Arm® Server Base System Architecture Compliance

Revision: r7p1

Validation Methodology

Non-Confidential

Issue 03

101544_0701_03_en

Arm® Server Base System Architecture Compliance
Validation Methodology

Release Information

Document history

Issue	Date	Confidentiality	Change
A	30 November 2016	Non-Confidential	Alpha release
B	31 March 2017	Non-Confidential	Beta release
C	13 July 2017	Non-Confidential	REL 1.0
D	19 January 2018	Non-Confidential	Alpha release for REL 2.0
E	11 May 2018	Non-Confidential	REL 2.0
0200-01	27 December 2018	Non-Confidential	REL 2.1. The document now follows a new numbering format.
0200-02	26 April 2019	Non-Confidential	REL 2.2
0200-03	18 September 2019	Non-Confidential	REL 2.3
0200-04	20 March 2020	Non-Confidential	REL 2.4
0300-01	30 September 2020	Non-Confidential	REL 3.0
0301-01	27 September 2021	Non-Confidential	REL 3.1
0302-01	26 July 2022	Non-Confidential	REL 3.2
0601-01	28 October 2022	Non-Confidential	REL 6.1

Issue	Date	Confidentiality	Change
0700-00	15 June 2022	Non-Confidential	REL 7.0 ALPHA release
0701-01	16 January 2023	Non-Confidential	REL 7.1 BETA-0 release
0701-02	28 March 2023	Non-Confidential	REL 7.1.1 BETA-1 release
0701-03	29 June 2023	Non-Confidential	REL 7.1.2 EAC release

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 Final, that is for a developed product.

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® Server Base System Architecture Compliance Validation Methodology.
- The number 101544_0701_03_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**.

Caution

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

Warning

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

Danger

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

Note

An important piece of information that needs your attention.

Tip

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

Remember

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® Base System Architecture 1.0	DEN0094C	Non-Confidential
Arm® Server Base System Architecture 7.1	DEN0029H	Non-Confidential
GICv3 and GICv4 Software Overview	DAI0492	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. About the Arm® SBSA ACS

This chapter provides an introduction to the Arm® SBSA 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
SBSA	Server Base System Architecture
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 Introduction to SBSA ACS

Server Base System Architecture (SBSA) specification specifies hardware system architecture which is based on Arm® 64-bit architecture that server system software such as operating systems, hypervisors, and firmware can rely on. It addresses PE features and key aspects of system architecture.

It ensures a standard system architecture to enable a suitably built single OS image to run on all hardware compliant with this specification. It also specifies features that firmware can rely on, allowing for some commonality in firmware implementation across platforms.

The SBSA architecture that is described in the *Arm® Server Base System Architecture Specification* defines the behavior of an abstract machine, referred to as an SBSA system. Implementations compliant with the SBSA architecture must conform to the behavior described in the specification.

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

2.3 Compliance tests

SBSA 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.
MPAM	Verifies MPAM compliance.
NIST	Verifies to determine the suitability of a generator for a cryptographic application.
PCIe	Verifies PCIe subsystem compliance.
PE	Verifies PE compliance.
Peripherals	Verifies USB, SATA, and UART compliance.
PMU	Verifies PMU compliance.
Power and Wakeup	Verifies system power states compliance.
RAS	Verifies RAS compliance.

Component	Description
SMMU	Verifies SMMU subsystem compliance.
Timer	Verifies PE timers and system timers compliance.
Watchdog	Verifies watchdog timer compliance.

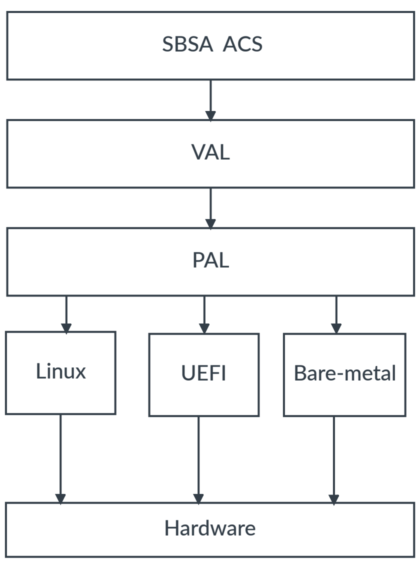
2.4 Layered software stack

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

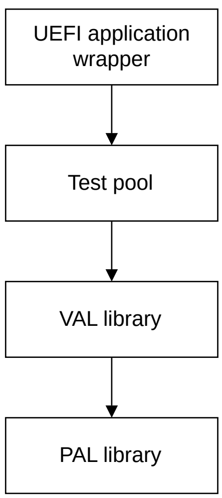
Layer	Description
SBSA ACS	Collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL.

Layer	Description
VAL	Provides a uniform view of all the underlying hardware and test infrastructure to the test suite.
PAL	Has 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

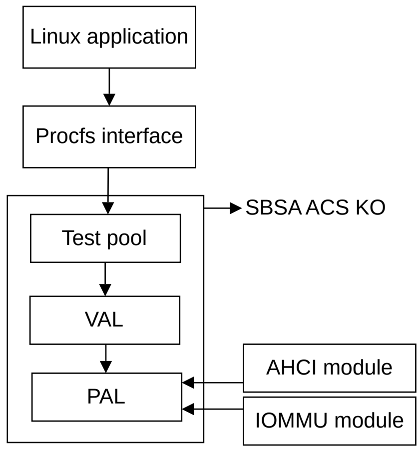


2.4.2 Compliance test software stack with Linux application

The stack is spread across user mode and kernel mode space. The Linux command-line application running in the user mode space and the kernel module communicate using a `procfs` interface. The test pool, VAL, and PAL layers are built as a kernel module.

The following figure is an example of the compliance test software stack with Linux application.

