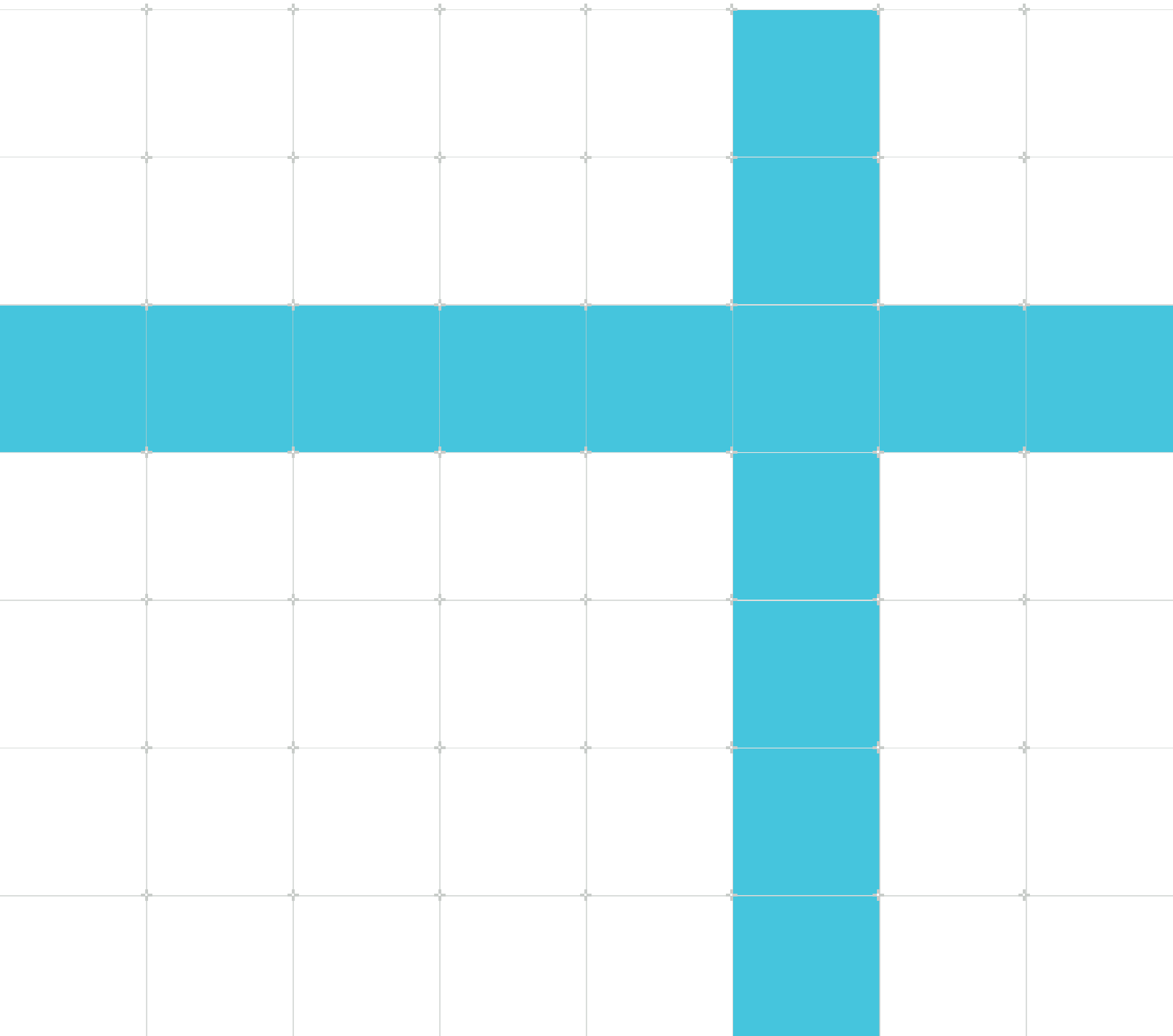
Arm® RME System Architecture Compliance Suite

Version 2.0

Validation Methodology

Non-Confidential

Issue 01
108004_0200_01_en



Arm® RME System Architecture Compliance Suite
Validation Methodology

Release Information

Document history

Issue	Date	Confidentiality	Change
0200-01	15 November 2024	Non-Confidential	Beta release for issue B
0100-02	9 August 2024	Non-Confidential	First internal release
0100-01	13 December 2023	Non-Confidential	First internal for v1.0
0007-01	6 November 2023	Non-Confidential	First release for v0.7
0006-01	25 August 2023	Confidential	First internal release for v0.6
0005-01	24 April 2023	Confidential	First internal release for v0.5

Proprietary Notice

Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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 a Beta product, that is a product under development.

Feedback on content

Information about how to give feedback on the content.

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

- The title Arm® RME System Architecture Compliance Suite Validation Methodology.
- The number 108004_0200_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.

Inclusive language commitment

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

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

To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.


Note

This information is important and needs your attention.


Tip

This information might help you perform a task in an easier, better, or faster way.


Remember

This information reminds you of something important relating to the current content.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Realm Management Extension (RME) System Architecture	DEN0129	Non-Confidential
Arm® System Memory Management Unit Architecture Specification	IHI0070	Non-Confidential

Arm architecture and specifications	Document ID	Confidentiality
Arm® Architecture Reference Manual for A-profile architecture	DDI0487	Non-Confidential
Arm® Generic Interrupt Controller Architecture Specification for GIC architecture version 3.0 and version 4.0	IHI0069	Non-Confidential


Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com.

1.3 Other information

See the Arm® website for other relevant information.

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

2. Overview to RME System ACS

This chapter provides an introduction to the Arm® RME System Architecture Compliance Suite.

2.1 Abbreviations

The following table lists the abbreviations used in this document.

