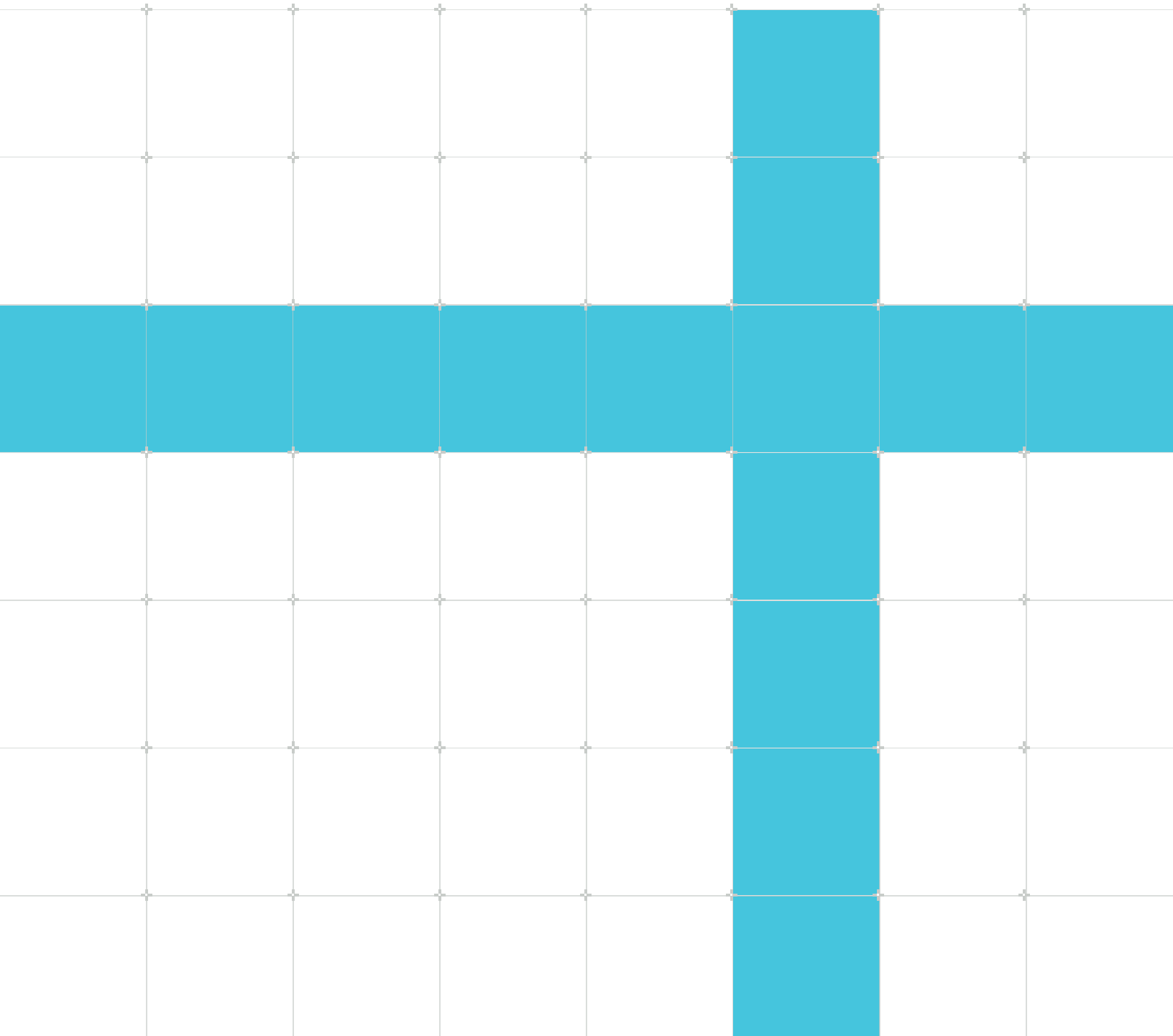
# Arm® RME System Architecture Compliance Suite

Version 0.7

# Validation Methodology

**Non-Confidential**


**Issue 01**
108004_0007_01_en



# Arm® RME System Architecture Compliance Suite
## Validation Methodology



## Release Information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0005-01 | 24 April 2023 | Confidential | First internal release for v0.5 |
| 0006-01 | 25 August 2023 | Confidential | First internal release for v0.6 |
| 0007-01 | 6 November 2023 | Non-Confidential | First release for v0.7 |

## Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## 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-systemready-acs@arm.com. Give:

- The title Arm® RME System Architecture Compliance Suite Validation Methodology.
- The number 108004_0007_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.


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.<br><br>For example:<br><br>`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**. |


Recommendations. Not following these recommendations might lead to system failure or damage.


Requirements for the system. Not following these requirements might result in system failure or damage.