Figure 2-3: Software stack with Linux application



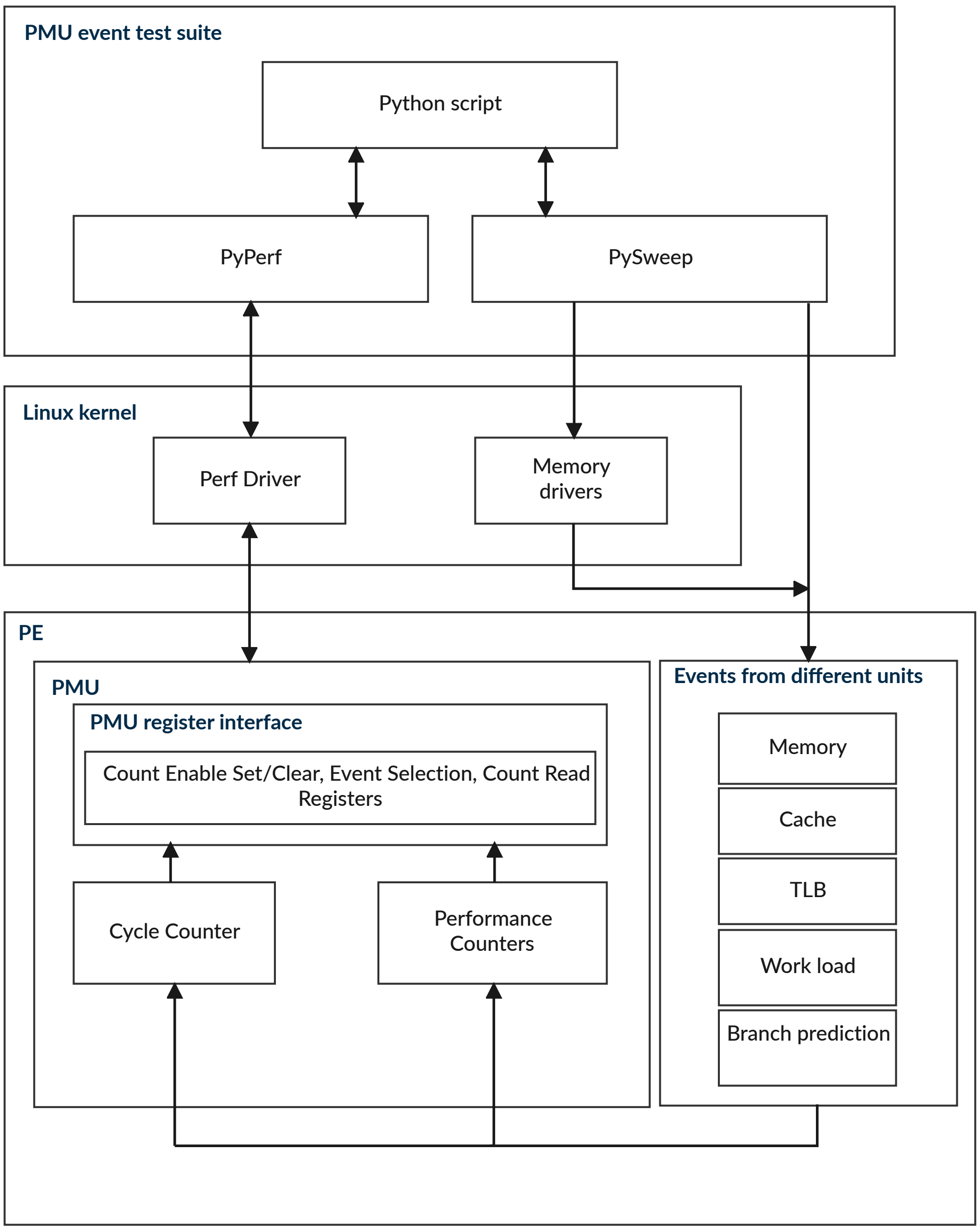
The SBSA command-line application initiates the tests and queries for status of the test using the standard `procfs` interface of the Linux OS. To avoid multiple data transfers between the kernel and user modes, the test suite, VAL, and PAL are built together as a kernel module.

Further, the PAL layer might need information from modules such as AHCI driver and the IOMMU driver which are outside the SBSA ACS kernel module. A separate patch file is provided to patch the drivers appropriately to export the required information. For details, see the *Arm® SBSA ACS User Guide*.

2.4.3 PMU event test suite

The PMU event test suite contains a Python script, Linux perf driver interface, and PySweep for workload generation. The following figure shows how the PMU event test suite Linux application is set up.

Figure 2-4: PMU event test suite Linux application



Python script

Reads the event number and event details from input text file. Calls event generation functions and monitor events.

PyPerf

Programs the PMU and monitor event count based on the event ID.

PySweep

Generates workload.

1. Instruction-related event generation: `src/loadcode.c` (I-Cache, I-TLB, INST_**, OP_**)
2. Data-related event generation: `src/loaddata.c` (D-Cache, D-TLB)
3. Branching event generation: `src/branch_prediction.c`

2.4.4 Coding guidelines

The coding guidelines followed for the implementation of the test suite are described in this section.

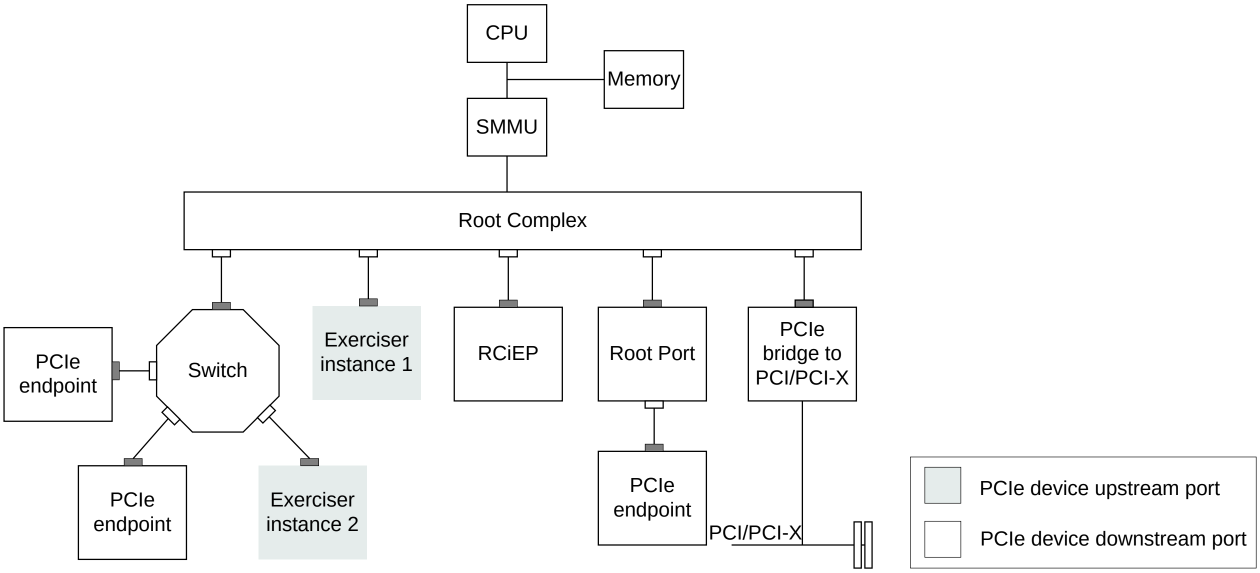
- All the tests call VAL APIs.
- VAL APIs might call PAL APIs depending on the requested functionality.
- A test does not directly interface with PAL functions.
- The test layer does not need any code modifications when porting from one platform to another.
- All the platform porting changes are limited 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 SBSA 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 2-5: Exerciser in an SoC



Root Complex integrated EndPoint (RCiEP) and Root Complex Event Collector (RCEC) are endpoints connected directly to Root Complex. PCIe endpoints are connected either to the Root Port or downstream ports. Bridges are used to connect PCI devices into PCIe hierarchy while switches are used to connect multiple PCIe devices to a single downstream port. PCIe devices access GIC, memory, and PE through the Root Complex, also called the host bridge.

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.