Table 2-1: Abbreviations and expansions

Abbreviation	Expansion
ACPI	Advanced Configuration and Power Interface
ACS	Architecture Compliance Suite
BDF	Bus, Device, and Function
DA	Device Assignment
DPT	Device Permission Table
ELx	Exception Level x (where x can be 0 to 3)
GIC	Generic Interrupt Controller
IDE	Integrity and Data Encryption
ITS	Interrupt Translation Service
IOMMU	Input-Output Memory Management Unit
KM	Key Management
LPI	Locality-specific Peripheral Interrupt
MPAM	Memory System Resource Partitioning and Monitoring
MSI	Message-Signaled Interrupt
PAL	Platform Abstraction Layer
PMU	Performance Monitoring Unit
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PSCI	Power State Coordination Interface
RAS	Reliability, Availability, and Serviceability
RCiEP	Root Complex integrated End Point
RME	Realm Management Extension
RMSD	Realm Management Security Domain
RMM	Realm Management Monitor
SMC	Secure Monitor Call
SMMU	System Memory Management Unit
SoC	System on Chip
TEE	Trusted Execution Environment
TDI	TEE Device Interface

Abbreviation	Expansion
TDISP	TEE Device Interface Security Protocol
UART	Universal Asynchronous Receiver and Transmitter
UEFI	Unified Extensible Firmware Interface
VAL	Validation Abstraction Layer

2.2 RME System ACS

Realm Management Extension Architecture is an extension to the Armv9-A profile architecture. It adds the following features:

- Two additional Security states, Root and Realm.
- Two additional physical address spaces, Root and Realm.
- The ability to dynamically transition memory granules between physical address spaces.
- Granule Protection Check mechanism.

With the other components of the Arm CCA, RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on Arm architecture.

The RME architecture defines the set of hardware features and properties that are required to comply with the Arm CCA architecture. Implementations compliant with the RME System architecture must conform to the behavior described in the Arm® Realm Management Extension (RME) System Architecture specification. The RME ACS is a comprehensive open source test suite which ensures that the implementations compliant with the RME System Architecture must comply with the behaviors described in the RME System Architecture specification for achieving confidential computing execution environment.

For more information on ACS coverage being incomplete, non-exhaustive and non-comprehensive, see the *RME System Architecture Compliance Suite Scenario Document*.

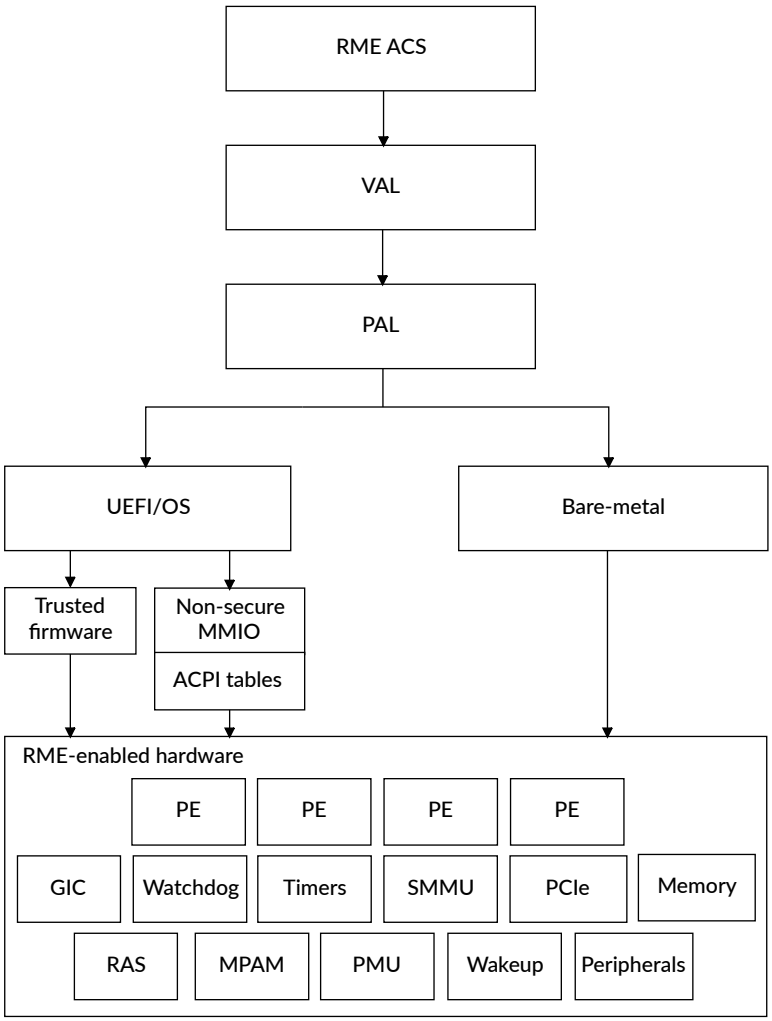
2.3 Layered software stack

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

The layered stack contains:

- Test suite
- Validation Abstraction Layer (VAL)
- Platform Abstraction Layer (PAL)

Figure 2-1: Layered software stack



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

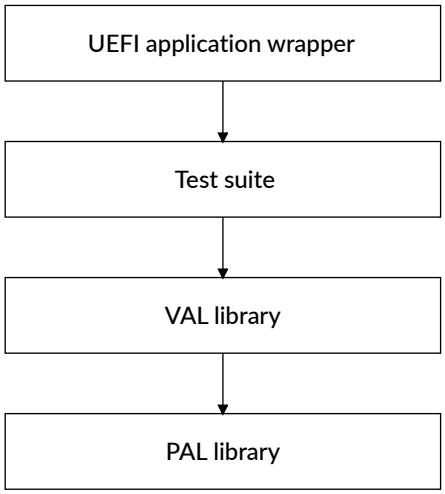
Table 2-2: Compliance test layers

Layer	Description
RME System ACS	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	Contains C-based Arm-defined APIs 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 the UEFI infrastructure and bare-metal abstraction.

2.3.1 Compliance test software stack with UEFI application

The following figure is an example of the compliance test software stack interplay with UEFI shell application.

Figure 2-2: Software stack UEFI shell application



2.3.2 Test and infrastructure guidelines

The coding guidelines followed for the implementation of the test suite are listed as follows:

- All the tests call VAL APIs.
- VAL APIs might call PAL APIs depending on the requested functionality.
- A test does not directly interface with PAL functions.
- The test layer does not need any code modifications when porting from one platform to another.
- All the platform porting changes are limited to PAL.
- The VAL may require changes if there are architectural changes impacting multiple platforms.