Requirements for the system. Not following these requirements will result in system failure or damage.


An important piece of information that needs your attention.


A useful tip that might make it easier, better or faster to perform a task.


A reminder of something important that relates to the information you are reading.

## 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* | DEN0129H | 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 |


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 |
| AEST | Arm Error Source Table |
| BDF | Bus, Device, and Function |
| ELx | Exception Level x (where x can be 0 to 3) |
| GCD | Grand Central Dispatch |
| GIC | Generic Interrupt Controller |
| HMAT | Heterogenous Memory Attribute Table |
| HVC | HyperVisor Call |
| ITS | Interrupt Translation Service |
| IOMMU | Input-Output Memory Management Unit |
| 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 |
| PPTT | Processor Properties Topology Table |
| PSCI | Power State Coordination Interface |
| RAS | Reliability, Availability, and Serviceability |
| RCiEP | Root Complex integrated End Point |
| SATA | Serial Advanced Technology Attachment |
| RME | Realm Management Extension |
| SMC | Secure Monitor Call |
| SMMU | System Memory Management Unit |
| SoC | System on Chip |
| SRAT | System Resource Affinity Table |
| STS | Statistical Test Suite |

| Abbreviation | Expansion |
|---|---|
| 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 Architecture Compliance Suite (ACS) is a set of examples of the specified invariant behaviors. The compliance suite verifies that these behaviors are correctly implemented on a system.

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

## 2.3 Compliance tests

RME compliance tests are self-checking, portable C-based tests with directed stimulus.

The following table describes the compliance test components.

**Table 2-2: Compliance test components**

| Component | Description |
|---|---|
| Exerciser | Verifies PCIe subsystem with a custom stimulus generator. |
| GIC | Verifies GIC compliance. |
| Memory | Verifies memory map compliance. |
| PE | Verifies PE compliance. |
| SMMU | Verifies SMMU subsystem compliance. |
| Watchdog | Verifies watchdog timer compliance. |

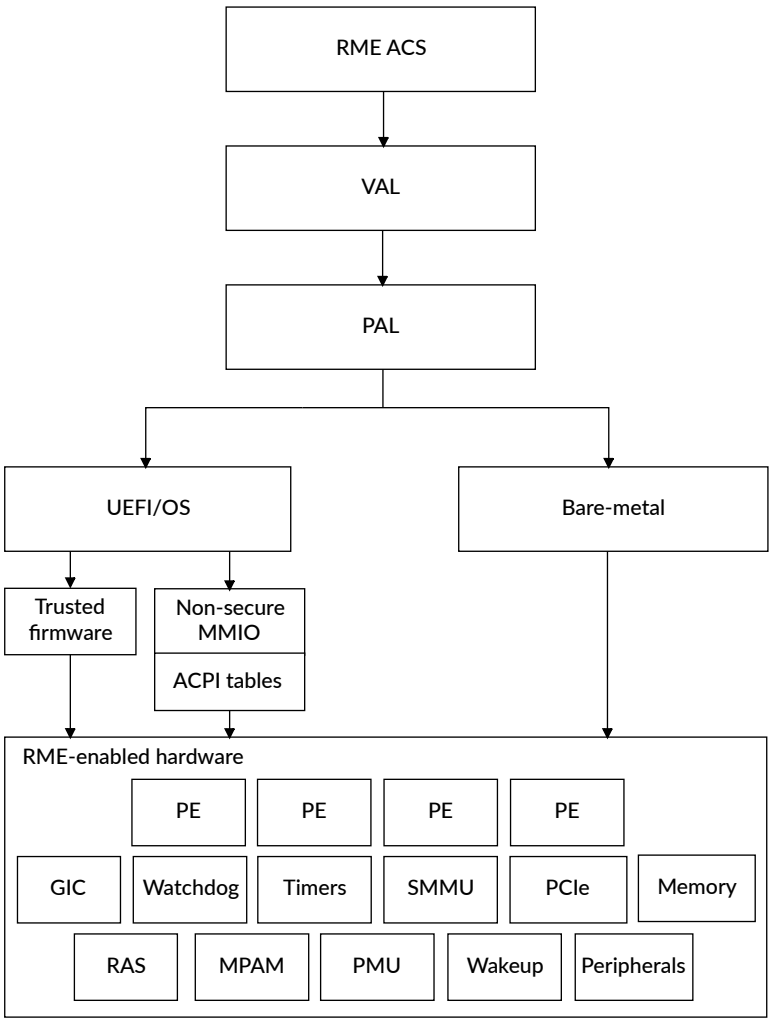
## 2.4 Layered software stack

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.