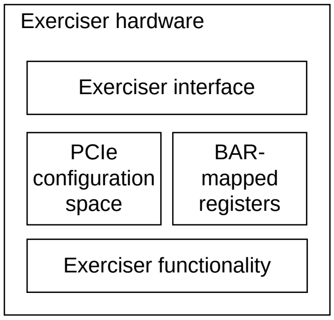
Note

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 2-6: Reference implementation of exerciser hardware

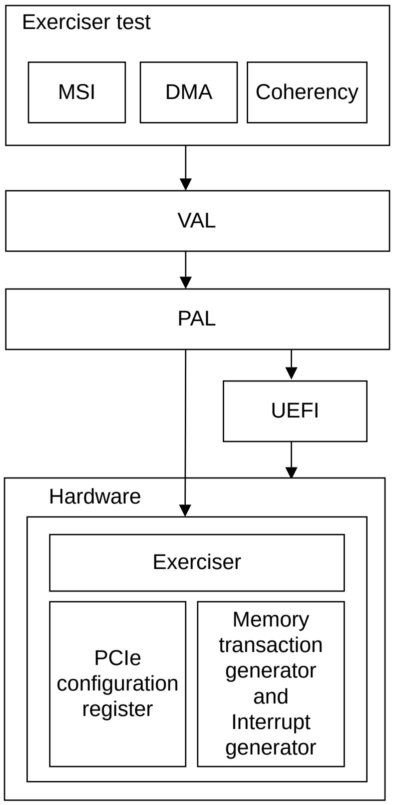


2.5.1 Compliance test software stack for exerciser with UEFI shell application

The exerciser tests validate device interrupts (legacy interrupt and MSI-X interrupt), DMA (address translation and memory access), and coherency behavior. 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 2-7: 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:

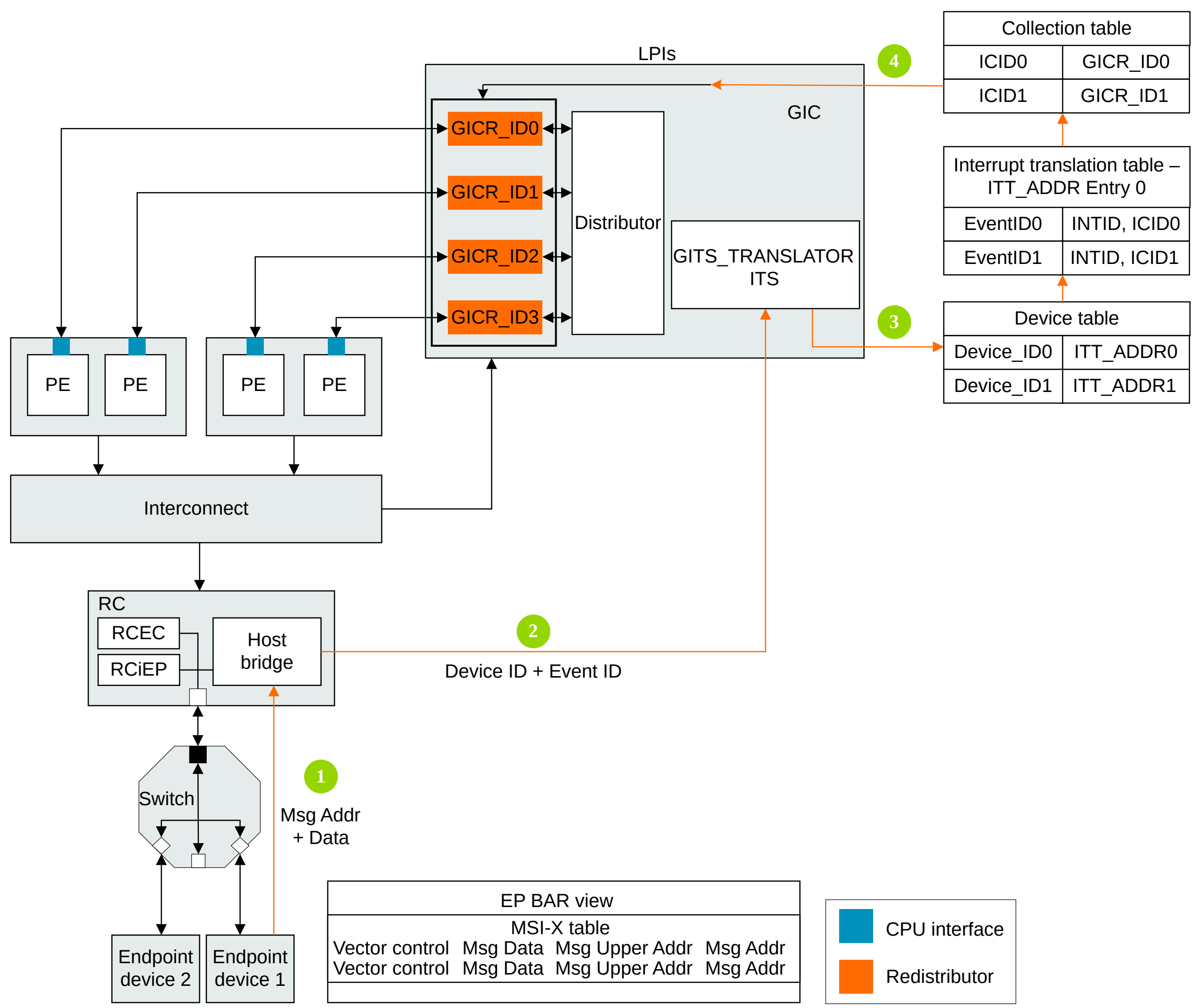
- The corresponding INTID for the input.
- The target Redistributor and, through this, the target PE for the INTID.

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.

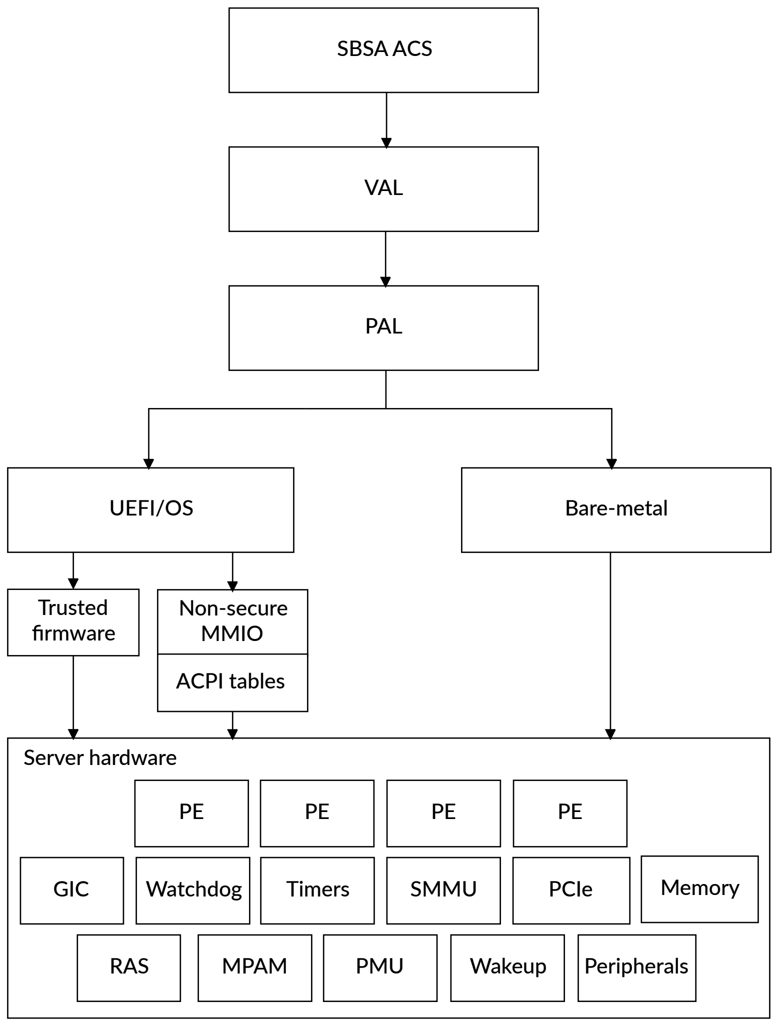
Figure 2-8: Routing MSI-X from Endpoint to PE through GIC ITS



2.7 Test platform abstraction

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

Figure 2-9: Test platform abstraction



The following table describes the SBSA abstraction terms.

