Arm® SBSA Architecture Compliance

Revision: r3p2

Validation Methodology

Confidential

Issue 01

101544_0302_01_en

Contents

1. Introduction..8
1.1 Conventions..8
1.2 Additional reading..8
1.3 Other information..9

2. About the Arm® SBSA ACS..10
2.1 Abbreviations..10
2.2 Introduction to SBSA ACS..10
2.3 Compliance tests..11
2.4 Layered software stack..11
2.4.1 Compliance test software stack with UEFI application..13
2.4.2 Compliance test software stack with Linux application..13
2.4.3 Coding guidelines..14
2.5 Exerciser..14
2.5.1 Compliance test software stack for exerciser with UEFI shell application..16
2.6 GIC ITS..17
2.7 Test platform abstraction..19

3. Execution flow control..20
3.1 Execution flow control..20
3.2 Test build and execution flow..20
3.2.1 Source code directory..21
3.2.2 Building the tests..22

4. Platform Abstraction Layer..23
4.1 Overview of PAL API..23
4.2 PAL API definitions..23
4.2.1 API naming convention..24
4.2.2 PE APIs..24
4.2.3 GIC APIs..25
4.2.4 Timer APIs..26
4.2.5 PCIe APIs..27
4.2.6 IO-Virt APIs..32

4.2.7 SMMU APIs....33
4.2.8 Peripheral APIs....34
4.2.9 DMA APIs....37
4.2.10 Exerciser....39
4.2.11 Miscellaneous APIs....41
4.2.12 NIST API....44

A. NIST Statistical Test Suite....45
A.1 NIST Statistical Test Suite....45

B. Revisions....47
B.1 Revisions....47

Arm® SBSA 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

Proprietary Notice

This document is CONFIDENTIAL and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorized by Arm to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: (i) for the purposes of determining whether implementations infringe any third party patents;(ii) for developing technology or products which avoid any of Arm's intellectual property; or (iii) as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or (iv) for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20348)

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

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-enterprise-acs@arm.com. Give:

- The title Arm® SBSA Architecture Compliance Validation Methodology.
- The number 101544_0302_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Inclusive language commitment

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

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

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

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.

Typographic conventions

Convention	Use
italic	Citations.
bold	Highlights interface elements, such as menu names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

1.2 Additional reading

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

Table 1-2: Arm publications

Document name	Document ID	Licensee only
Arm® Server Base System Architecture 7.0	DEN-0029F	No

Document name	Document ID	Licensee only
Arm® Architecture Reference Manual for A-profile Architecture	DDI 0487G.b (ID072021)	No
Arm® Generic Interrupt Controller Architecture Specification for GIC architecture version 3.0 and version 4.0	IHI 0069C (ID070116)	No
GICv3 and GICv4 Software Overview	DAI 0492	No

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
BDF	Bus, Device, and Function
ELx	Exception Level x (where x can be 0 to 3)
GCD	Grand Central Dispatch
GIC	Generic Interrupt Controller
ITS	Interrupt Translation Service
LPI	Locality-specific Peripheral Interrupt
MSI	Message-Signaled Interrupt
PAL	Platform Abstraction Layer
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PSCI	Power State Coordination Interface
SBSA	Server Base System Architecture
SMC	Secure Monitor Call
SoC	System on Chip
STS	Statistical Test Suite
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
PE	Verifies PE compliance.
GIC	Verifies GIC compliance.
Timer	Verifies PE timers and system timers compliance.
Watchdog	Verifies watchdog timer compliance.
PCIe	Verifies PCIe sub-system compliance.
Peripherals	Verifies USB, SATA, and UART compliance.
Power states	Verifies system power states compliance.
SMMU	Verifies SMMU sub-system compliance.
Exerciser	Verifies PCIe sub-system with a custom stimulus generator.
NIST	Verifies to determine the suitability of a generator for a cryptographic application.