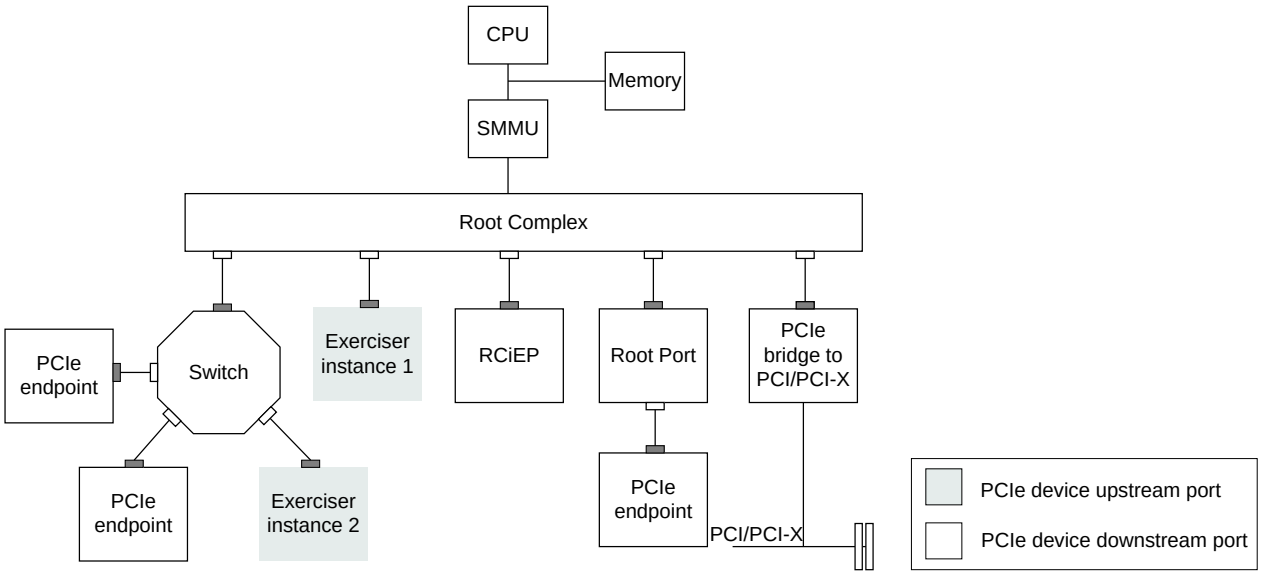
2.4 Exerciser

Exerciser is a PCIe endpoint device that can be programmed to generate custom stimuli for verifying the RME compliance of PCIe IP integration into an Arm SoC. The stimulus is used

in verifying the compliance of PCIe functionality like IO coherency, snoop behavior, address translation, PASID transactions, DMA transactions, MSI, and legacy interrupt behavior.

The following figure shows a PCIe hierarchy consisting of various endpoints, switches, and bridges.

Figure 2-3: Exerciser in an SoC



The figure shows two instances of the exerciser that are present in the system. Instance 1 is connected directly to the Root Complex as a RCiEP and instance 2 is connected to the downstream port of a switch as a PCIe endpoint device.

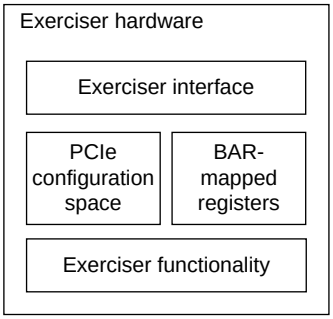
Note

The number of exercisers instantiated is platform-specific. To achieve higher coverage, Arm recommends that you present multiple exercisers to the ACS.

To generate custom stimuli, the exerciser must provide functionality to configure interrupt and DMA attributes, trigger them, and know the status of these operations, the details of which are **IMPLEMENTATION DEFINED**. This can be done by providing a set of BAR-mapped registers and writing specific values to trigger the necessary operations.

The following figure shows the reference implementation of exerciser hardware.

Figure 2-4: Reference implementation of exerciser hardware

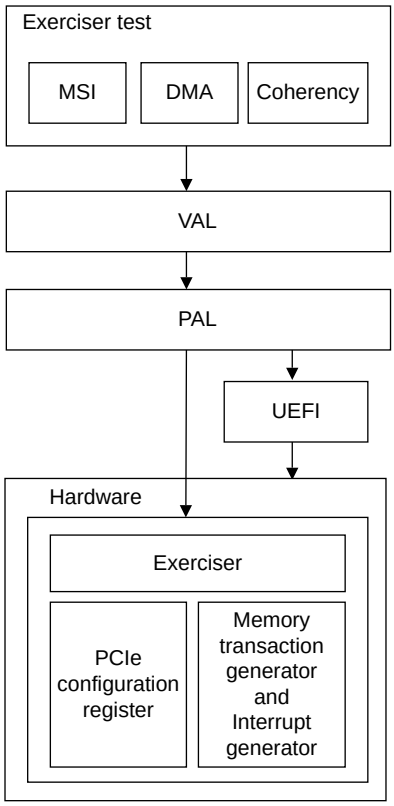


2.4.1 Compliance test software stack for exerciser with UEFI shell application

The exerciser validates PCIe devices that only access non-secure address space. The exerciser PCIe configuration space is accessed using UEFI or MMIO APIs and exerciser functionality like interrupt generation and DMA transactions can be accessed using exerciser APIs.

The following figure shows the compliance test software stack for exerciser with UEFI shell application.

Figure 2-5: Exerciser with UEFI shell application



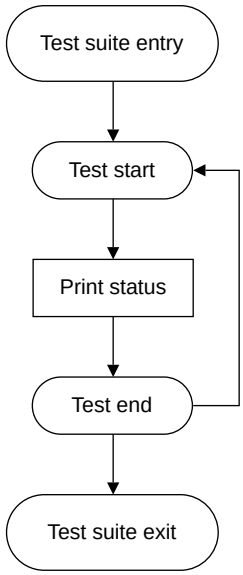
3. Execution flow control

This chapter describes the execution flow control used for RME System ACS.

3.1 Execution flow control

The following figure describes the execution flow control of the compliance suite.

Figure 3-1: Execution flow control



The process that is followed for the flow control is:

1. The execution environment such as the UEFI shell, invokes the test entry point.
2. Start the test iteration loop.
3. Print status during the test execution as required.
4. Reboot or put the system to sleep as required.
5. Loop until all the tests are completed.

3.2 Test build and execution flow

This section describes the source code directory structure and provides references for building the tests.