Table 2-4: Abstraction terms and descriptions

Abstraction	Description
UEFI or OS	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 SBSA specification.

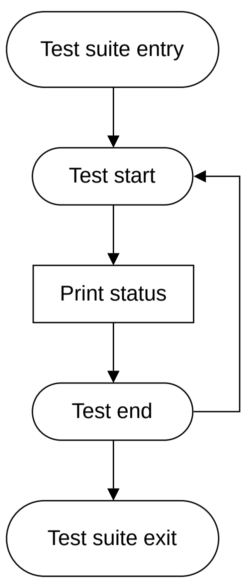
3. Execution flow control

This chapter describes the execution flow control used for SBSA 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.

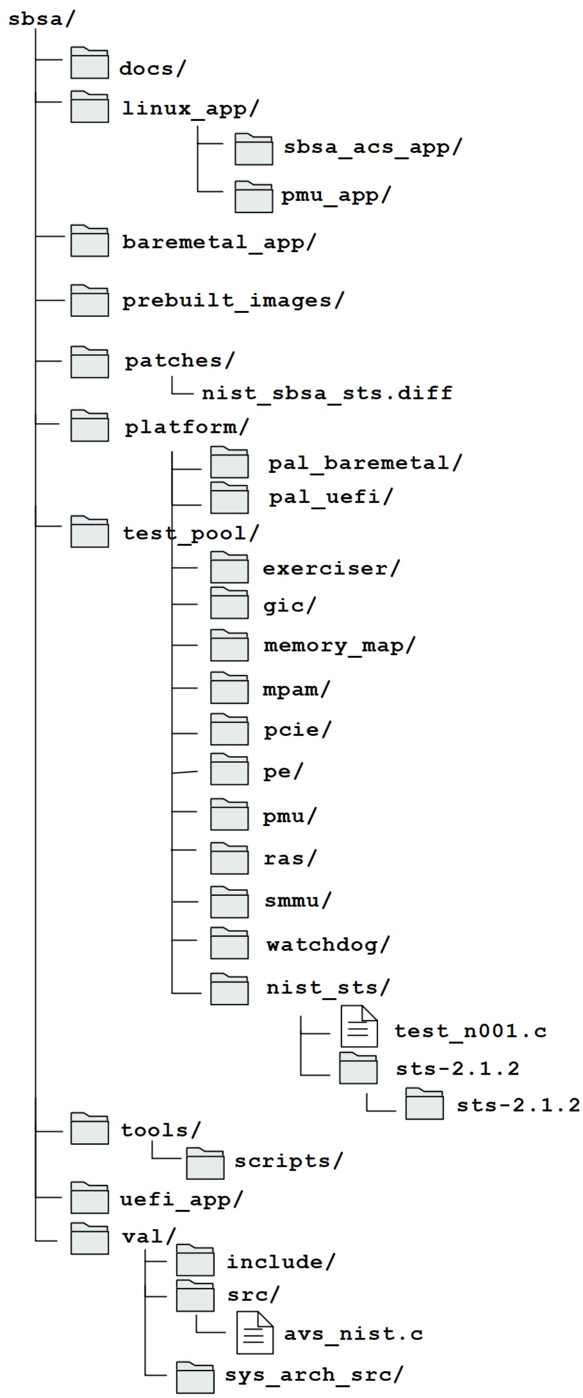
3.2 Test build and execution flow

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 SBSA ACS.

Figure 3-2: SBSA ACS directory structure



The following describes all the directories in SBSA ACS.

`docs`	Documentation.
`linux_app`	Linux command-line executable source code.
`baremetal_app`	Reference bare-metal application source to call into the test entry point.
`prebuilt_images`	Contains prebuilt images for the releases.
`patches`	Contains the SBSA NIST Statistical Test Suite (STS) patch.
`pal_uefi`	Platform code targeting UEFI implementation.
`pal_baremetal`	Platform reference code for Bare-metal.
`test_pool`	Test case source files for the test suite.
`tools`	Consists of scripts written for this suite.
`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.

3.2.2 Building the tests

This section provides reference information for building SBSA ACS as a UEFI Shell application and SBSA ACS kernel module.

Test build for UEFI

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

Test build for OS-based tests

The build steps for the Linux application-driven compliance suite, and SBSA ACS kernel module, which has a dependency for the SBSA ACS Linux application, are available in the README.

The build steps for the PMU events test, Linux application-driven compliance suite, and SBSA ACS kernel module, which has a dependency for the SBSA ACS Linux application, are available in the README.

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 test 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 locations:

- UEFI
- Linux
- Bare-metal

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
PE	pe
GIC	gic
Timer	timer
Watchdog	wd
PCIE	pcie
IOVirt	iovirt
SMMU	smmu
Peripheral	per
DMA	dma
Memory	memory
Exerciser	exerciser
Miscellaneous	print, mem, mmio
NIST	nist
MPAM	mpam
RAS	ras
PMU	pmu

4.2.2 PE APIs

These 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	`void pal_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.

API name	Function prototype	Description
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.
psci_get_conduit	`uint32_t pal_psci_get_conduit(void)`	Checks whether PSCI is implemented. If yes, which conduit does it use (HVC or SMC). Returns: • CONDUIT_NONE: PSCI is not implemented • CONDUIT_SMC: PSCI is implemented and uses SMC as the conduit. • CONDUIT_HVC: PSCI is implemented and uses HVC as the conduit.