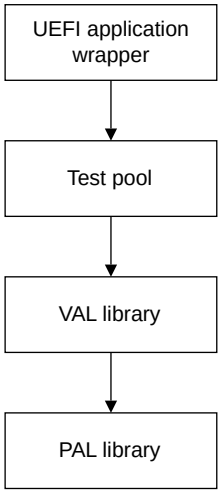
**Table 2-3: Compliance test layers**

| 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. |

### 2.4.1 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**




### 2.4.2 Coding guidelines

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

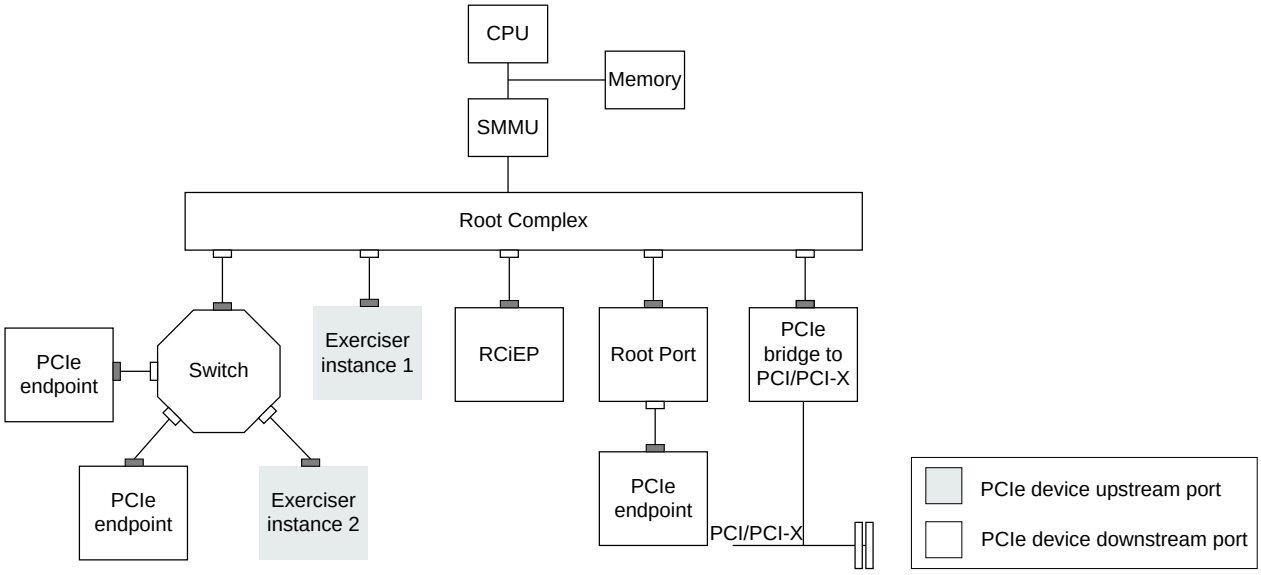
- All the tests call VAL APIs.
- VAL APIs might call PAL APIs depending on the requested functionality.
- A test does not directly interface with PAL functions.
- The test layer does not need any code modifications when porting from one platform to another.
- All the platform porting changes are limited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

## 2.5 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

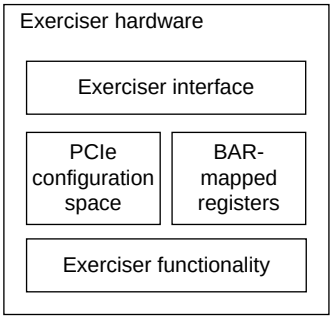
The number of exercisers instantiated is platform-specific. To achieve higher coverage, Arm recommends that you present multiple exercisers to the ACS.

---

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**

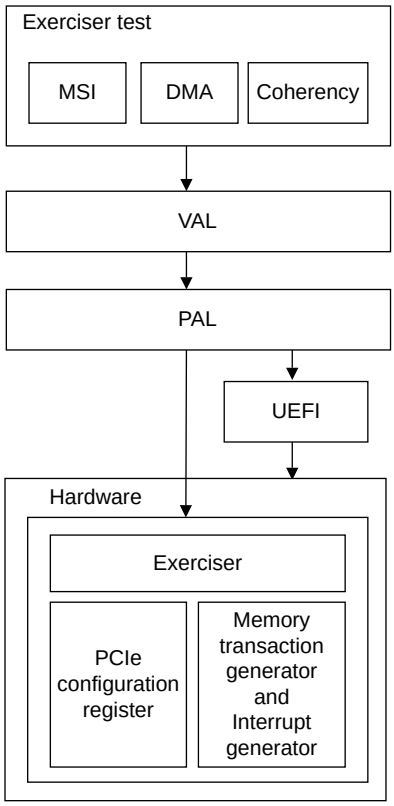



### 2.5.1 Compliance test software stack for exerciser with UEFI shell application

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**




## 2.6 GIC ITS

The Interrupt Translation Service (ITS) translates an input EventID from a device, identified by its DeviceID and determines:

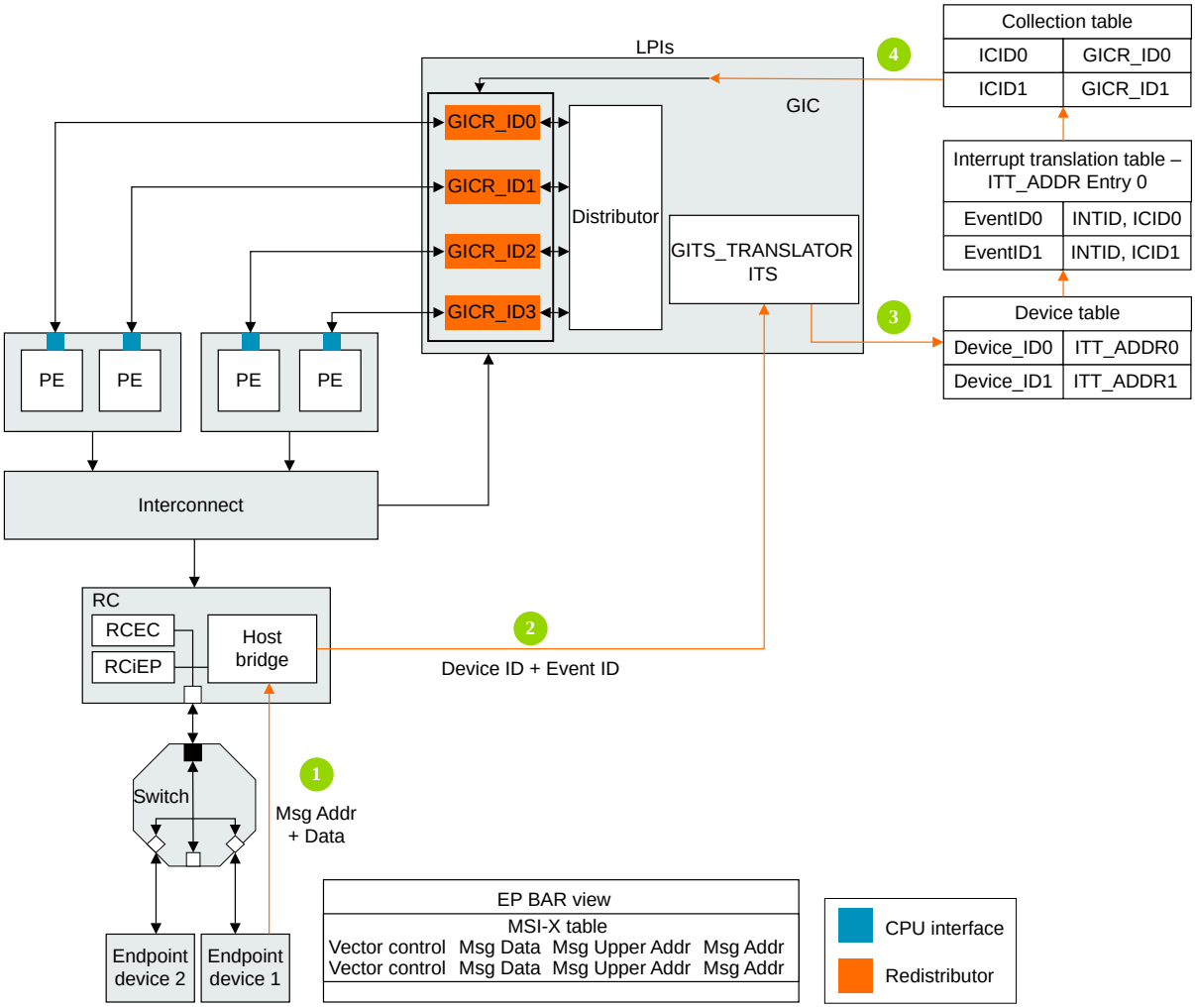
- The corresponding INTID for the input.
- The target Redistributor and, through this, the target PE for the INTID.

Endpoint device 1 triggers a write on MSI address from the MSI table, which gets converted to a Locality-specific Peripheral Interrupt (LPI) using the ITS tables. To generate an MSI, ITS must be configured before running the ACS. The software must allocate memory for different ITS tables. ITS table mappings must be updated using the ITS commands, Device ID, LPI Interrupt ID, and Redistributor Base.

For more information on GIC ITS, see *Arm® GIC Architecture Specification* and *Arm® GICv3 Software Overview*.

The following figure shows how an MSI is converted to an LPI using ITS.