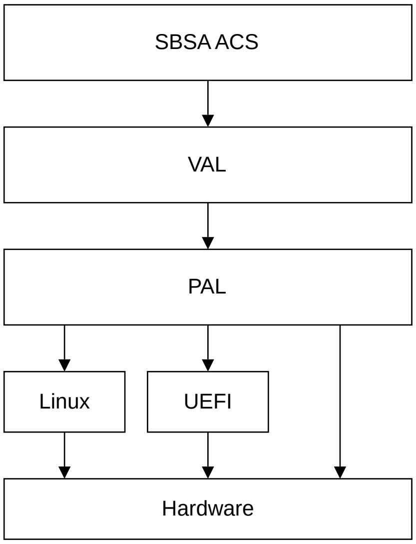
2.4 Layered software stack

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.

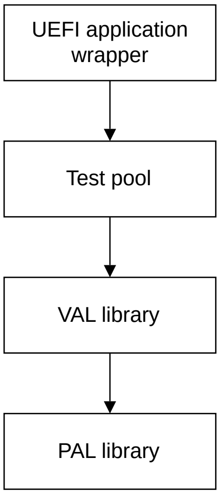
Table 2-3: Compliance test layers

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.
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

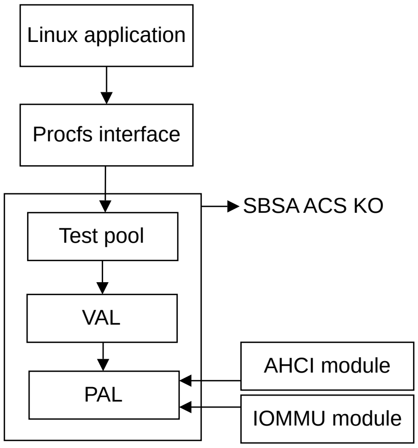


2.4.2 Compliance test software stack with Linux application

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 Coding guidelines

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

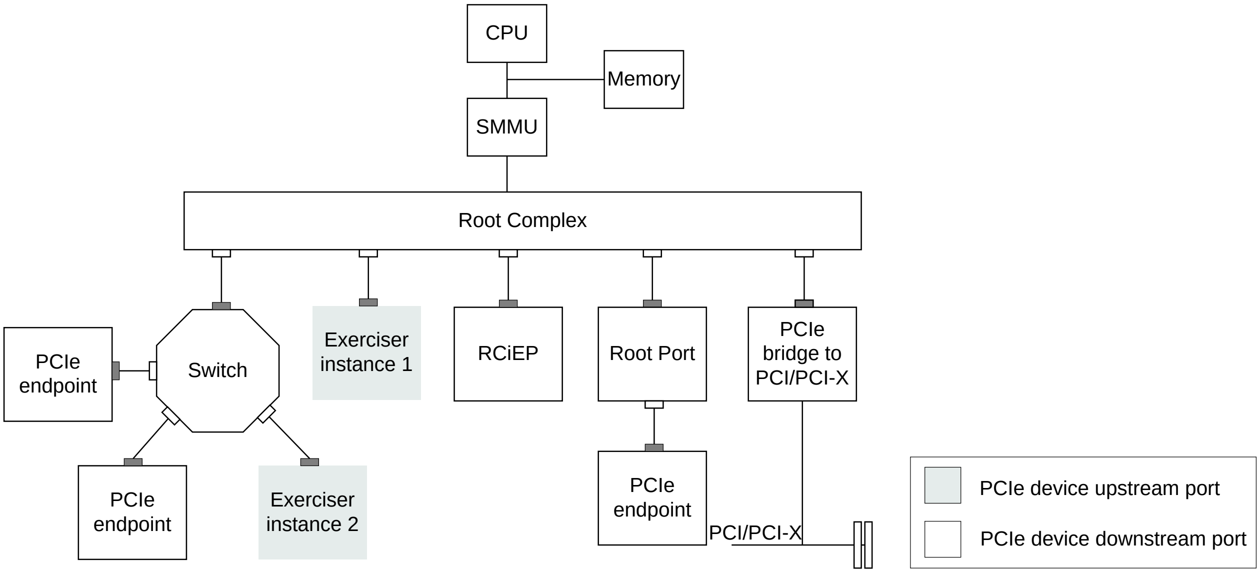
- All the tests call VAL APIs.
- VAL APIs might call PAL APIs depending on the requested functionality.
- A test does not directly interface with PAL functions.
- The test layer does not need any code modifications when porting from one platform to another.
- All the platform porting changes are limited 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 2-4: 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.