3.2.1 Source code directory

The following describes all the directories in the RME System ACS.

docs	Documentation.
baremetal_app	Reference bare-metal application source to call into the test entry point.
pal_el3	Contains EL3 code abstraction for val_el3.
pal_baremetal	Platform reference code for Bare-metal.
pal_uefi	Platform code targeting UEFI implementation.
test_pool	Test case source files for the test suite.
tools	Consists of scripts written for this suite and SysARCUI for partner input.
uefi_app	UEFI application source to call into the tests entry point.
val	Common code that is used by the tests. Makes calls to PAL as necessary.
val_el3	Common EL3 code that is used by the tests. Makes calls to pal_el3 as necessary.

Note

Inputs from the partners must be provided in the `sys_config.h` and `sys_config.c` files only.

3.2.2 Building the tests

The build steps for the compliance suite to be compiled as a UEFI shell application are available in the README.

3.3 EL3 ACS code integration with EL3 firmware

Install the software stack in which trusted firmware package is available. It is the reference implementation of secure world software for Arm A-Profile architectures, including an Exception Level 3 (EL3) Secure Monitor.

3.3.1 Prerequisites

- When smc with SMC_Fid is executed from Non-secure EL2, make sure you branch to the ACS User SMC handler function which is predefined in ACK.
- 4KB/16KB/64KB shared memory between EL3 and EL2 to share data to reflect the needs and requirements of the test clearly at both the levels.
- 2MB flat mapped memory used for MMU tables in EL3 and SP_EL3 by ACS User SMC handler function in EL3.
- 2MB free PA used only in the tests as PA.

- 512MB unused VA space (within 48bits) used in the tests as VA.
- 4KB of non-volatile memory used in the reset tests.


Note

See the val_el3/README.md for the ACS SMC handler function and SMC_FID.

3.3.2 EL3 structure and components

val_el3 folder contains the code that runs in EL3.

aarch64/

asm_helper_function.S - Contains ASM functions that help reading, loading, and storing contents to registers. Called by SmcHandlerAck.c and pgt_common.c.

plat_acs_smc_handler.c

Contains exception handler installing related functions, UserSmcCall function and access_mut function to access MUT.

pgt_common.c

Contains functions related to GPT mapping, MMU mapping at EL3.

ack_include.h

Contains structure defines that are used in the val_el3 files as well as function, and declarations, along with useful defines that are used in the other val_el3 files.

ack_common.c

Contains common el3 functions other than related to GPT, MMU mappings.

`pal_el3` contains acs_el3.c and acs_el3.h files that are responsible for programming NSEncryption, Legacy_TZ_EN and PAS_FILTER active mode and pal_el3.h is responsible for the val_el3 abstraction.

3.3.3 Building EL3 code

To generate binary file for EL3 code, follow the build steps in README of val_el3.

4. Platform Abstraction Layer

This chapter provides an overview of PAL API and its categories.

4.1 Overview of PAL API

The PAL is a C-based, Arm-defined API that you can implement.

Each test platform requires a PAL implementation of its own. The PAL APIs are meant for the compliance tests to reach or use other abstractions in the test platform such as the UEFI infrastructure and Linux OS modules. PAL implementation can also be bare-metal code.

The reference PAL implementations are available in the following directories:

- UEFI
- PAL_EL3


Note

The PAL bare-metal reference code provides a reference implementation for a subset of APIs. The current version of the repository contains the reference code for creation of information tables like PE, GIC, timer, and watchdog. Additional code must be implemented to match the target SoC implementation under test.

4.2 PAL API definitions

The PAL API contains APIs that:

- Are called by the VAL and implemented by the platform.
- Begin with the prefix `pal`.
- Have a second word on the API name that indicates the module which implements this API.
- Have the mapping of the module as per the table below.
- Create and fill structures needed as prerequisites for the test suite, named as `pal_<module>_create_info_table`.

4.2.1 API naming convention

The PAL API interface `<module>` names are mapped as shown in the following table.

Table 4-1: Modules and corresponding API names

Module	API name
DA	`da`

Module	API name
RME	`rme`
GIC	`gic`
SMMU	`smmu`
Legacy System	`legacy`

4.2.2 PE APIs

The following APIs provide the information and functionality required by the test suite that accesses features of a PE.

Table 4-2: PE APIs and their descriptions

API name	Function prototype	Description
get_num	`uint32_t pal_pe_get_num();`	Returns the number of PEs in the system.
create_info_table	`void pal_pe_create_info_table(PE_INFO_TABLE *PeTable);`	Gathers information about the PEs in the system and fills the info_table with the relevant data.
call_smc	`void pal_pe_call_smc(ARM_SMC_ARGS *args);`	Abstracts the smc instruction. The input arguments to this function are x0 to x7 registers filled in with the appropriate parameters.
execute_payload	`voidpal_pe_call_smc(ARM_SMC_ARGS *ArmSmcArgs, int32_t Conduit)`	Abstracts the PE wakeup and execute functionality. Ideally, this function calls the PSCI_ON SMC command.
update_elr	`void pal_pe_update_elr(void *context, uint64_t offset);`	Updates the ELR to return from exception handler to a required address.
get_esr	`uint64_t pal_pe_get_esr(void *context);`	Returns the exception syndrome from exception handler.
data_cache_ops_by_va	`void pal_pe_data_cache_ops_by_va(uint64_t addr, uint32_t type);`	Performs cache maintenance operation on an address.
get_far	`uint64_t pal_pe_get_far(void *context);`	Returns the FAR from exception handler.
install_esr	`uint32_t pal_pe_install_esr(uint32_t exception_type, void (*esr)(uint64_t, void *));`	Abstracts the exception handler installation steps. The input arguments are exception type and function pointer of the handler that has to be called when the exception of the given type occurs. It returns zero on success and non-zero on failure.
pal_enable_ns_encryption	`void pal_enable_ns_encryption(void)`	If NS Encryption is programmable then this API must enable NS_Encryption.
void pal_disable_ns_encryption	`void pal_disable_ns_encryption(void)`	If NS Encryption is programmable then this API must disable NS_Encryption.
pal_pas_filter_active_mode	`void pal_pas_filter_active_mode(int enable)`	This API is used to set or clear the active mode of PAS_FILTER in the system.
pal_prog_legacy_tz	`void pal_prog_legacy_tz(int enable)`	This API is used to program the LEGACY_TZ input for enabling/disabling it in the system.
pal_write_reset_status	`void pal_write_reset_status (uint64_t nvm_mem, uint32_t status)`	This API is used to write the reset status on Non-Volatile memory.

