call the Secure entry hook function of the test.
6. Call the Secure payload test function.
7. Call the Secure exit hook.
8. Call the Non-secure entry hook function.
9. Call the Non-secure payload function.
10. Call the Non-secure exit hook function.
11. Signal test completion and log test status.

The results from execution of each test are saved to memory. If a display console is not available, the PAL must make available the test results to the external world through other means.

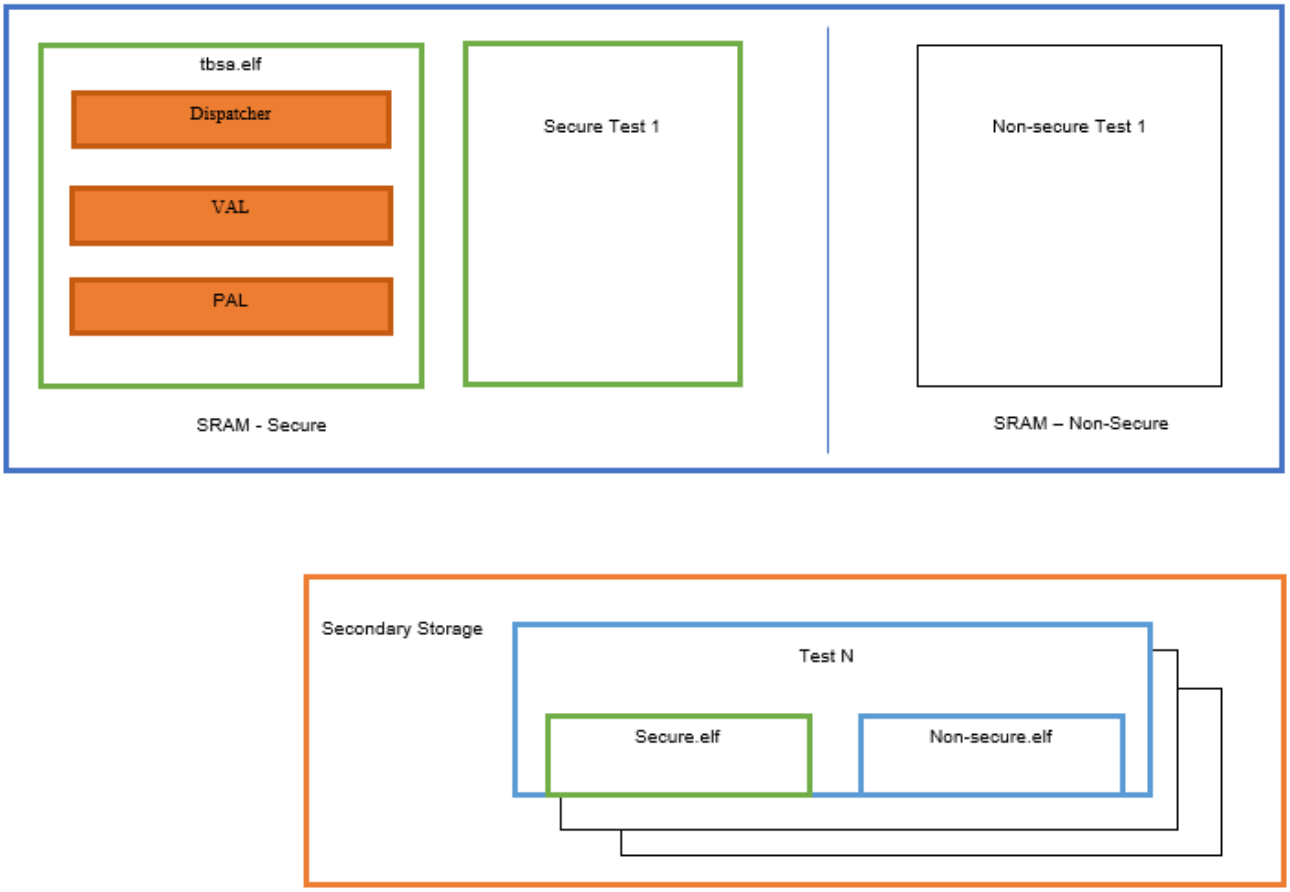

Figure 3-5 Loading a test from secondary storage

The `tbsa.elf` is loaded by the bootloader or embedded OS to the Secure memory. The dispatcher within the `tbsa.elf` loads each of the Secure test section to Secure memory and Non-secure test section to Non-secure memory. The addresses of the various sections are predetermined for a given platform as defined by the user and vary for different targets.