Note

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

```
typedef struct {
  uint32_t   pe_num;                          /* PE Index */
  uint32_t   attr;                            /* PE attributes */
  uint64_t   mpidr;                           /* PE MPIDR */
  uint32_t   pmu_gsiv;                        /* PMU Interrupt */
  uint32_t   gmain_gsiv;                      /* GIC Maintenance
 Interrupt */
  uint32_t   acpi_proc_uid;                   /* ACPI Processor UID */
  uint32_t   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.

API name	Function prototype	Description
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. `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_t type;
  uint64_t base;
  uint32_t entry_id;
  uint64_t length;
  uint32_t flags;
  uint32_t spi_count;
  uint32_t 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_t bdf, uint32_t offset, uint32_t 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_t pal_pcie_get_bdf_wrapper (uint32_t ClassCode, uint32_t 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
device_driver_present	`uint32_t pal_pcie_device_driver_present(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn)`	Returns if driver present for PCIe device. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number Returns 0 if Driver present else 1
ecam_base	`uint64_t pal_pcie_ecam_base(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t func)`	Returns the ECAM address of the input PCIe device. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number Returns ECAM address if success, else NULL address
bar_mem_read	`uint32_t pal_pcie_bar_mem_read(uint32_t Bdf, uint64_t address, uint32_t *data)`	Reads 32-bit data from BAR space pointed by Bus, Device, Function and register offset. `Bdf`: BDF value for the device. `address`: BAR memory address. `data`: 32 bit value at BAR address.

API name	Function prototype	Description
bar_mem_write	`uint32_t pal_pcie_bar_mem_write(uint32_t Bdf, uint64_t address, uint32_t data)`	Writes 32-bit data to BAR space pointed by Bus, Device, Function and register offset. `Bdf`: BDF value for the device. `address`: BAR memory address. `data`: 32 bit value to write to BAR address.

Note

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;
```

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 IO-Virt APIs

These APIs provide the information and functionality required by the test suite that accesses features of IO virtualization system.

Table 4-5: IO-Virt APIs and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_iovirt_create_info_table(IOVIRT_INFO_TABLE *iovirt);`	Abstracts the steps to fill in the iovirt table with the details of the Virtualization sub-system in the system.
unique_rid_strid_map	`uint32_t pal_iovirt_unique_rid_strid_map(uint64_t rc_block);`	Abstracts the mechanism to check if a Root Complex node has unique requestor ID to Stream ID mapping. 0 indicates a fail since the mapping is not unique. 1 indicates a pass since the mapping is unique.

API name	Function prototype	Description
check_unique_ctx_initd	uint32_t pal_iovirt_check_unique_ctx_intid(uint64_t smmu_block);	Abstracts the mechanism to check if a given SMMU node has unique context bank interrupt IDs. 0 indicates fail and 1 indicates pass.
get_rc_smmu_base	uint64_t pal_iovirt_get_rc_smmu_base(IOVIRT_INFO_TABLE *iovirt, uint32_t rc_seg_num, uint32_t rid);	Returns the base address of SMMU if a Root Complex is behind an SMMU, otherwise returns NULL.

Note

The following data structure is filled in by the above function. This data structure captures all the information related to SMMUs, PCIe root complex, GIC-ITS and any other named components involved in the Virtualization sub-system of the SoC.

The information captured includes interrupt routing tables, memory maps, and the base addresses of the various components.

```
typedef struct {
  uint32_t arch_major_rev;  /* Version 1 or 2 or 3 */
  uint64_t base;            /* SMMU Controller base address */
}SMMU_INFO_BLOCK;

typedef struct {
  uint32_t segment;
  uint32_t ats_attr;
  uint32_t cca;             /* Cache Coherency Attribute */
  uint64_t smmu_base;
}IOVIRT_RC_INFO_BLOCK;

typedef struct {
uint64_t base;
uint32_t overflow_gsiv;
uint32_t node_ref; /* offest to the IORT node in IORT ACPI table*/
uint64_t smmu_base; /* SMMU base to which component is attached, else
 NULL */
}IOVIRT_PMCG_INFO_BLOCK;

typedef struct {
  uint64_t smmu_base;                   /* SMMU base to which
 component is attached, else NULL */
  uint32_t cca;                         /* Cache Coherency Attribute
 */
  char name[MAX_NAMED_COMP_LENGTH];   /* Device object name */
} IOVIRT_NAMED_COMP_INFO_BLOCK;

typedef struct {
  uint32_t input_base;
  uint32_t id_count;
  uint32_t output_base;
  uint32_t output_ref;     /* output ref captured here is offset to
 iovirt block in
                            IOVIRT info table not IORT ACPI table in
 memory */
}ID_MAP;

typedef union {
  uint32_t id[4];
  ID_MAP map;
}NODE_DATA_MAP;

typedef union {
```

```
  IOVIRT_NAMED_COMP_INFO_BLOCK named_comp;
  IOVIRT_RC_INFO_BLOCK rc;
  IOVIRT_PMCG_INFO_BLOCK pmcg;
  uint32_t its_count;
  SMMU_INFO_BLOCK smmu;
}NODE_DATA;

typedef struct {
  uint32_t type;
  uint32_t num_data_map;
  NODE_DATA data;
  uint32_t flags;
  NODE_DATA_MAP data_map[];
}IOVIRT_BLOCK;

typedef struct {
  uint32_t num_blocks;
  uint32_t num_smmus;
  uint32_t num_pci_rcs;
  uint32_t num_named_components;
  uint32_t num_its_groups;
  uint32_t num_pmcgs;
  IOVIRT_BLOCK blocks[];
}IOVIRT_INFO_TABLE;
```

4.2.6 SMMU APIs

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

Table 4-6: 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. 0 is returned if address belongs to the device. Non-zero is returned if there are **ARCHITECTURE 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.

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. `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.

4.2.7 DMA APIs

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

Table 4-7: 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. dma_target_buf is the target physical address in the memory where the DMA data is to be written. 0: success. **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. dma_source_buf: physical address in the memory where the DMA data is read from and has to be written to the device. 0: success **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. Supported values for flags are: 1: DMA_COHERENT 2: DMA_NOT_COHERENT `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. 0: success. **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. 0: success. 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. `buffer`: memory mapped to the DMA that is to be freed `mem_dma`: DMA address with respect to device `length`: size of the memory `port`: ATA port structure `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;
```

4.2.8 Exerciser APIs

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

Table 4-8: 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.

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: 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.
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

API Name	Function prototype	Description
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.
get_legacy_irq_map	`uint32_t pal_exerciser_get_legacy_irq_map(uint32_t Seg, uint32_t Bus, uint32_t Dev, uint32_t Fn, PERIPHERAL_IRQ_MAP *IrqMap)`	Get legacy IRQ routing for a PCI device. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number irq_map pointer to IRQ map structure Returns IRQ routing map

4.2.9 Miscellaneous APIs

Miscellaneous APIs are described in the following table.

Table 4-9: 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.

API name	Function prototype	Description
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_t pal_mmio_read(uint64_t 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	`uint8_t pal_mmio_read8(uint64_t 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
mmio_read16	`uint16_t pal_mmio_read16(uint64_t 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	`uint64_t pal_mmio_read64(uint64_t 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(uint64_t addr,uint32_t 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	`void pal_mmio_write8(unit64_t addr,uint8_t 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	`void pal_mmio_write16(unit64_t addr,uint16_t 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

API name	Function prototype	Description
mmio_write64	`void pal_mmio_write8(unit64_t addr,uint64_t 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	`void pal_mem_free_shared(void);`	Frees the allocated shared memory region.
mem_get_shared_addr	`uint64_t 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.
mem_allocate_shared	`void 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);`	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_t pal_mem_compare(void *src, void *dest, uint32_t 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

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. `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
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_t pal_time_delay_ms (uint64_t 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_t size, uint8_t 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_t 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_free_aligned	`void pal_mem_free_aligned(void *buffer);`	Frees the aligned memory allocated by aligned_alloc. Buffer: The base address of the aligned memory range.
mem_alloc_at_address	`void *pal_mem_alloc_at_address(uint64_t mem_base, uint64_t size);`	Allocate 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);`	Free the allocated memory in the given memory base.

4.2.10 NIST API

This API is used for randomness testing.

Table 4-10: NIST API and its description

API name	Function prototype	Description
generate_rng	uint32_t pal_nist_generate_rng(uint32_t *rng_buffer);	Generates a 32-bit random number. `rng_buffer`: pointer to store the random data Returns success or failure.

4.2.11 PMU APIs

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

Table 4-11: PMU API and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_pmu_create_info_table(PMU_INFO_TABLE *PmuTable);`	Abstracts the steps to gather information about system PMU in the platform and fills the PmuTable.
get_event_info	`uint32_t pal_pmu_get_event_info (PMU_EVENT_TYPE_e event_type, PMU_NODE_INFO_TYPE node_type);`	Returns the IMPLEMENTATION DEFINED event ID based on the requested PMU event and PMU node type. The IMPLEMENTATION DEFINED event ID must be filled into `event_list` array at `pal_pmu.c`.
check_monitor_count_value	`uint32_t pal_pmu_check_monitor_count_value(uint64_t interface_acpiid, uint32_t count_value, uint32_t eventid);`	This API checks if the PMU monitor count value is valid.
generate_traffic	`uint32_t pal_generate_traffic(uint64_t interface_acpiid, uint32_t pmu_node_index, uint32_t mon_index, uint32_t eventid);`	This API generates required workload for the given PMU node and event id.
get_multi_traffic_support_interface	`uint32_t pal_pmu_get_multi_traffic_support_interface(uint64_t *interface_acpiid, uint32_t *num_traffic_type_support);`	This API returns the ACPI HID of the interface which supports multi traffic and number of multi traffic supported.

Note

The following structures hold information about the system PMU.