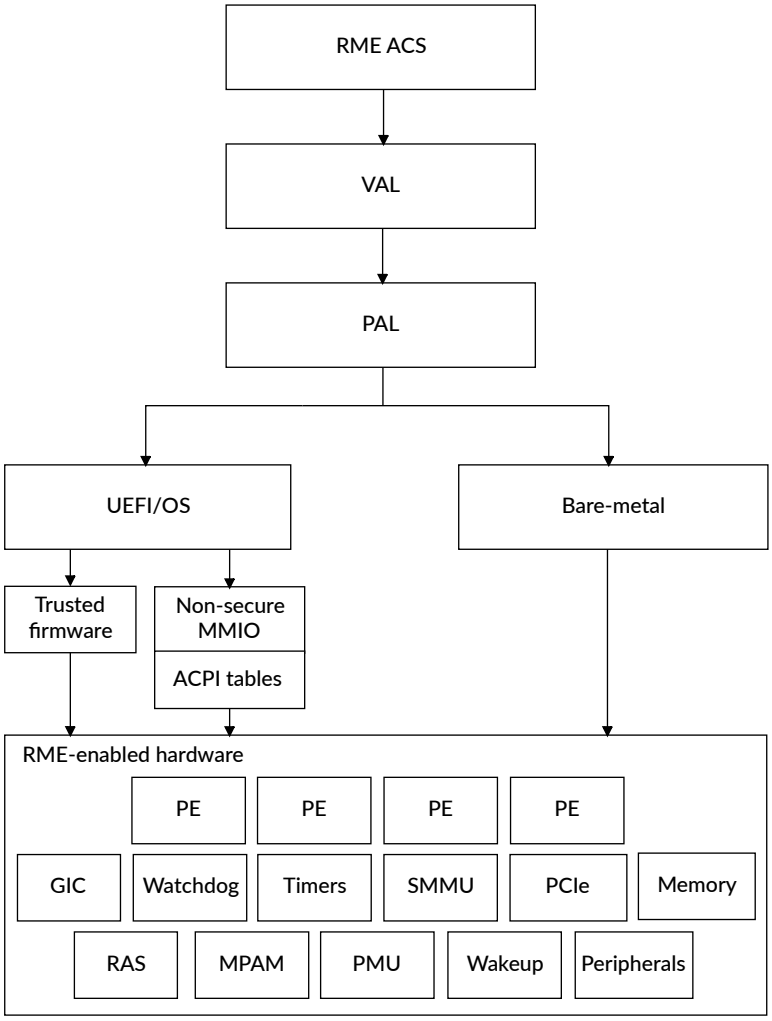
**Figure 2-6: Routing MSI-X from Endpoint to PE through GIC ITS**



## 2.7 Test platform abstraction

The compliance suite defines and uses the test platform abstraction as illustrated in the figure below.

**Figure 2-7: Test platform abstraction**




The following table describes the RME abstraction terms.

**Table 2-4: Abstraction terms and descriptions**

| Abstraction | Description |
|---|---|
| UEFI | UEFI Shell application or operating system provides infrastructure for console and memory management. This module runs at EL2. |
| Trusted firmware | Firmware which runs at EL3. |

| Abstraction | Description |
|---|---|
| ACPI | Interface layer which provides platform-specific information, removing the need for the test suite to be ported for every platform. |
| Hardware | PE and controllers that are specified as part of the RME System specification. |

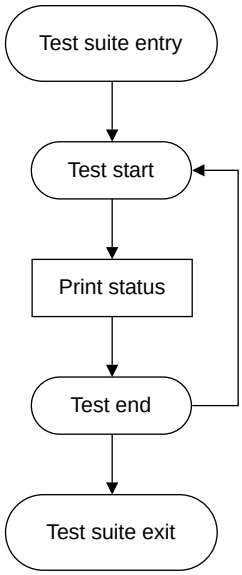
# 3. Execution flow control

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.

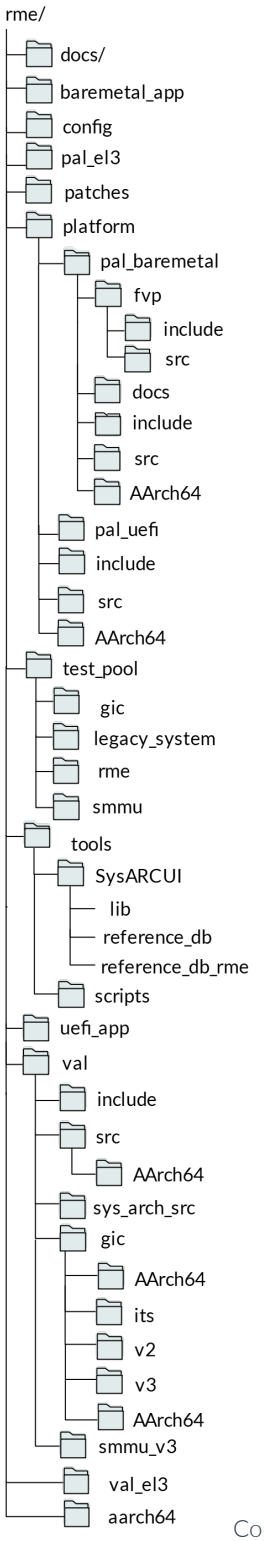
## 3.2 Test build and execution flow

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

### 3.2.1 Source code directory

The following figure shows the source code directory for the RME System ACS.