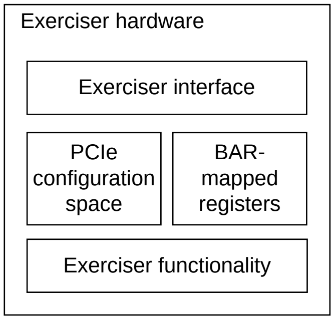
Note

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 2-5: Reference implementation of exerciser hardware

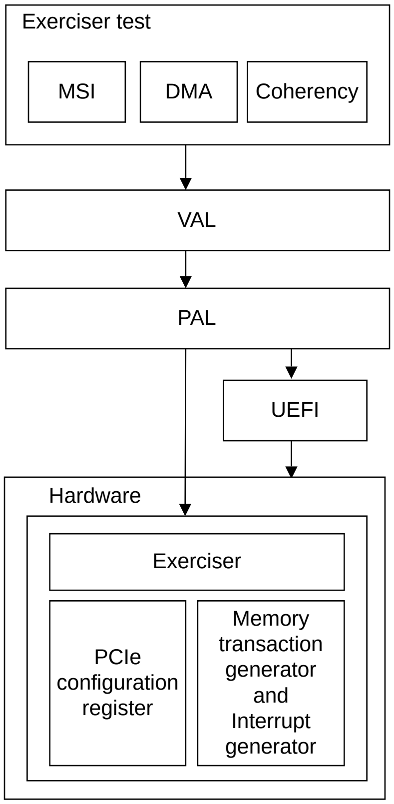


2.5.1 Compliance test software stack for exerciser with UEFI shell application

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 2-6: 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:

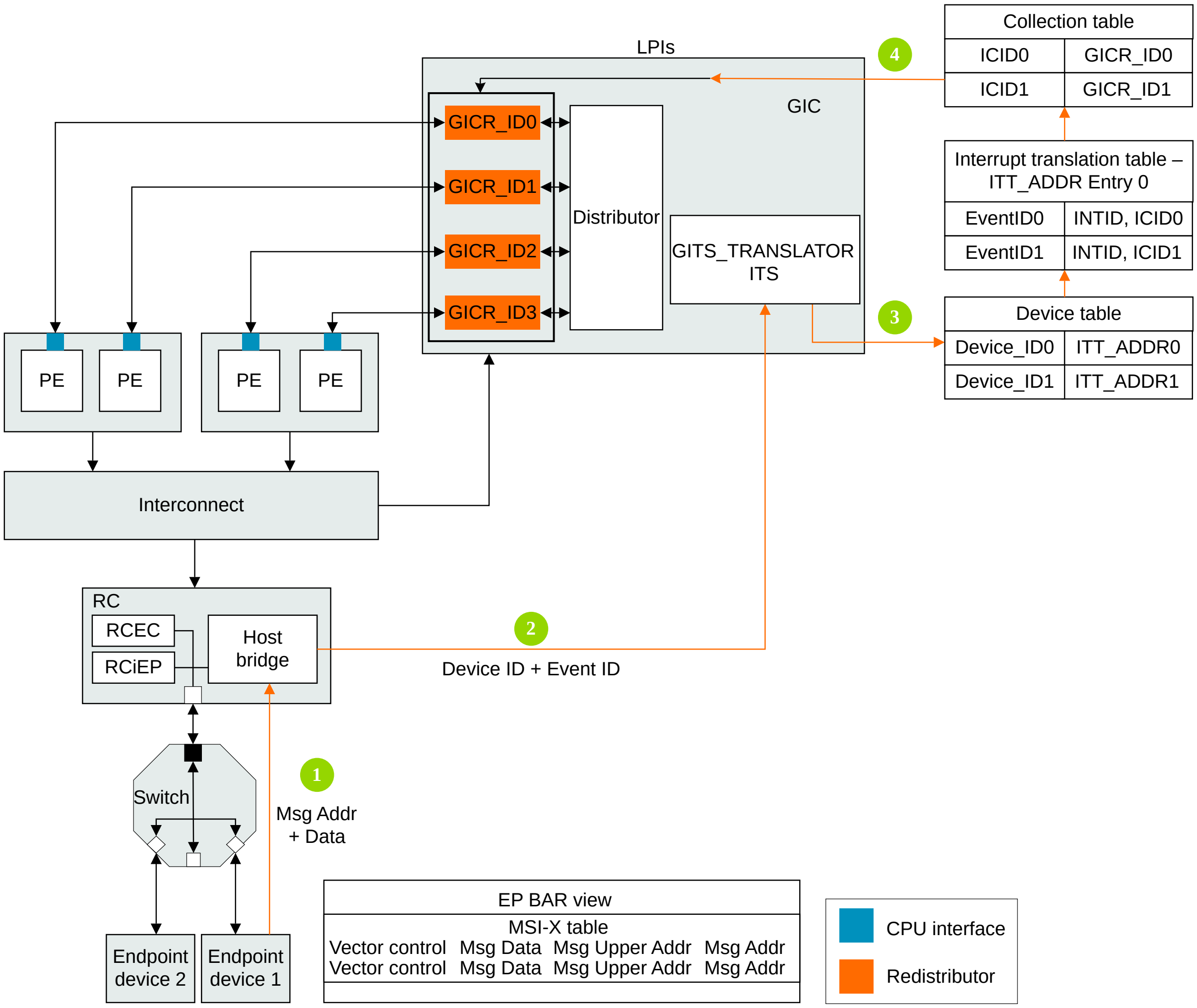
- The corresponding INTID for the input.
- The target Redistributor and, through this, the target PE for the INTID.

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.

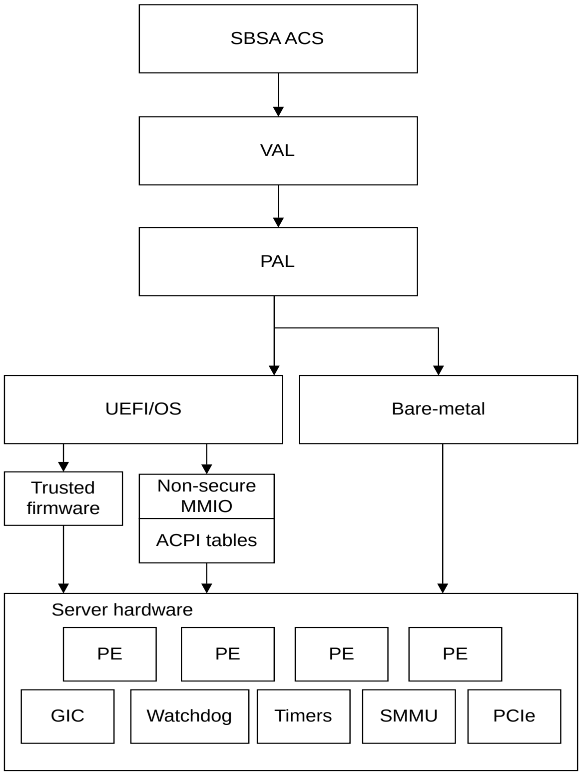
Figure 2-7: Routing MSI-X from Endpoint to PE through GIC ITS



2.7 Test platform abstraction

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

Figure 2-8: 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.
ACPI	Interface layer which provides platform-specific information, removing the need for the test suite to be ported for every platform.
Shared memory	Memory that is visible to all the PE and test peripherals.
Hardware	PE and controllers that are specified as part of the SBSA specification.