3.5 Test naming conventions

TBSA-v8M tests are named based on the components as defined in the following table.

Table 3-2 Test naming conventions

Component name	Base address	Test number range
Base	0	1-19
Boot	1	21-39
Crypto	2	41-59
Debug	3	61-79
EIP	4	81-99
Interrupts	5	101-119
Secure RAM	6	121-139
Peripherals	7	141-159
Trusted timers	8	161-179
Version counters	9	181-199

Each component can have a maximum of 19 tests. Hence the test numbers will be incremented from their respective base values.

For example, if you see a display as shown below, then for the `Secure RAM`, the component base value is 200. The test name reports it as 201, which means that it is the first test in the `Secure RAM` component.

```
-------------------- Secure RAM Tests --------------------
201 : "R180_TBSA_INFRA"
```

3.6 Test status reporting

When the test suite is run on a target platform, each successfully run test of the suite must report either PASS or SKIP.

The following is a snippet of a successful pass test on the display console.

```
----------------------- Secure RAM Tests   -----------------------
201 : "R180_TBSA_INFRA"
      "Secure RAM access from Trusted world only"
      # Secure                                                PASS
      # Non Secure                                            PASS
```

The following is a snippet of a successful skip test on the display console.

```
----------------------- Secure RAM Tests   -----------------------
201 : "R180_TBSA_INFRA"
      "Secure RAM access from Trusted world only"
      # Secure                                                SKIP
      # Non Secure                                            SKIP
```

This section contains the following subsection:

- *3.6.1 Debugging of a failing test* on page 3-30.

3.6.1 Debugging of a failing test

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

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

```
----------------------- Trusted Boot Tests   -----------------------
11  : "R020/R090_TBSA_BOOT"
      "Trusted boot operation from Trusted and Non-trusted world"
      # Secure
        Checkpoint C01 : Status = 8C
                                                                FAIL
      # Non Secure
                                                                FAIL
```

Here are some debugging points to consider.

1. This test is failing for ‘Trusted Boot’ component. Hence the test should be under `test_pool/boot/` directory.
2. The test ID is 11 which means first test in ‘Trusted Boot’ component. Hence the test is `test_pool/boot/test_s001` directory.
3. Each test will have a Secure portion and a Non-secure portion. In the above snippet, the failure is from the Secure portion. Hence it is required to see `test_pool/boot/test_s001/secure.c` file.
4. Since the failure is shown as ‘Checkpoint C01, the first checkpoint is failing with a status ‘8C’.
5. Status of the failure is mapped with a structure `tbsa_status_t` that is available at `val/include/val_common.h`. In this example, ‘8C’ means that the test is failing for an incorrect value.

Chapter 4
Abstraction layer APIs

Read this chapter for information on Abstraction layer APIs.

It contains the following section:

4.1 PAL APIs

The following table shows the PAL APIs and their descriptions.