API name	Function prototype	Description
pal_read_reset_status	`uint32_t pal_read_reset_status (uint64_t nvm_mem)`	This API reads the reset status from Non-Volatile memory.
pal_save_global_test_data	`void pal_save_global_test_data (uint64_t nvm_mem, uint32_t total_tests, uint32_t tests_passed, uint32_t tests_failed)`	This API saves the test status like total tests, tests passed and tests failed before reset on NV memory.
pal_restore_global_test_data	`void pal_restore_global_test_data (uint64_t nvm_mem, uint32_t *total_tests, uint32_t *tests_passed, uint32_t *tests_failed)`	This API restores the tests status like total tests, tests passed and tests failed from NV memory after a system reset.

Note

Each PE information entry structure can hold information for a PE in the system. The types of information are:

```
typedef struct {
  UINT32   pe_num;                        /* PE Index */
  UINT32   attr;                          /* PE attributes */
  UINT64   mpidr;                         /* PE MPIDR */
  UINT32   pmu_gsiv;                      /* PMU Interrupt */
  UINT32   gmain_gsiv;                    /* GIC Maintenance Interrupt
 */
  UINT32   acpi_proc_uid;                 /* ACPI Processor UID */
  UINT32   level_1_res[MAX_L1_CACHE_RES]; /* index of level 1 cache(s)
 in cache_info_table */
}PE_INFO_ENTRY;
```

4.2.3 GIC APIs

These APIs provide the information and functionality required by the test suite that accesses features of a GIC.

Table 4-3: GIC APIs and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_gic_create_info_table(GIC_INFO_TABLE *gic_info_table);`	Gathers information about the GIC sub-system and fills the gic_info_table with the relevant data.
install_isr	`uint32_t pal_gic_install_isr(uint32_t int_id, void (*isr)(void));`	Abstracts the steps required to register an interrupt handler to an IRQ number. It also enables the interrupt in the GIC CPU interface and Distributor. It returns 0 on success and -1 on failure.
end_of_interrupt	`uint32_t pal_gic_end_of_interrupt(uint32_t int_id);`	Indicates completion of interrupt processing by writing to the end of interrupt register in the GIC CPU interface. It returns 0 on success and -1 on failure.

API name	Function prototype	Description
request_irq	`uint32_t pal_gic_request_irq(unsigned int irq_num, unsigned int mapped_irq_num, void *isr);`	Registers the interrupt handler for a given IRQ. `irq_num`: hardware IRQ number `mapped_irq_num`: mapped IRQ number `isr`: Interrupt Service Routine that returns the status
free_irq	`void pal_gic_free_irq(unsigned int irq_num, unsigned int mapped_irq_num);`	Frees the registered interrupt handler for a given IRQ. `irq_num`: hardware IRQ number `mapped_irq_num`: mapped IRQ number
set_intr_trigger	`uint32_t pal_gic_set_intr_trigger (uint32_t int_id, INTR_TRIGGER_INFO_TYPE_e trigger_type);`	Sets the trigger type to edge or level. `int_id`: interrupt ID which must be enabled and the service routine installed for `trigger_type`: interrupt trigger type edge or level

Note

- Each GIC information entry structure can hold information for any of the seven types of GIC components. The seven types of entries are:

```
typedef enum {
  ENTRY_TYPE_CPUIF = 0x1000,
  ENTRY_TYPE_GICD,
  ENTRY_TYPE_GICC_GICRD,
  ENTRY_TYPE_GICR_GICRD,
  ENTRY_TYPE_GICITS,
  ENTRY_TYPE_GIC_MSI_FRAME,
  ENTRY_TYPE_GICH
}GIC_INFO_TYPE_e;
```

- In addition to the type, each entry contains the base address of each type, entry_id for entry type ITS, and length in case of Redistributor range address length.

```
typedef struct {
  UINT32 type;
  UINT64 base;
  UINT32 entry_id;
  UINT64 length;
  UINT32 flags;
  UINT32 spi_count;
  UINT32 spi_base;
}GIC_INFO_ENTRY;
```

4.2.4 PCIe APIs

These APIs provide the information and functionality required by the test suite that accesses features of PCIe subsystem.

Table 4-4: PCIe APIs and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_pcie_create_info_table(PCIE_INFO_TABLE *PcieTable);`	Abstracts the steps to gather PCIe information in the system and fills the PCIe info_table. Ideally, this function reads the ACPI MCFG table to retrieve the ECAM base address.
enumerate	`void pal_pcie_enumerate(void);`	Performs the PCIe enumeration.
io_read_cfg	`uint32_t pal_pcie_io_read_cfg(uint32_t bdf, uint32_t offset, uint32_t *data);`	Abstracts the configuration space read of a device identified by Bus, Device, and Function (BDF). This is used only in peripheral tests and need not be implemented in Linux. It returns either success or failure. `bdf`: PCI Bus, Dev, and Func `offset`: Offset in the configuration space from where data is to be read `data`: Stores the value read from the configuration space
io_write_cfg	`void pal_pcie_io_write_cfg(uint32 bdf, uint32 offset, uint32 data);`	Abstracts the configuration space write of a device identified by BDF (Bus, Device, and Function). Writes 32-bit data to the configuration space of the device at an offset. `bdf`: PCI Bus, Dev, and Func `offset`: Offset in the configuration space from where data is to be read `data`: Stores the value read from the configuration space
get_mcfg_ecam	`uint64_t pal_pcie_get_mcfg_ecam();`	Returns the PCI ECAM address from the ACPI MCFG table address.

API name	Function prototype	Description
get_msi_vectors	`uint32_t pal_get_msi_vectors(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn, PERIPHERAL_VECTOR_LIST **mvector);`	Creates a list of MSI(X) vectors for a device. It returns the number of MSI(X) vectors. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number `mvector`: Pointer to MSI(X) address
scan_bridge_devices_and_check_memtype	`uint32_t pal_pcie_scan_bridge_devices_and_check_memtype (uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Scans the bridge devices and checks the memory type. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number
get_pcie_type	`uint32_t pal_pcie_get_pcie_type(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Gets the PCIe device or port type. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number
p2p_support	`uint32_t pal_pcie_p2p_support();`	Checks P2P support in the PCIe hierarchy. Returns 1 if P2P feature is not supported and 0 if it is supported.
dev_p2p_support	`uint32_t pal_pcie_dev_p2p_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Checks the PCIe device P2P support. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number Returns 1 if P2P feature is not supported, else 0.