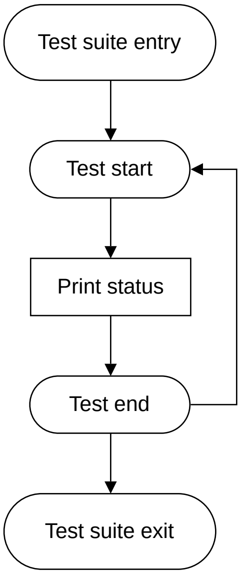
3. Execution flow control

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.

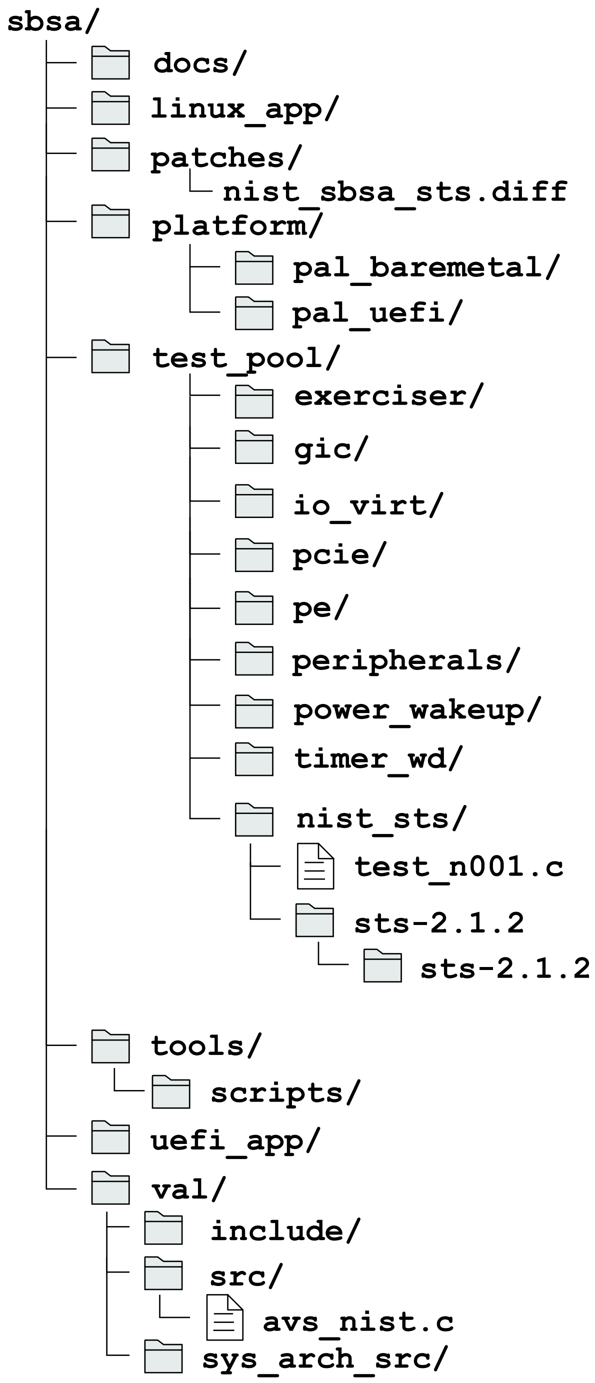
3.2 Test build and execution flow

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.

pal_uefi

Platform code targeting UEFI implementation.

pal_baremetal

Example PAL bare-metal reference code.

val

Common code that is used by the tests. Makes calls to PAL as necessary.

uefi_app

UEFI application source to call into the tests entry point.

test_pool

Test case source files for the test suite.

linux_app

Linux command-line executable source code.

docs

Documentation.

Tools

Consists of scripts written for this suite.

patches

Contains the SBSA NIST Statistical Test Suite (STS) patch.

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*.

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

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_execute_payload(ARM_SMC_ARGS *args);	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.