Table 4-1 PAL APIs and descriptions

Function name	Prototype	Description
`pal_NVIC_EnableIRQ`	`void pal_NVIC_EnableIRQ(uint32_t intr_num);`	Enables interrupt. `intr_num`: interrupt number.
`pal_NVIC_DisableIRQ`	`void pal_NVIC_DisableIRQ(uint32_t intr_num);`	Disables interrupt. `intr_num`: interrupt number.
`pal_NVIC_ClearTargetState`	`uint32_t pal_NVIC_ClearTargetState(uint32_t intr_num);`	Clears interrupt target state. `intr_num`: interrupt number. It returns 0 if the interrupt is assigned to Secure, and 1 if the interrupt is assigned to Non-secure.
`pal_NVIC_SetTargetState`	`uint32_t pal_NVIC_SetTargetState(uint32_t intr_num);`	Sets interrupt target state. `intr_num`: interrupt number. It returns 0 if the interrupt is assigned to Secure, and 1 if the interrupt is assigned to Non-secure.
`pal_NVIC_SetPriority`	`void pal_NVIC_SetPriority(uint32_t intr_num, uint32_t priority);`	Sets interrupt priority. `intr_num`: interrupt number. `priority`: priority to set.
`pal_NVIC_GetPriority`	`uint32_t pal_NVIC_GetPriority(uint32_t intr_num);`	Gets interrupt priority. `intr_num`: interrupt number. It returns the interrupt priority.
`pal_NVIC_SetPendingIRQ`	`void pal_NVIC_SetPendingIRQ(uint32_t intr_num);`	Sets pending interrupt. `intr_num`: interrupt number.
`pal_NVIC_GetPendingIRQ`	`uint32_t pal_NVIC_GetPendingIRQ(uint32_t intr_num);`	Clears the pending interrupt. `intr_num`: interrupt number.
`pal_NVIC_ClearPendingIRQ`	`void pal_NVIC_ClearPendingIRQ(uint32_t intr_num);`	Gets pending interrupt. `intr_num`: interrupt number. It returns 0 if interrupt status is not pending, and returns 1 if interrupt status is pending.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_NVIC_GetActive	uint32_t pal_NVIC_GetActive(uint32_t intr_num);	Gets active interrupts. intr_num: interrupt number. It returns 0 if interrupt status is not active, and returns 1 if interrupt status is active.
pal_i2c_init	void pal_i2c_init(void);	Initializes the I2C peripheral.
pal_i2c_read	int pal_i2c_read(uint32_t slv_addr, uint8_t *rd_data, uint32_t len);	Reads peripherals using I2C. slv_addr: address of the peripheral. rd_data: read buffer. len: length of the read buffer in bytes. It returns the error status.
pal_i2c_write	int pal_i2c_write(uint32_t slv_addr, uint8_t *wr_data, uint32_t len);	Writes peripherals using I2C. slv_addr: address of the peripheral. wr_data: write buffer. len: length of the write buffer in bytes.
pal_spi_init	void pal_spi_init(void);	Initializes the SPI peripheral.
pal_spi_read	int pal_spi_read(uint32_t addr, uint8_t *data, uint32_t len);	Reads peripherals using SPI commands. addr: address of the peripheral. data: read buffer. len: length of the read buffer in bytes.
pal_spi_write	int pal_spi_write(uint32_t addr, uint8_t *data, uint32_t len);	Writes peripherals using SPI commands. addr: address of the peripheral. data: write buffer. len: length of the write buffer in bytes. It returns the error status.
pal_crypto_init	void pal_crypto_init(addr_t crypto_base_addr);	Initializes the cryptographic functions.
pal_timer_init	int pal_timer_init (addr_t base_addr, uint32_t time_us, uint32_t timer_tick_us);	Initializes a hardware timer. base_addr: base address of the timer module. time_us: time in micro seconds. timer_tick_us: number of ticks per micro second. It returns the SUCCESS or FAILURE.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_timer_enable	int pal_timer_enable (addr_t base_addr);	Enables a hardware timer. base_addr: base address of the timer module. It returns the SUCCESS or FAILURE.
pal_timer_disable	int pal_timer_disable (addr_t base_addr);	Disables a hardware timer. base_addr: base address of the timer module. It returns the SUCCESS or FAILURE.
pal_timer_interrupt_clear	int pal_timer_interrupt_clear (addr_t base_addr);	Clears the interrupt status of the timer. base_addr: base address of the timer module. It returns the SUCCESS or FAILURE.
pal_wd_timer_init	int pal_wd_timer_init (addr_t base_addr, uint32_t time_us, uint32_t timer_tick_us);	Initializes a hardware watchdog timer. base_addr: base address of the watchdog module. time_us: time in micro seconds. timer_tick_us: number of ticks per micro second. It returns the SUCCESS or FAILURE.
pal_wd_timer_enable	int pal_wd_timer_enable (addr_t base_addr);	Enables a hardware watchdog timer. base_addr: base address of the watchdog module. It returns the SUCCESS or FAILURE.
pal_is_wd_timer_enabled	int pal_is_wd_timer_enabled (addr_t base_addr);	Disables a hardware watchdog timer. base_addr: base address of the watchdog module. It returns the SUCCESS or FAILURE.
pal_wd_timer_disable	int pal_wd_timer_disable (addr_t base_addr);	Checks whether hardware watchdog is enabled. base_addr: base address of the watchdog module. It returns 1 if timer is enabled and 0 if the timer is disabled.
pal_crypto_aes_init	void pal_crypto_aes_init(void);	Initializes the specified AES context. It must be the first API called before using the context.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_crypto_aes_setkey_enc	int pal_crypto_aes_setkey_enc(uint8_t *key, uint32_t keysize);	Sets an encryption key. key: the encryption key. keysize: the size of the data passed in bits. Valid options are 128 bits, 192 bits, and 256 bits. It returns 0 if successful, or ERR_AES_INVALID_KEY_LENGTH/ ERR_AES_INVALID_INPUT_LENGTH
pal_crypto_aes_setkey_dec	int pal_crypto_aes_setkey_dec(uint8_t *key, uint32_t keysize);	Sets the decryption key. key: the decryption key. keysize: the size of the data passed in bits. Valid options are 128 bits, 192 bits, and 256 bits. It returns 0 if successful, or ERR_AES_INVALID_KEY_LENGTH otherwise.