**Figure 3-2: RME System directory structure**



The following describes all the directories in 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

- Branch to the `UserSmcCall` function which is predefined in ACK when smc with imm = 0x100 is executed from Non-secure EL2.
- 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 `UserSmcCall` 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-nolatile memory used in the reset tests.

### 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.

**SmcHandlerAck.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


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 |
|---|---|
| RME | `rme` |

| Module | API name |
|---|---|
| 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. |
| pal_read_reset_status | `uint32_t pal_read_reset_status (uint64_t nvm_mem)` | This API reads the reset status from Non-Volatile memory. |

| API name | Function prototype | Description |
|---|---|---|
| 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. |

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


Note

```
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. |
| 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.<br><br>`irq_num`: hardware IRQ number<br><br>`mapped_irq_num`: mapped IRQ number<br><br>`isr`: Interrupt Service Routine that returns the status |

| API name | Function prototype | Description |
|---|---|---|
| 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.<br><br>`irq_num`: hardware IRQ number<br><br>`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.<br><br>`int_id`: interrupt ID which must be enabled and the service routine installed for<br><br>`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. |

| API name | Function prototype | Description |
|---|---|---|
| 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.<br><br>`bdf`: PCI Bus, Dev, and Func<br><br>`offset`: Offset in the configuration space from where data is to be read<br><br>`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.<br><br>`bdf`: PCI Bus, Dev, and Func<br><br>`offset`: Offset in the configuration space from where data is to be read<br><br>`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. |
| 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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number<br><br>`mvector`: Pointer to MSI(X) address |

| API name | Function prototype | Description |
|---|---|---|
| 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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number |
| p2p_support | `uint32_t pal_pcie_p2p_support();` | Checks P2P support in the PCIe hierarchy.<br><br>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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number<br><br>Returns 1 if P2P feature is not supported, else 0. |
| 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).<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number<br><br>Returns 0 if the device does not have ATC, else 1. |

| API name | Function prototype | Description |
|---|---|---|
| is_onchip_peripheral | `uint32_t pal_pcie_is_onchip_peripheral(uint32_t bdf);` | Checks if a PCIe function is an on-chip peripheral.<br><br>`bdf`: Segment, PCI Bus, Device, and Function.<br>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.<br><br>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.<br><br>`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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number<br><br>Returns 0 if RP is not involved in transaction forwarding, else 1. |
| 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.<br><br>`seg`: PCI segment number<br><br>`bus`: PCI bus number<br><br>`dev`: PCI device number<br><br>`fn`: PCI function number<br><br>`ext_cap_id`: PCI capability ID<br><br>`offset`: offset of the word in the capability configuration space<br><br>`val`: return value |

| API name | Function prototype | Description |
|---|---|---|
| get_bdf_wrapper | `uint32 pal_pcie_get_bdf_wrapper (uint32 ClassCode, uint32 StartBdf);` | Returns the Bus, Device, and Function for a matching class code.<br><br>`ClassCode`: 32-bit value of format `ClassCode << 16 \| sub_class_code`<br><br>`StartBdf:`<br><br>`0`: start enumeration from host bridge.<br><br>`1`: start enumeration from the input segment, Bus, Device.<br><br>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.<br><br>`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.<br><br>`bdf`: PCI Bus, Device, and Function<br><br>`offset`: offset in the PCI configuration space for that BDF<br><br>`val`: return value |
| 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.<br><br>`bdf`: PCI Bus, Device, and Function<br><br>`offset`: offset in the PCI configuration space for that BDF<br><br>`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.<br><br>`type`: Size of the offset required |

This data structure holds the PCIe subsystem information.


Note