API name	Function prototype	Description
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.

Note

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

```
typedef struct {
UINT32 pe_num;      ///< PE Index
UINT32 attr;        ///< PE attributes
UINT64 mpidr;       ///< PE MPIDR
UINT32 pmu_gsiv;    ///< PMU Interrupt ID
}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. `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

API name	Function prototype	Description
set_intr_trigger	uint32_t pal_gic_set_intr_trigger (uint32_t int_id, INTR_TRIGGER_INFO_TYPE_e trigger_type);	Sets the trigger type to edge or level. `int_id`: interrupt ID which must be enabled and the service routine installed for `trigger_type`: interrupt trigger type edge or level

---

Note

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

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

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

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

4.2.4 Timer APIs

These APIs provide the information and functionality required by the test suite that accesses features of local and system timers, and watchdog timer.

Table 4-4: Timer APIs and their descriptions

API name	Function prototype	Description
create_info_table	void pal_timer_create_info_table(TIMER_INFO_TABLE *timer_info_table);	Abstracts the steps to discover and fill in the timer_info_table with information about the available local and system timers in the system.
wd_create_info_table	void pal_wd_create_info_table(WD_INFO_TABLE *wd_table);	Abstracts the steps to gather information about watchdogs in the platform and fills the wd_table.
get_counter_frequency	uint64_t pal_timer_get_counter_frequency(void);	This API gets the counter frequency value from the user.

Note

- This structure holds the timer-related information of the system. All the timer tests depend on the information in this structure.

```
typedef struct {
uint32_t s_el1_timer_flag;
uint32_t ns_el1_timer_flag;
uint32_t el2_timer_flag;
uint32_t el2_virt_timer_flag;
uint32_t s_el1_timer_gsiv;
uint32_t ns_el1_timer_gsiv;
uint32_t el2_timer_gsiv;
uint32_t virtual_timer_flag;
uint32_t virtual_timer_gsiv;
uint32_t el2_virt_timer_gsiv;
uint32_t num_platform_timer;
uint32_t num_watchdog;
uint32_t sys_timer_status;
}TIMER_INFO_HDR;
```

- This data structure contains information that is specific to system timer.

```
typedef struct {
uint32_t type;
uint32_t timer_count;
uint64_t block_cntl_base;
uint8_t frame_num[8];
uint64_t GtCntBase[8];
uint64_t GtCntEl0Base[8];
uint32_t gsiv[8];
uint32_t virt_gsiv[8];
uint32_t flags[8];
}TIMER_INFO_GTBLOCK;
```

- This data structure holds the watchdog information.

```
typedef struct {
uint64_t wd_ctrl_base;    ///< Watchdog Control Register Frame
uint64_t wd_refresh_base; ///< Watchdog Refresh Register Frame
uint32_t wd_gsiv;         ///< Watchdog Interrupt ID
uint32_t wd_flags;
}WD_INFO_BLOCK;
```

4.2.5 PCIe APIs

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

Table 4-5: 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.

API name	Function prototype	Description
io_read_cfg	uint32_t pal_pcie_io_read_cfg(uint32_t bdf, uint32_t offset, uint32_t *data);	Abstracts the configuration space read of a device identified by Bus, Device, and Function (BDF). This is used only in peripheral tests and need not be implemented in Linux. It returns either success or failure. `bdf`: PCI Bus, Dev, and Func `offset`: Offset in the configuration space from where data is to be read `data`: Stores the value read from the configuration space
io_write_cfg	void pal_pcie_io_write_cfg(uint32 bdf, uint32 offset, uint32 data)	Abstracts the configuration space write of a device identified by BDF (Bus, Device, and Function). Writes 32-bit data to the configuration space of the device at an offset. `bdf`: PCI Bus, Dev, and Func `offset`: Offset in the configuration space from where data is to be read `data`: Stores the value read from the configuration space
get_mcfg_ecam	uint64_t pal_pcie_get_mcfg_ecam();	Returns the PCI ECAM address from the ACPI MCFG table address.
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

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. `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.
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.

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. `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.
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

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.
		`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
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

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;
```