API name	Function prototype	Description
is_cache_present	`uint32_t pal_pcie_is_cache_present (uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Checks whether the PCIe device has an *Address Translation Cache* (ATC). `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number Returns 0 if the device does not have ATC, else 1.
is_onchip_peripheral	`uint32_t pal_pcie_is_onchip_ peripheral(uint32_t bdf);`	Checks if a PCIe function is an on-chip peripheral. `bdf`: Segment, PCI Bus, Device, and Function. Returns 1 if the PCIe function is an on-chip peripheral, else 0.
check_device_list	`uint32_t pal_pcie_ check_device_list(void);`	Checks if the PCIe hierarchy matches with the topology described in the information table. Returns 0 if device entries match, else 1.
check_device_valid	`uint32_t pal_pcie_check_device_ valid(uint32_t bdf);`	This API is used as a placeholder to check if the bdf obtained is valid or not. `bdf`: PCI Seg, bus, device, and function
get_rp_transaction_frwd_support	`uint32_t pal_pcie_get_rp_transaction_ frwd_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Gets Root Port (RP) transaction forwarding support. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number Returns 0 if RP is not involved in transaction forwarding, else 1.

API name	Function prototype	Description
read_ext_cap_word	`void pal_pcie_read_ext_cap_word(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn, uint32_t ext_cap_id, uint8_t offset, uint16_t *val);`	Reads the extended PCIe configuration space at an offset for a capability. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number `ext_cap_id`: PCI capability ID `offset`: offset of the word in the capability configuration space `val`: return value

get_bdf_wrapper	`uint32 pal_pcie_get_bdf_wrapper (uint32 ClassCode, uint32 StartBdf);`	Returns the Bus, Device, and Function for a matching class code.
		
		`ClassCode`: 32-bit value of format `ClassCode << 16 | sub_class_code`
		
		`StartBdf:`
		
		`0`: start enumeration from host bridge.
		
		`1`: start enumeration from the input segment, Bus, Device.
		
		This is needed since multiple controllers with the same class code are potentially present in a system.

bdf_to_dev	`void *pal_pci_bdf_to_dev(uint32_t bdf);`	Returns the PCI device structure for the given bdf. `bdf`: PCI Bus, Device, and Function.
read_config_byte	`void pal_pci_read_config_byte(uint32_t bdf, uint8_t offset, uint8_t *val);`	Reads one byte from the PCI configuration space for the current BDF at given offset. `bdf`: PCI Bus, Device, and Function `offset`: offset in the PCI configuration space for that BDF `val`: return value

API name	Function prototype	Description
write_config_byte	`void pal_pci_write_config_byte(uint32_t bdf, uint8_t offset, uint8_t val);`	Writes one byte from the PCI configuration space for the current BDF at a given offset. `bdf`: PCI Bus, Device, and Function `offset`: offset in the PCI configuration space for that BDF `val`: return value
mem_get_offset	`uint32_t pal_pcie_mem_get_offset(uint32_t type);`	Returns the memory offset that can be accessed from the BAR base. `type`: Size of the offset required

This data structure holds the PCIe subsystem information.

```
/**
@brief PCI Express Info Table
**/
typedef struct {
addr_t ecam_base;       ///< ECAM Base address
uint32_t segment_num;   ///< Segment number of this ECAM
uint32_t start_bus_num; ///< Start Bus number for this ecam space
uint32_t end_bus_num;   ///< Last Bus number
}PCIE_INFO_BLOCK;
```

Note

The data structure is repeated for the number of ECAM ranges in the system.

```
typedef struct {
uint32_t num_entries;
PCIE_INFO_BLOCK block[];
}PCIE_INFO_TABLE;
```

4.2.5 SMMU APIs

These functions abstract information that is specific to the operations of the SMMUs in the system.

Table 4-5: SMMU APIs and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_smmu_create_info_table(SMMU_INFO_TABLE *smmu_info_table);`	Abstracts the steps to gather information about SMMUs in the system and fills the info_table.

API name	Function prototype	Description
check_device_iova	`uint32_t pal_smmu_check_device_iova(void *port, uint64_t dma_addr);`	Checks if the input DMA address belongs to the input device. This can be done by keeping track of the DMA addresses generated by the device using the start and stop monitor calls defined below or by reading the IOVA table of the device and looking for the input address. 0 is returned if address belongs to the device. Non-zero is returned if there are **IMPLEMENTATION DEFINED** error values.
device_start_monitor_iova	`void pal_smmu_device_start_monitor_iova(void *port);`	A hook to start the process of saving DMA addresses being used by the input device. It is used by the test to indicate the upcoming DMA transfers to be recorded and the test queries for the address through the check_device_iova call.
device_stop_monitor_iova	`void pal_smmu_device_stop_monitor_iova(void *port);`	Stops the recording of the DMA addresses being used by the input port.
max_pasids	`uint32_t pal_smmu_max_pasids(uint64_t smmu_base);`	Returns the maximum PASID value supported by the SMMU controller. For SMMUv3, this value can be read from the IDR1 register. 0 is returned when PASID support is not detected. Non-zero is returned if maximum PASID value supported for the input SMMU.
pa2iova	`uint64_t pal_smmu_pa2iova(uint64_t SmmuBase, uint64_t Pa);`	Converts physical address to I/O virtual address. `SmmuBase`: physical address of the SMMU for conversion to virtual address. `Pa`: physical address to use in conversion. Returns 0 on success and 1 on failure.
smmu_disable	`uint32_t pal_smmu_disable(uint64_t SmmuBase);`	Globally disables the SMMU based on input base address. `SmmuBase`: physical address of the SMMU that needs to be globally disabled. Returns 0 for success and 1 for failure.
create_pasid_entry	`uint32_t pal_smmu_create_pasid_entry(uint64_t smmu_base, uint32_t pasid);`	Prepares the SMMU page tables to support input PASID. `smmu_base`: physical address of the SMMU for which PASID support is needed. `pasid`: Process Address Space IDentifier. Returns 0 for success and 1 for failure.