pal_crypto_aes_crypt_ecb	int pal_crypto_aes_crypt_ecb(int mode, uint8_t *input, uint8_t *output);	This function performs an AES-ECB single-block encryption or decryption operation. It performs the operation defined in the mode parameter (encrypt or decrypt), on the input data buffer defined in the input parameter. mbedtls_aes_init(), and either mbedtls_aes_setkey_enc() or mbedtls_aes_setkey_dec() must be called before the first call to this API with the same context. mode: the AES operation: AES_ENCRYPT or AES_DECRYPT input : 16-byte input block. output : 16-byte output block. It returns 0 if successful.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
`pal_crypto_aes_crypt_cbc`	`int pal_crypto_aes_crypt_cbc(int mode, uint64_t len, uint8_t *iv, uint8_t *input, uint8_t *output);`	Performs an AES-CBC encryption or decryption operation on full blocks. It performs the operation defined in the mode parameter (encrypt or decrypt), on the input data buffer defined in the input parameter. It can be called as many times as needed, until all the input data is processed. `mbedtls_aes_init()`, and either `mbedtls_aes_setkey_enc()` or `mbedtls_aes_setkey_dec()` must be called before the first call to this API with the same context. `mode` : The AES operation: AES_ENCRYPT or AES_DECRYPT `length` : The length of the input data in bytes. This must be a multiple of the block size (16 bytes). `iv` : initialization vector (updated after use). `input` : 16-byte input block `output` : 16-byte output block It returns 0 if successful.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
`pal_crypto_aes_crypt_cfb`	`int pal_crypto_aes_crypt_cfb(int mode, uint64_t len, uint64_t *iv_offset, uint8_t *iv, uint8_t *input, uint8_t *output);`	Performs an AES-CFB128 encryption or decryption operation. It performs the operation defined in the mode parameter (encrypt or decrypt), on the input data buffer defined in the input parameter. For CFB, you must set up the context with `mbedtls_aes_setkey_enc()`, regardless of whether you are performing an encryption or decryption operation, that is, regardless of the mode parameter. This is because CFB mode uses the same key schedule for encryption and decryption. `mode` : The AES operation: AES_ENCRYPT or AES_DECRYPT `length` : The length of the input data in bytes. This must be a multiple of the block size (16 bytes). `iv_offset`: The offset in IV (updated after use). `iv` : Initialization vector (updated after use). `input` : 16-byte input block `output` : 16-byte output block It returns 0 if successful.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
`pal_crypto_aes_crypt_ctr`	`int pal_crypto_aes_crypt_ctr(uint64_t len, uint64_t *nc_offset, uint8_t *nonce_cntr, uint8_t *stream_block, uint8_t *input, uint8_t *output);`	Performs an AES-CTR encryption or decryption operation. It performs the operation defined in the mode parameter (encrypt or decrypt), on the input data buffer defined in the input parameter. For CTR, you must set up the context with `mbedtls_aes_setkey_enc()`, regardless of whether you are performing an encryption or decryption operation, that is, regardless of the mode parameter. This is because CTR mode uses the same key schedule for encryption and decryption. `length` : The length of the input data in bytes. This must be a multiple of the block size (16 bytes). `nc_offset` : The offset in the current stream_block, for resuming within the current cipher stream. The offset pointer should be 0 at the start of a stream. `nonce_counter` : The 128-bit nonce and counter. `stream_block` : The saved stream block for resuming. This is overwritten by the function. `input` : 16-byte input block `output` : 16-byte output block It returns 0 if successful.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_crypto_aes_crypt_ofb	int pal_crypto_aes_crypt_ofb(uint64_t len, uint64_t *nc_offset, uint8_t *nonce_cntr, uint8_t *stream_block, uint8_t *input, uint8_t *output);	Performs an AES-CTR encryption or decryption operation. It performs the operation defined in the mode parameter (encrypt or decrypt), on the input data buffer defined in the input parameter. For CTR, you must set up the context with mbedtls_aes_setkey_enc(), regardless of whether you are performing an encryption or decryption operation, that is, regardless of the mode parameter. This is because CTR mode uses the same key schedule for encryption and decryption. length : The length of the input data in bytes. This must be a multiple of the block size (16 bytes). nc_offset : The offset in the current stream_block, for resuming within the current cipher stream. The offset pointer should be 0 at the start of a stream. nonce_counter : The 128-bit nonce and counter. stream_block : The saved stream block for resuming. This is overwritten by the function. input : 16-byte input block output : 16-byte output block It returns 0 if successful.
pal_crypto_sha256_init	void pal_crypto_sha256_init(void);	Initializes the SHA256 context. It must be the first API called before using the context.
pal_crypto_sha256_start	int pal_crypto_sha256_start(uint32_t is_224);	Starts a SHA-224 or SHA-256 checksum calculation. is_224 : determines which function to use. 0: Use SHA-256 1: Use SHA-224 It returns 0 on success.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_crypto_sha256_update	int pal_crypto_sha256_update(uint8_t *input, uint64_t ilen);	Feeds an input buffer into an ongoing SHA-256 checksum calculation. input : the buffer holding the data. ilen : the length of the input data. It returns 0 on success.
pal_crypto_sha256_finish	int pal_crypto_sha256_finish(uint8_t *output);	Finishes the SHA-256 operation and writes the result to the output buffer. output : the SHA-224 or SHA-256 checksum result. It returns 0 on success.