```
/*Instance of system pmu info*/
typedef struct {
 uint8_t  type;              /* The component that this PMU block is
 associated with*/
 uint64_t primary_instance;  /* Primary node instance, specific to the
 PMU type*/
 uint32_t secondary_instance;/* Secondary node instance, specific to
 the PMU type*/
 uint8_t dual_page_extension;/* Support of the dual-page mode*/
 uint64_t base0;             /* Base address of Page 0 of the PMU*/
 uint64_t base1;       /* Base address of Page 1 of the PMU,valid only
 if dual_page_extension is 1*/
} PMU_INFO_BLOCK;
```

```
typedef struct {
    uint32_t pmu_count;        /* Total number of PMU info blocks*/
    PMU_INFO_BLOCK info[];     /* PMU info blocks for each PMU nodes*/
} PMU_INFO_TABLE;
```

4.2.12 RAS APIs

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

Table 4-12: RAS APIs and their descriptions

API name	Function prototype	Description
ras_create_info_table	`void pal_ras_create_info_table(RAS_INFO_TABLE *RasInfoTable);`	Abstracts the steps to gather information about RAS from AEST ACPI Table in the platform and fills the `RasInfoTable`.
ras2_create_info_table	`void pal_ras2_create_info_table(RAS2_INFO_TABLE *ras2_info_table);`	Abstracts the steps to gather information about RAS from RAS2 ACPI Table in the platform and fills the `Ras2InfoTable`.
setup_error	`uint32_t pal_ras_setup_error(RAS_ERR_IN_t in_param, RAS_ERR_OUT_t *out_param);`	Platform defined API to setup the error. `in_param`: Input parameters set from the test which are used to setup the error. `out_param`: Return values from the platform which are used in the test.
inject_error	`uint32_t pal_ras_inject_error(RAS_ERR_IN_t in_param, RAS_ERR_OUT_t *out_param);`	Platform defined API to inject the error. `in_param`: Input parameters set from the test which are used to setup the error. `out_param`: Return values from the platform which are used in the test.
wait_timeout	`void pal_ras_wait_timeout(uint32_t count);`	Platform defined API to wait for a timeout. `count`: Multiplier for timeout_small/medium.

API name	Function prototype	Description
check_plat_poison_support	`uint32_t pal_ras_check_plat_poison_support()`	Returns the Poison storage and forwarding support capability in the platform. 0: Poison storage and forwarding not supported. 1: Poison storage and forwarding supported.

Note

- The following enumerator holds the RAS node types:

```
typedef enum {
   NODE_TYPE_PE = 0x0,
   NODE_TYPE_MC = 0x1,
   NODE_TYPE_SMMU = 0x2,
   NODE_TYPE_VDR = 0x3,
   NODE_TYPE_GIC = 0x4,
   NODE_TYPE_LAST_ENTRY
 } RAS_NODE_TYPE_e;
```

- The following structures hold information about each RAS node.

```
typedef enum {
  RAS_INTF_TYPE_SYS_REG,      /* System register RAS node interface
 type */
  RAS_INTF_TYPE_MMIO          /* MMIO RAS node interface type */
} RAS_NODE_INTF_TYPE;

typedef struct {
  uint32_t  processor_id;
  uint32_t  resource_type;
  uint32_t  flags;
  uint64_t  affinity;
  uint64_t  res_specific_data;  /* Resource Specific Data */
} RAS_NODE_PE_DATA;

typedef struct {
  uint32_t  proximity_domain;
} RAS_NODE_MC_DATA;

typedef struct {
  RAS_NODE_INTF_TYPE  intf_type;   /* Interface Type */
  uint32_t  flags;
  uint64_t  base_addr;              /* Base address to MMIO region,
 valid for MMIO intf type */
  uint32_t  start_rec_index;        /* Start Record Index */
  uint32_t  num_err_rec;            /* Number of error records
 (implemented & unimplemented)*/
  uint64_t  err_rec_implement;      /* bitmap of error records
 implemented */
  uint64_t  err_status_reporting;   /* bitmap indicates which error
 records within this error
                                     node support error status
 reporting using ERRGSR */
  uint64_t  addressing_mode;        /* bitmap based policy for
 ERR<n>ADDR field of error records */
} RAS_INTERFACE_INFO;

typedef struct {
  uint32_t  type;
  uint32_t  flag;
  uint32_t  gsiv;
  uint32_t  its_grp_id;
} RAS_INTERRUPT_INFO;

typedef union {
  RAS_NODE_PE_DATA    pe;
```

```
  RAS_NODE_MC_DATA    mc;
} RAS_NODE_DATA;

typedef struct {
  RAS_NODE_TYPE_e     type;              /* Node Type PE/GIC/SMMU */
  UINT16_t              length;            /* Length of the Node */
  uint64_t              num_intr_entries; /* Number of Interrupt
 Entry */
  RAS_NODE_DATA       node_data;         /* Node Specific Data */
  RAS_INTERFACE_INFO  intf_info;         /* Node Interface Info */
  RAS_INTERRUPT_INFO  intr_info[2];      /* Node Interrupt Info */
} RAS_NODE_INFO;