```
/**
@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;
```

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. |
| 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.<br><br>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.<br><br>0 is returned when PASID support is not detected. Non-zero is returned if maximum PASID value supported for the input SMMU. |

| API name | Function prototype | Description |
|---|---|---|
| pa2iova | `uint64_t pal_smmu_pa2iova(uint64_t SmmuBase, uint64_t Pa);` | Converts physical address to I/O virtual address.<br><br>`SmmuBase`: physical address of the SMMU for conversion to virtual address.<br><br>`Pa`: physical address to use in conversion.<br><br>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.<br><br>`SmmuBase`: physical address of the SMMU that needs to be globally disabled.<br><br>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.<br><br>`smmu_base`: physical address of the SMMU for which PASID support is needed.<br><br>`pasid`: Process Address Space IDentifier.<br><br>Returns 0 for success and 1 for failure. |

### 4.2.6 DMA APIs

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

**Table 4-6: DMA APIs and their descriptions**

| API name | Function prototype | Description |
|---|---|---|
| create_info_table | `void pal_dma_create_info_table(DMA_INFO_TABLE *dma_info_table);` | Abstracts the steps to gather information on all the DMA-enabled controllers present in the system and fill the information in the dma_info_table. |
| start_from_device | `uint32_t pal_dma_start_from_device(void *dma_target_buf, uint32_t length,void *host, void *dev);` | Abstracts the functionality of performing a DMA operation from the device to DDR memory.<br><br>dma_target_buf is the target physical address in the memory where the DMA data is to be written.<br><br>0: success.<br><br>**IMPLEMENTATION DEFINED**: on error, the status is a non-zero value which is **IMPLEMENTATION DEFINED**. |
| start_to_device | `uint32_t pal_dma_start_to_device(void *dma_source_buf, uint32_t length, void *host, void *target, uint32_t timeout);` | Abstracts the functionality of performing a DMA operation to the device from DDR memory.<br><br>dma_source_buf: physical address in the memory where the DMA data is read from and has to be written to the device.<br><br>0: success.<br><br>**IMPLEMENATTION DEFINED**: on error, the status is a non-zero value which is **IMPLEMENATTION DEFINED**. |

| API name | Function prototype | Description |
|---|---|---|
| mem_alloc | `uint64_t pal_dma_mem_alloc(void **buffer, uint32_t length, void *dev, uint32_t flags);` | Allocates contiguous memory for DMA operations.<br><br>Supported values for flags are:<br><br>1: DMA_COHERENT<br><br>2: DMA_NOT_COHERENT<br><br>`dev` is a void pointer which can be used by the PAL layer to get the context of the request. This is same value that is returned by PAL during info table creation.<br><br>0: success.<br><br>**IMPLEMENTATION DEFINED**: on error, the status is a non-zero value which is **IMPLEMENTATION DEFINED**. |
| scsi_get_dma_addr | `void pal_dma_scsi_get_dma_addr(void *port, void *dma_addr, uint32_t *dma_len);` | This is a hook provided to extract the physical DMA address used by the DMA Requester for the last transaction. It is used by the test to verify if the address used by the DMA Requester was the same as the one allocated by the test. |
| mem_get_attrs | `int pal_dma_mem_get_attrs(void *buf, uint32_t *attr, uint32_t *sh);` | Returns the memory and Shareability attributes of the input address. The attributes are returned as per the MAIR definition in the Arm® ARM VMSA section.<br><br>0: success.<br><br>Non-zero: error, ignore the attribute and Shareability parameters. |
| dma_mem_free | `void pal_dma_mem_free(void *buffer, addr_t mem_dma, unsigned int length, void *port, unsigned int flags);` | Free the memory allocated by pal_dma_mem_alloc.<br>`buffer`: memory mapped to the DMA that is to be freed.<br><br>`mem_dma`: DMA address with respect to device.<br><br>`length`: size of the memory<br><br>`port`: ATA port structure.<br><br>`flags`: Value can be DMA_COHERENT or DMA_NOT_COHERENT. |


Note

This data structure captures the information about SATA or USB controllers which are DMA-enabled.

```
typedef struct {
uint32_t num_dma_ctrls;
DMA_INFO_BLOCK info[];  ///< Array of information blocks - per DMA
 controller
}DMA_INFO_TABLE;
```

This includes pointers to information such as port information and targets connected to the port. The present structures are defined only for SATA and USB. If other peripherals are to be supported, these structures must be enhanced.

```
/**
@brief DMA controllers info structure
**/
typedef enum {
DMA_TYPE_USB = 0x2000,
DMA_TYPE_SATA,
DMA_TYPE_OTHER,
}DMA_INFO_TYPE_e;