4.2.6 Exerciser APIs

These APIs abstract information specific to the operations of PCIe stimulus generation hardware.

Table 4-6: Exerciser APIs and descriptions

API Name	Function prototype	Description
set_param	`uint32_t pal_exerciser_set_param(EXERCISER_PARAM_TYPE type, uint64_t value1, uint64_t value2, uint32_t instance)`	Writes the configuration parameters to the PCIe stimulus generation hardware indicated by the instance number. The supported configuration parameters include: 1 – SNOOP_ATTRIBUTES 2 – LEGACY_IRQ 3 – DMA_ATTRIBUTES 4 – P2P_ATTRIBUTES 5 – PASID_ATTRIBUTES 6 – MSIX_ATTRIBUTES 7 – CFG_TXN_ATTRIBUTES 8 – ERROR_INJECT_TYPE `value2` is an optional argument and must be ignored for some configuration parameters.
get_param	`uint32_t pal_exerciser_get_param(EXERCISER_PARAM_TYPE type, uint64_t *value1, uint64_t *value2, uint32_t instance)`	Returns the requested configuration parameter values through 64-bit input arguments `value1` and `value2`. The function returns a value of 1 to indicate read success and 0 to indicate read failure.
set_state	`uint32_t pal_exerciser_set_state(EXERCISER_STATE state, uint64_t *value, uint32_t instance)`	Sets the state of the PCIe stimulus generation hardware. The supported states include: 1 – RESET, hardware in reset state. 2 – ON, this state is set after hardware is initialized and is ready to generate stimulus. 3 – OFF, this state is set to indicate that hardware can no longer generate stimulus. 4 – ERROR, this state is set to signal an error with hardware.
get_state	`uint32_t pal_exerciser_get_state(EXERCISER_STATE state, uint64_t *value, uint32_t instance)`	Returns the state of the PCIe stimulus generation hardware of the requested instance.

API Name	Function prototype	Description
ops	`uint32_t pal_exerciser_ops(EXERCISER_OPS ops, uint64_t param, uint32_t instance)`	Abstracts the steps to implement the requested operation on the PCIe stimulus generation hardware. Following are the supported operations: 1 – START_DMA 2 – GENERATE_MSI 3 – GENERATE_L_INTR 4 – MEM_READ 5 – MEM_WRITE 6 – CLEAR_INTR 7 – PASID_TLP_START 8 – PASID_TLP_STOP 9 – TXN_NO_SNOOP_ENABLE 10 – TXN_NO_SNOOP_DISABLE 11 – START_TXN_MONITOR 12 – STOP_TXN_MONITOR 13 – ATS_TXN_REQ 14 – INJECT_ERROR
get_data	`uint32_t pal_exerciser_get_data(EXERCISER_DATA_TYPE type, exerciser_data_t *data, uint32_t instance)`	Returns either the configuration space or the BAR space information depending on the input argument type. The argument type can take one of the following two values: 1 – EXERCISER_DATA_CFG_SPACE 2 – EXERCISER_DATA_BAR0_SPACE
is_bdf_exerciser	`uint32_t pal_is_bdf_exerciser(uint32_t bdf)`	Checks if the device is an exerciser. Returns 1 if device is an exerciser, else 0.
get_ecsr_base	`uint64_t pal_exerciser_get_ecsr_base(uint32_t Bdf,uint32_t BarIndex)`	Returns the ECSR base address of a particular BAR Index.
get_pcie_config_offset	`uint64_t pal_exerciser_get_pcie_config_offset(uint32_t Bdf)`	Returns the configuration address of the given bdf.
start_dma_direction	`uint32_t pal_exerciser_start_dma_direction(uint64_t Base, EXERCISER_DMA_ATTRDirection)`	Triggers the DMA operation.
find_pcie_capability	`uint32_t pal_exerciser_find_pcie_capability(uint32_t ID, uint32_t Bdf, uint32_t Value, uint32_t *Offset)`	Returns 0 if the PCI capability is found.
device_lock	`uint32_t pal_device_lock(uint32_t bdf)`	Transitions the device TDI state to RUN.

API Name	Function prototype	Description
device_unlock	`uint32_t pal_device_unlock(uint32_t bdf)`	Transitions the TDI from RUN to CONFIG_UNLOCKED state.

4.2.7 Miscellaneous APIs

Miscellaneous APIs are described in the following table.

Table 4-7: Miscellaneous APIs and their descriptions

API name	Function prototype	Description
print	`void pal_print(char *string, uint64_t data);`	Sends a formatted string to the output console. `string`: An ASCII string. `data`: Data for the formatted output.
print_raw	`void pal_print_raw(uint64_t addr, char *string, uint64_t data);`	Sends a string to the output console without using the platform print function. This function gets COMM port address and directly writes to the address character by character. `addr`: Address to be written. `string`: An ASCII string. `data`: Data for the formatted output.
strncmp	`pal_strncmp uint32_t pal_strncmp (char *FirstString, char *SecondString, uint32_t Length);`	Compares two strings. Returns zero if strings are identical, else a nonzero value. `FirstString`: The pointer to the first null-terminated ASCII string. `SecondString`: The pointer to the second null-terminated ASCII string. `Length`The maximum number of ASCII characters for comparison.
mmio_read	`uint32 pal_mmio_read(uint64 addr);`	Provides a single point of abstraction to read from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 32-bit data read from the input address
mmio_read8	`pal_mmio_read8(uint64 addr);`	Provides a single point of abstraction to read 8-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 8-bit data read from the input address

API name	Function prototype	Description
mmio_read16	`pal_mmio_read16(uint64 addr);`	Provides a single point of abstraction to read 16-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 16-bit data read from the input address
mmio_read64	`pal_mmio_read64(uint64 addr);`	Provides a single point of abstraction to read 64-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 64-bit data read from the input address
mmio_write	`void pal_mmio_write(unit64 addr,uint32 data);`	Provides a single point of abstraction to write to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 32-bit data to write to address
mmio_write8	`pal_mmio_write8(unit64 addr,uint8 data);`	Provides a single point of abstraction to write 8-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 8-bit data to write to address
mmio_write16	`pal_mmio_write16(unit64 addr,uint16 data);`	Provides a single point of abstraction to write 16-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 16-bit data to write to address
mmio_write64	`pal_mmio_write(unit64 addr,uint64 data);`	Provides a single point of abstraction to write 64-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 64-bit data to write to address
mem_free_shared	`pal_mem_free_shared(void);`	Frees the allocated shared memory region.
mem_get_shared_addr	`pal_mem_get_shared_addr(void);`	Returns the base address of the shared memory region to the VAL layer.
mem_alloc	`void pal_mem_alloc(unsigned int size);`	Allocates memory of the requested size. `size`: size of the memory region to be allocated Returns virtual address on success and null on failure.
mem_calloc	`void * pal_mem_calloc(uint32_t num, uint32_t Size);`	Allocates requested buffer size in bytes with zeros in a contiguous memory and returns the base address of the range.