typedef struct {
  uint32_t  num_nodes;      /* Number of total RAS Nodes */
  uint32_t  num_pe_node;    /* Number of PE RAS Nodes */
  uint32_t  num_mc_node;    /* Number of Memory Controller Nodes */
  RAS_NODE_INFO  node[];  /* Array of RAS nodes */
} RAS_INFO_TABLE;

typedef enum {
  RAS2_TYPE_MEMORY = 0   /* RAS2 memory feature type*/
} RAS2_FEAT_TYPE;

typedef struct {
  uint32_t  proximity_domain;      /* Proximity domain of the memory
 */
  uint32_t  patrol_scrub_support;  /* Patrol srub support flag */
} RAS2_MEM_INFO;

typedef union {
  RAS2_MEM_INFO mem_feat_info;   /* Memory feature specific info */
} RAS2_BLOCK_INFO;

typedef struct {
  RAS2_FEAT_TYPE type;                        /* RAS2 feature type*/
  RAS2_BLOCK_INFO block_info;     /* RAS2 block info */
} RAS2_BLOCK;

typedef struct {
  uint32_t num_all_block;         /* Number of RAS2 feature blocks */
  uint32_t num_of_mem_block;      /* Number of memory feature blocks
 */
  RAS2_BLOCK blocks[];
} RAS2_INFO_TABLE;
```

4.2.13 MPAM APIs

These APIs provide the information and functionality required by the test suite that accesses features of MPAM and other information to verify it.

Table 4-13: MPAM APIs and their descriptions

API name	Function prototype	Description
create_info_table	`void pal_mpam_create_info_table(MPAM_INFO_TABLE *MpamTable);`	Abstracts the steps to gather information about MPAM in the platform and fills the MpamTable.
create_info_table	`void pal_hmat_create_info_table(HMAT_INFO_TABLE *HmatTable);`	Abstracts the steps to gather information about memory attributes in the platform and fills the HmatTable.
create_info_table	`void pal_srat_create_info_table(SRAT_INFO_TABLE * SratTable);`	Abstracts the steps to gather information about memory range in the platform and fills the SratTable.

API name	Function prototype	Description
create_info_table	`void pal_cache_create_info_table(CACHE_INFO_TABLE *CacheTable, PE_INFO_TABLE *PeTable);`	Abstracts the steps to gather information about Caches in the platform and fills the CacheTable.

Note

- The following structures hold information about the cache information platform.

```
/*only the fields and flags required by ACS are parsed from ACPI
 PPTT table*/
/*Cache flags indicate validity of cache info provided by PPTT
 Table*/
typedef struct {
  uint32_t size_property_valid;
  uint32_t cache_type_valid;
  uint32_t cache_id_valid;
} CACHE_FLAGS;

/* Since most of platform doesn't support cache id field (ACPI
 6.4+), ACS uses PPTT offset as key
   to uniquely identify a cache, In future once platforms align with
 ACPI 6.4+ my_offset member
   might be removed from cache entry*/
typedef struct {
  CACHE_FLAGS flags;         /* Cache flags */
  uint32_t my_offset;          /* Cache PPTT structure offset */
  uint32_t next_level_index;   /* Index of next level cache entry in
 CACHE_INFO_TABLE */
  uint32_t size;               /* Size of the cache in bytes */
  uint32_t cache_id;           /* Unique, non-zero identifier for
 this cache */
  uint32_t is_private;         /* Field indicate whether cache is
 private */
  UINT8  cache_type;         /* Cache type */
} CACHE_INFO_ENTRY;

typedef struct {
  uint32_t num_of_cache;             /* Total of number of cache info
 entries */
  CACHE_INFO_ENTRY cache_info[];   /* Array of cache info entries */
} CACHE_INFO_TABLE;
```

- The following structures hold memory attribute information about the platform.

```
typedef struct {
  uint32_t mem_prox_domain;   /* Proximity domain of the memory
 region*/
  uint64_t write_bw;          /* Maximum write bandwidth */
  uint64_t read_bw;           /* Maximum read bandwidth */
} HMAT_BW_ENTRY;

typedef struct {
  uint32_t num_of_mem_prox_domain; /* Number of Memory Proximity
 Domains */
  HMAT_BW_ENTRY bw_info[];         /* Array of bandwidth info based
 on proximity domain */
} HMAT_INFO_TABLE;
```

- The following structures hold the memory range information and GICC affinity structure information about the platform.

```
typedef enum {
SRAT_NODE_MEM_AFF = 0x01,
SRAT_NODE_GICC_AFF = 0x03
} SRAT_NODE_TYPE_e;

/**@brief SRAT GICC Affinity Structure**/

typedef struct {
```

```
uint32_t prox_domain;   /* Proximity domain*/
uint32_t proc_uid;      /* ACPI Processor UID */
uint32_t flags;         /* Flags*/
uint32_t clk_domain;    /* Clock Domain*/
} SRAT_GICC_AFF_ENTRY;

/**@brief SRAT Memory Affinity Structure**/

typedef struct {
uint32_t prox_domain;   /* Proximity domain */
uint32_t flags;         /* flags */
uint64_t addr_base;     /* mem range address base */
uint64_t addr_len;      /* mem range address len */
} SRAT_MEM_AFF_ENTRY;

typedef union {
SRAT_MEM_AFF_ENTRY mem_aff;
SRAT_GICC_AFF_ENTRY gicc_aff;
} SRAT_NODE_INFO;

typedef struct {
uint32_t node_type;      /* Node type*/
SRAT_NODE_INFO node_data;
} SRAT_INFO_ENTRY;

typedef struct {
uint32_t num_of_srat_entries;
uint32_t num_of_mem_ranges;
SRAT_INFO_ENTRY srat_info[];
} SRAT_INFO_TABLE;
```

- The following structures hold information about MPAM in the platform.

```
/*MPAM resource Node*/
typedef struct {
  uint8_t  ris_index;
  uint8_t  locator_type; /* Identifies location of this resource */
  uint64_t descriptor1;  /* Primary acpi description of location */
  uint32_t descriptor2;  /* Secondary acpi description of location
 */
} MPAM_RESOURCE_NODE;

/*MPAM MSC Node*/

typedef struct {
  uint64_t  msc_base_addr; /* base addr of mem-map MSC reg */
  uint32_t  msc_addr_len;  /*  MSC mem map size */
  uint32_t  max_nrdy;      /* max time in microseconds that MSC not
 ready after config change */
  uint32_t   rsrc_count;   /* number of resource nodes */
  MPAM_RESOURCE_NODE rsrc_node[]; /* Details of resource node */
} MPAM_MSC_NODE;