typedef struct {
DMA_INFO_TYPE_e type;
void *target;          ///< The actual info stored in these pointers is
 implementation specific.
void *port;
void *host;           ///< It will be used only by PAL. hence void.
uint32_t flags;
}DMA_INFO_BLOCK;
```

## 4.2.7 Exerciser APIs

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

**Table 4-7: 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:<br>1 – SNOOP_ATTRIBUTES<br><br>2 – LEGACY_IRQ<br><br>3 – DMA_ATTRIBUTES<br><br>4 – P2P_ATTRIBUTES<br><br>5 – PASID_ATTRIBUTES<br><br>6 – MSIX_ATTRIBUTES<br><br>7 – CFG_TXN_ATTRIBUTES<br><br>8 – ERROR_INJECT_TYPE<br><br>`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. |

| API Name | Function prototype | Description |
|---|---|---|
| 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:<br>1 – RESET, hardware in reset state.<br><br>2 – ON, this state is set after hardware is initialized and is ready to generate stimulus.<br><br>3 – OFF, this state is set to indicate that hardware can no longer generate stimulus.<br><br>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. |
| 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:<br>1 – START_DMA<br><br>2 – GENERATE_MSI<br><br>3 – GENERATE_L_INTR<br><br>4 – MEM_READ<br><br>5 – MEM_WRITE<br><br>6 – CLEAR_INTR<br><br>7 – PASID_TLP_START<br><br>8 – PASID_TLP_STOP<br><br>9 – TXN_NO_SNOOP_ENABLE<br><br>10 – TXN_NO_SNOOP_DISABLE<br><br>11 – START_TXN_MONITOR<br><br>12 – STOP_TXN_MONITOR<br><br>13 – ATS_TXN_REQ<br><br>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:<br>1 – EXERCISER_DATA_CFG_SPACE<br><br>2 – EXERCISER_DATA_BAR0_SPACE |
| is_bdf_exerciser | `uint32_t pal_is_bdf_exerciser(uint32_t bdf)` | Checks if the device is an exerciser.<br><br>Returns 1 if device is an exerciser, else 0. |

| API Name | Function prototype | Description |
|---|---|---|
| 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. |

### 4.2.8 Miscellaneous APIs

Miscellaneous APIs are described in the following table.

**Table 4-8: 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.<br><br>`string`: An ASCII string.<br><br>`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.<br><br>`addr`: Address to be written.<br><br>`string`: An ASCII string.<br><br>`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.<br><br>`FirstString`: The pointer to the first null-terminated ASCII string.<br><br>`SecondString`: The pointer to the second null-terminated ASCII string.<br><br>`Length`The maximum number of ASCII characters for comparison. |

| API name | Function prototype | Description |
|---|---|---|
| mmio_read | `uint32 pal_mmio_read(uint64 addr);` | Provides a single point of abstraction to read from all memory-mapped I/O addresses.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`return`: 8-bit data read from the input address |
| 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.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`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.<br><br>`addr`: 64-bit input address<br><br>`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. |

| API name | Function prototype | Description |
|---|---|---|
| mem_alloc | `void pal_mem_alloc(unsigned int size);` | Allocates memory of the requested size.<br><br>`size`: size of the memory region to be allocated<br><br>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. |
| 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.<br><br>`num_pe`: number of PEs in the system<br><br>`sizeofentry`: size of memory region allocated to each PE<br><br>Returns none. |
| mem_free | `void pal_mem_free(void *buffer);` | Frees the memory allocated by UEFI framework APIs.<br><br>`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.<br><br>`dest_buffer`: pointer to the destination buffer of the memory copy<br><br>`src_buffer`: pointer to the source buffer of the memory copy<br><br>len: number of bytes to copy from source buffer to destination buffer<br><br>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.<br><br>`src`: base address of the memory, source buffer to be compared<br><br>`dest`: destination buffer to be compared with<br><br>`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.<br><br>`bdf`: BDF of the requesting PCIe device<br><br>`size`: size of the memory region to be allocated<br><br>`pa`: physical address of the allocated memory |

| API name | Function prototype | Description |
|---|---|---|
| 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.<br><br>`bdf`: Bus, Device, and Function of the requesting PCIe device<br><br>`size`: size of memory region to be freed<br><br>`va`: virtual address of the memory to be freed<br><br>`pa`: physical address of the memory to be freed |
| mem_virt_to_phys | `void pal_mem_virt_to_phys(void *va);` | Returns the physical address of the input virtual address.<br><br>`va`: virtual address of the memory to be converted<br><br>Returns the physical address. |
| time_delay_ms | `uint64 pal_time_delay_ms (uint64 MicroSeconds);` | Stalls the CPU for the specified number of microseconds.<br><br>`MicroSeconds`: the minimum number of microseconds to be delayed<br><br>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.<br><br>`buf`: pointer to the buffer to fill<br><br>`size`: number of bytes in the buffer to fill<br><br>`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.<br>`Pa`: Physical Address of the memory to be converted.<br><br>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.<br><br>`alignment`: Specifies the alignment.<br><br>`size`: Requested memory allocation size.<br><br>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. |

| API name | Function prototype | Description |
|---|---|---|
| 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: Issue 0006-01**

| Change | Location |
|---|---|
| Added new PE APIs | 4.2.2 PE APIs on page 25 |

**Table A-3: Issue 0007-01**

| Change | Location |
|---|---|
| Added new module and API | 4.2.1 API naming convention on page 24 |
| Added new PE APIs | 4.2.2 PE APIs on page 25 |