API name	Function prototype	Description
mem_allocate_shared	`pal_mem_allocate_shared (uint32_t num_pe, uint32_t sizeofentry);`	Allocates memory which is to be used to share data across PEs. `num_pe`: number of PEs in the system `sizeofentry`: size of memory region allocated to each PE Returns none.
mem_free	`void pal_mem_free(void *buffer);`	Frees the memory allocated by UEFI framework APIs. `buffer`: the base address of the memory range to be free
mem_cpy	`void *pal_memcpy(void *dest_buffer, void *src_buffer, uint32_t len);: base address of the memory range to be freed`	Copies a source buffer to a destination buffer and returns the destination buffer. `dest_buffer`: pointer to the destination buffer of the memory copy `src_buffer`: pointer to the source buffer of the memory copy len: number of bytes to copy from source buffer to destination buffer Returns the destination buffer.
mem_compare	`uint32 pal_mem_compare(void *src, void *dest, uint32 len);`	Compares the contents of the source and destination buffers. `src`: base address of the memory, source buffer to be compared `dest`: destination buffer to be compared with `len`: length of the comparison to be performed
mem_alloc_cacheable	`void pal_mem_alloc_cacheable(uint32_t bdf, uint32_t size, void *pa);`	Allocates cacheable memory of the requested size. `bdf`: BDF of the requesting PCIe device `size`: size of the memory region to be allocated `pa`: physical address of the allocated memory
mem_free_cacheable	`void pal_mem_free_cacheable(uint32_t bdf, uint32_t size, void *va, void *pa);`	Frees the cacheable memory allocated by Linux DMA Framework APIs. `bdf`: Bus, Device, and Function of the requesting PCIe device `size`: size of memory region to be freed `va`: virtual address of the memory to be freed `pa`: physical address of the memory to be freed

API name	Function prototype	Description
mem_virt_to_phys	`void pal_mem_virt_to_phys(void *va);`	Returns the physical address of the input virtual address. `va`: virtual address of the memory to be converted Returns the physical address.
time_delay_ms	`uint64 pal_time_delay_ms (uint64 MicroSeconds);`	Stalls the CPU for the specified number of microseconds. `MicroSeconds`: the minimum number of microseconds to be delayed Returns the value of the microseconds given as input.
mem_set	`void pal_mem_set (void *buf, uint32 size, uint8 value);`	A buffer with a known specified input value. `buf`: pointer to the buffer to fill `size`: number of bytes in the buffer to fill `value`: value to fill the buffer with
page_size	`uint32_t pal_mem_page_size();`	Returns the memory page size (in bytes) used by the platform.
alloc_pages	`void* pal_mem_alloc_pages (uint32 NumPages);`	Allocates the requested number of memory pages.
free_pages	`void pal_mem_free_pages (void *PageBase, uint32_t NumPages);`	Frees pages as requested.
phys_to_virt	`void* pal_mem_phys_to_virt (uint64_t Pa);`	Returns the VA of the input PA. `Pa`: Physical Address of the memory to be converted. Returns the VA.
target_is_bm	`uint32_t pal_target_is_bm();`	Checks if the system information is passed using bare-metal.
aligned_alloc	`void *pal_aligned_alloc( uint32_t alignment, uint32_t size);`	Allocates memory with the given alignment. `alignment`: Specifies the alignment. `size`: Requested memory allocation size. Returns pointer to the allocated memory with requested alignment.
mem_alloc_at_address	`void *pal_mem_alloc_at_address(uint64_t mem_base, uint64_t size);`	Allocates memory in the given memory base.
mem_free_at_address	`void pal_mem_free_at_address(uint64_t mem_base, uint64_t size);`	Frees the allocated memory in the given memory base.

Appendix A Revisions

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

A.1 Revisions

The following tables describe the changes between different issues of this document.

Table A-1: Issue 0005-01

Change	Location
First release.	-

Table A-2: Differences between Issue 0005-01 and Issue 0006-01

Change	Location
Added new PE APIs.	See 4.2.2 PE APIs on page 21.

Table A-3: Differences between Issue 0006-01 Issue 0007-01

Change	Location
Added new module and API.	See 4.2.1 API naming convention on page 20.
Added new PE APIs.	See 4.2.2 PE APIs on page 21.

Table A-4: Differences between Issue 0006-01 and Issue 0007-01

Change	Location
Added new module and API.	See 4.2.1 API naming convention on page 20.
Added new PE APIs.	See 4.2.2 PE APIs on page 21.

Table A-5: Differences between Issue 0007-01 and Issue 0100-01

Change	Location
Updated the prerequisites and added a note.	See 3.3.1 Prerequisites on page 18.
Updated the EL3 code.	See 3.3.2 EL3 structure and components on page 19.

Table A-6: Differences between Issue 0100-01 and Issue 0100-02

Change	Location
Added new abbreviations.	See 2.1 Abbreviations on page 10.
Updated the description.	See 2.2 RME System ACS on page 11 .
Removed the sections compliance test and test platform abstraction.	See 2. Overview to RME System ACS on page 10.
Updated the section coding guidelines to test and infrastructure guidelines.	See 2.3.2 Test and infrastructure guidelines on page 13.
Updated the system directory structure.	See 3.2.1 Source code directory on page 17.
Added a module, DA.	See 3.2.1 Source code directory on page 17.

Table A-7: Differences between Issue 0100-02 and Issue 0200-01

Change	Location
Added new abbreviation.	See 2.1 Abbreviations on page 10.
Added new APIs in the exerciser APIs and descriptions table.	See 4.2.6 Exerciser APIs on page 29.
Removed the section GIC ITS.	-
Removed the RME system directory structure image.	See 3.2.1 Source code directory on page 17.