/*MPAM info table*/

 typedef struct {
     uint32_t msc_count;        /* Number of MSC node */
     MPAM_MSC_NODE msc_node[];  /* Details of MSC node */
 } MPAM_INFO_TABLE;
```

Appendix A NIST Statistical Test Suite

This appendix describes the integration of NIST Statistical Test Suite with SBSA ACS.

A.1 NIST Statistical Test Suite

Randomness testing plays a fundamental role in many areas of computer science, especially cryptography. Well-designed cryptographic primitives like hash functions and stream ciphers should produce pseudorandom data.

The outputs of such generators may be used in cryptographic applications like generation of key material. Generators suitable for use in cryptographic applications must meet stronger requirements than for other applications. In particular, their outputs must be unpredictable in the absence of knowledge of the inputs.

Statistical test suites

Randomness testing is performed using test suites consisting of many tests, each focusing on a different feature. These tests can be used as the first steps in determining if a generator is suitable for a particular cryptographic application.

SBSA ACS with NIST STS

There are five well-known statistical test suites namely NIST Statistical Test Suite (STS), Diehard, TestU01, ENT, and CryptX. Only the first three test suites are commonly used for the randomness analysis because CryptX is a commercial software and ENT provides only basic randomness testing. Since NIST STS has a special position for being published as an official document, it is often used in the preparation of formal certifications or approvals.

Building NIST STS with SBSA ACS

To build NIST STS with SBSA ACS, NIST STS 2.1.2 package is required and downloaded automatically as part of the build process.

See the updated version of the NIST STS tool for randomness testing documentation. The reason for the update is, the original source code provided with NIST does not compile cleanly in UEFI because it does not provide erf() and erfc() functions in the standard math library. Implementation of these functions has been added as part of SBSA VAL and a patch file is created.

Running NIST STS with SBSA ACS

For information on running NIST STS, see the *Arm® SBSA NIST User Guide*. For details about NIST STS, see A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.

Interpreting the results

The final analysis report is generated after the statistical testing is complete. It contains a summary of empirical results that are displayed on the console. A test is unsuccessful when P-value < 0.01. Then the sequence under test should be considered as non-random.

The minimum pass rate for each statistical test except for the random excursion (variant) test is approximately 8 for a sample size of ten binary sequences. The minimum pass rate for the random excursion (variant) test is undefined.

Note

For SBSA compliance, passing NIST STS is optional.

Appendix B Revisions

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

B.1 Revisions

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

Table B-1: Differences between Issue E and Issue 0200-01

Change	Location
Information about exerciser is added.	See the following sections: • 2.3 Compliance tests on page 12. • 3.2 Test build and execution flow on page 24. • 4.2.1 API naming convention on page 27. • 4.2.8 Exerciser APIs on page 41.

Table B-2: Differences between Issue 0200-01 and Issue 0200-02

Change	Location
A note about exerciser is added.	See 2.3 Compliance tests on page 12.
`pal_baremetal` folder is added to the directory structure.	See 3.2 Test build and execution flow on page 24.
Added a note about PAL bare-metal reference code.	See 4.1 Overview of PAL API on page 27.

Table B-3: Differences between Issue 0200-02 and Issue 0200-03

Change	Location
No technical changes.	-

Table B-4: Differences between Issue 0200-03 and Issue 0200-04

Change	Location
A new section about exerciser is added.	See 2.5 Exerciser on page 17.
NIST STS information is updated in these topics.	See • 2.3 Compliance tests on page 12. • 3.2 Test build and execution flow on page 24. • 4.2 PAL API definitions on page 27.
APIs are added in all the modules.	See 4.2 PAL API definitions on page 27.
A new appendix about NIST STS is added.	See A. NIST Statistical Test Suite on page 53.

Table B-5: Differences between Issue 0200-04 and Issue 0300-01

Change	Location
A new section about GIC ITS is added.	See 2.6 GIC ITS on page 20.
GIC ITS PAL APIs are added to GIC APIs section.	See 4.2.3 GIC APIs on page 29.
SBSA ACS directory structure is updated.	See 3.2.1 Source code directory on page 24.

Change	Location
read_cfg and write_cfg APIs in the PCIe APIs table are updated.	See 4.2.4 PCIe APIs on page 30.
New configuration parameters are added to the Exerciser APIs set_param and ops.	See 4.2.8 Exerciser APIs on page 41.
New APIs are added to Miscellaneous APIs section.	See 4.2.9 Miscellaneous APIs on page 43.

Table B-6: Differences between Issue 0300-01 and Issue 0700-00

Change	Location
Updated the SBSA ACS directory structure.	See 3.2.1 Source code directory on page 24.
Added a new section for PMU event test suite.	See 2.4.3 PMU event test suite on page 15.
Added new APIs.	See: • 4.2.11 PMU APIs on page 47 • 4.2.12 RAS APIs on page 48 • 4.2.13 MPAM APIs on page 50
Updated the following sections: • Abbreviations • Compliance tests • Test platform abstraction • Building the tests • API naming convention	See: • 2.1 Abbreviations on page 11 • 2.3 Compliance tests on page 12 • 2.7 Test platform abstraction on page 21 • 3.2.2 Building the tests on page 26 • 4.2.1 API naming convention on page 27

Table B-7: Differences between Issue 0302-01 and Issue 0601-01

Change	Location
Added an abbreviation for HVC	See 2.1 Abbreviations on page 11
Enhancement changes	Applicable sections.

Table B-8: Differences between Issue 0700-00 and Issue 0701-01

Change	Location
Updated abbreviations table.	See 2.1 Abbreviations on page 11
Added memory to the compliance test components.	See: • 2.3 Compliance tests on page 12 • 2.7 Test platform abstraction on page 21
Added baremetal_app and prebuilt_images folders to the directory structure.	See 3.2.1 Source code directory on page 24
Updated supported configuration parameters for set_param and ops.	See 4.2.8 Exerciser APIs on page 41
Added new APIs in PMU API.	See 4.2.11 PMU APIs on page 47
Updated RAS APIs.	See 4.2.12 RAS APIs on page 48
Updated information structures.	See: • 4.2.2 PE APIs on page 28 • 4.2.5 IO-Virt APIs on page 36 • 4.2.12 RAS APIs on page 48 • 4.2.13 MPAM APIs on page 50

Table B-9: Differences between Issue 0701-01 and Issue 0701-02

Change	Location
Updated the Figure 2-1: Layered software stack	See 2.4 Layered software stack on page 13
Updated PAL API's to PE, PCIe and Miscellaneous modules	See: • 4.2.2 PE APIs on page 28 • 4.2.4 PCIe APIs on page 30 • 4.2.9 Miscellaneous APIs on page 43

Table B-10: Differences between Issue 0701-02 and Issue 0701-03

Change	Location
Replaced UINT32 with uint32_t and UINT64 with uint64_t in PCIe APIs and PE APIs.	See: • 4.2.2 PE APIs on page 28 • 4.2.4 PCIe APIs on page 30
Updated the interrupt routine table in IO-Virt APIs.	See 4.2.5 IO-Virt APIs on page 36
Updated the Function prototype for Miscellaneous APIs.	See 4.2.9 Miscellaneous APIs on page 43
Updated the Function prototype for NIST API.	See 4.2.10 NIST API on page 47
Replaced UINT32 with uint32_t and UINT64 with uint64_t in PMU APIs and updated the API name.	See 4.2.11 PMU APIs on page 47
Replaced UINT32 with uint32_t and UINT64 with uint64_t in RAS APIs.	See 4.2.12 RAS APIs on page 48
Replaced UINT32 with uint32_t in MPAM APIs.	See 4.2.13 MPAM APIs on page 50