pal_crypto_aes_generate_key	int pal_crypto_aes_generate_key(uint8_t *key, uint32_t size);	Generates AES key using various specified entropy sources. key : the buffer where the generated key is stored. size : size of the key to be generated. Valid options are 128 bits, 192 bits, or 256 bits It returns 0 on success.
pal_crypto_compute_hash	int pal_crypto_compute_hash(unsigned char *input, size_t ilen, unsigned char *output, int algo);	Calculates the SHA-224 or SHA-256 checksum of a buffer. input : the buffer holding the data. ilen : the length of the input data. output : the SHA-224 or SHA-256 checksum result. algo : determines which function to use. 0: Use SHA-256. 1: Use SHA-224. It returns 0 on success.
pal_uart_init	void pal_uart_init (addr_t uart_base_addr);	Initializes the UART. uart_base_addr : base address of the UART
pal_uart_tx	void pal_uart_tx (uint8_t data);	Sends data to UART TX FIFO. data : data to be written to TX FIFO.
pal_get_target_cfg_start	void *pal_get_target_cfg_start(void);	Provides the database source location. It returns base address of database.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_nvram_write	int pal_nvram_write(addr_t base, uint32_t offset, void *buffer, int size);	Writes 'size' bytes from buffer into NVRAM at a given 'base + offset'. base : base address of NVRAM. offset : offset. buffer : pointer to source address. size : number of bytes. It returns the error status.
pal_nvram_read	int pal_nvram_read (addr_t base, uint32_t offset, void *buffer, int size);	Reads 'size' bytes from NVRAM at a given 'base + offset' into given buffer. base : base address of NVRAM offset : offset buffer : pointer to source address size : number of bytes It returns 1 or 0.
pal_system_warm_reset	void pal_system_warm_reset(void);	Generates system Warm reset.
pal_system_cold_reset	void pal_system_cold_reset(void);	Generates system Cold reset.
pal_is_cold_reset	int pal_is_cold_reset(void);	Reports the last reset type. It returns YES(1) or NO(0).
pal_is_warm_reset	int pal_is_warm_reset(void);	Reports the last reset type. It returns YES(1) or NO(0).
pal_dpm_set_access_ns_only	int pal_dpm_set_access_ns_only(uint32_t index, bool_t access_ns);	Sets the debug permission based on the input argument. index : DPM index. access_ns : TRUE - allow debug access only for Non-secure address. FALSE - allow debug access to both Secure and Non-secure addresses. It returns the error status.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_mpc_configure_mem_to_nonsecure	int pal_mpc_configure_mem_to_nonsecure (addr_t start_addr,addr_t end_addr);	Allows a memory region to be configured as Non-secure via MPC. instance : MPC instance number. addr : memory address to be configured by MPC. sec_attr : map the address to either Secure or Non-secure security attribute. It returns the error status.
pal_mpc_configure_mem_to_secure	int pal_mpc_configure_mem_to_secure (addr_t start_addr,addr_t end_addr);	Allows a memory region to be configured as Secure via MPC. instance : MPC instance number. addr : memory address to be configured by MPC. sec_attr : map the address to either Secure or Non-secure security attribute. It returns the error status.
pal_fuse_read	int pal_fuse_read(addr_t addr, uint32_t *data, size_t size);	Read the value of given fuse address. addr : address of the fuse. data : buffer to store the data. size : number of words to be read. It returns the error status.
pal_fuse_write	int pal_fuse_write(addr_t addr, uint32_t *data, size_t size);	Writes the value in given fuse address. addr : address of the fuse. data : data to be written. It returns the error status.
pal_fuse_count_zeros_in_rotpk	int pal_fuse_count_zeros_in_rotpk(uint32_t *zero_cnt);	Counts the number of zeros in ROTPK. zero_cnt : buffer to store the zero count. It returns the error status.
pal_fuse_count_zeros	void pal_fuse_count_zeros(uint32_t value, uint32_t *zero_cnt);	Counts the number of zeros in the given value. value : number of zeros to be determined. zero_cnt : buffer to store the zero count.
pal_fuse_get_lcs	int pal_fuse_get_lcs(uint32_t *pLcs);	Reads the LCS register. It returns the error status.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_crypto_validate_certificate	int pal_crypto_validate_certificate(uint32_t certificate_base_addr, uint32_t public_key_addr, uint32_t certificate_size, uint32_t public_key_size);	Validates the certificate using public key. certificate_base_addr : base address of the certificate where it is stored in memory. public_key_addr : base address of the public key where it is stored in memory. certificate_size : certificate memory size public_key_size : public key memory size It returns 0 on success.
pal_crypto_get_uniqueID_from_certificate	int pal_crypto_get_uniqueID_from_certificate(uint32_t certificate_base_addr, uint32_t public_key_addr, uint32_t certificate_size, uint32_t public_key_size);	Gets unique ID from valid certificate using public key. certificate_base_addr : base address of the certificate where it is stored in memory. public_key_addr : base address of the public key where it is stored in memory. certificate_size : certificate memory size. public_key_size : public key memory size It returns the unique ID of the certificate.
pal_rtc_init	int pal_rtc_init(addr_t base_addr);	Initializes RTC. base_addr: base address of the given RTC instance. It returns the error status.
pal_is_rtc_trustable	int pal_is_rtc_trustable(addr_t base_addr);	RTC validity mechanism to indicate whether RTC is Trusted or Non-trusted. base_addr: base address of the given RTC instance. It returns the error status.
pal_is_rtc_synced_to_server	int pal_is_rtc_synced_to_server(addr_t base_addr);	RTC validity mechanism to indicate RTC is synced with server or not. base_addr: base address of the given RTC instance. It returns the error status.

Table 4-1 PAL APIs and descriptions (continued)

Function name	Prototype	Description
pal_crypto_get_dpm_from_key	int pal_crypto_get_dpm_from_key(uint32_t public_key_addr, uint32_t public_key_size, uint32_t *dpm_field);	Gets DPM field from public key. public_key_addr : base address of the public key where it is stored in memory. public_key_size : public key memory size. dpm_field : buffer to store DPM number. It returns the error status.
pal_crypto_get_dpm_from_certificate	int pal_crypto_get_dpm_from_certificate(uint32_t certificate_base_addr, uint32_t certificate_size, uint32_t *dpm_field);	Gets DPM field from certificate. certificate_base_addr : base address of the certificate where it is stored in memory. certificate_size : certificate memory size. dpm_field : buffer to store DPM number. It returns the error status.
pal_firmware_version_update	int pal_firmware_version_update(uint32_t instance, firmware_version_type_t firmware_version_type, uint32_t fw_ver_cnt);	Updates the firmware version. instance : instance if the firmware. fw_ver_cnt : version of the firmware. It returns the error status.
pal_firmware_version_read	int pal_firmware_version_read(uint32_t instance, firmware_version_type_t firmware_version_type);	Reads the firmware version. instance : instance if the firmware. It returns the error status.

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-46.

A.1 Revisions

Table A-1 Issue PJDOC-2042731200-3327

Change	Location	Affects
This is the first revision of the document.	-	All revisions

Table A-2 Differences between Issue PJDOC-2042731200-3327 and Issue 101308-01

Change	Location	Affects
Updated directory structure.	See *1.5.1 Directory structure* on page 1-15.	All revisions
Updated porting steps.	See *2.1.1 Porting steps to create a new target* on page 2-19.	All revisions
Updated test naming conventions.	See *3.5 Test naming conventions* on page 3-29.	All revisions

Table A-3 Differences between Issue 101308-01 and Issue 101308-02

Change	Location	Affects
Updated test naming conventions table.	See *3.5 Test naming conventions* on page 3-29	All revisions.
Updated PAL APIs.	See *4.1 PAL APIs* on page 4-32	All revisions.