4.2.6 IO-Virt APIs

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

Table 4-6: 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.
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 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;
```

4.2.7 SMMU APIs

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

Table 4-7: 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.
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.

API name	Function prototype	Description
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.

4.2.8 Peripheral APIs

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

Table 4-8: Peripheral APIs and their descriptions

API name	Function prototype	Description
create_info_table	void pal_peripheral_create_info_table(PERIPHERAL_INFO_TABLE *per_info_table);	Abstracts the steps to gather information on all the peripherals present in the system and fills the information in the per_info_table.
memory_create_info_table	void pal_memory_create_info_table(MEMORY_INFO_TABLE *memoryInfoTable);	Fills in the MEMORY_INFO_TABLE with information about memory in the system. This is achieved by parsing the UEFI memory map. `peripheralInfoTable` : Address where the peripheral information must be filled. Returns none.
memory_ioremap	uint64_t pal_memory_ioremap(void *addr, uint32_t size, uint32_t attr);	Maps the memory region into the virtual address space. 64-bit address in virtual address space.
memory_unmap	void pal_memory_unmap(void *addr);	Unmaps the memory region which was mapped to the virtual address space.
memory_get_unpopulated_addr	uint64_t pal_memory_get_unpopulated_addr(uint64_t *addr, uint32_t instance);	Returns the address of unpopulated memory of the requested instance from Grand Central Dispatch (GCD) memory map. `addr`: Address of the unpopulated memory. `instance`: Instance of memory. Returns 0 for success.

API name	Function prototype	Description
get_legacy_irq_map	uint32_t pal_pcie_get_legacy_irq_map(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn, PERIPHERAL_IRQ_MAP *irq_map);	Returns the IRQ-mapping list for the legacy interrupts of a PCIe endpoint device. A possible way of returning this information is to query the _PRT method of the device ACPI namespace. `seg`: PCI segment number `bus`: PCI bus number `dev`: PCI device number `fn`: PCI function number `irq_map`: Pointer to IRQ map structure The following are the return values: 0: success. `irq_map` successfully retrieved in `irq_map` buffer. 1: unable to access the PCI bridge device of the input PCI device 2: unable to fetch the ACPI _PRT handle 3: unable to access the ACPI _PRT object 4: legacy interrupt out of range
is_device_behind_smmu	uint32_t pal_pcie_is_device_behind_smmu(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Checks if a device with the input BDF is behind an SMMU. One way of confirming this in Linux is to check if the `iommu_group` value of this device is non-zero. 1: device is behind SMMU 0: device is not behind SMMU or SMMU is in bypass mode
get_root_port	uint32_t pal_pcie_get_root_port_bdf(uint32_t *seg, uint32_t *bus, uint32_t *dev, uint32_t *func);	Returns the Bus, Device, and Function values of the Root Port of the device. The same function arguments are used to pass the input address of the device and also the output address of the Root Port. 0: success 1: input BDF device cannot be found 2: Root Port for the input device cannot be determined.
get_device_type	uint32_t pal_pcie_get_device_type(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Returns the PCIe device type of the input BDF. 0: Error: could not determine device structures 1: normal PCIe device 2: PCIe host bridge 3: PCIe bridge

API name	Function prototype	Description
get_snoop_bit	uint32_t pal_pcie_get_snoop_bit(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Returns if the snoop capability is enabled for the input device. 0: snoop capability is disabled 1: snoop capability is enabled 2: PCIe device is not found
get_dma_support	uint32_t pal_pcie_get_dma_support(uint32_t bus, uint32_t dev, uint32_t fn);	Returns if the PCIe device supports DMA capability or not. 0: DMA capability is not supported 1: DMA capability is supported 2: PCIe device is not found
is_devicedma_64bit	uint32_t pal_pcie_is_devicedma_64bit(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Returns the DMA addressability of the device. 0: does not support 64-bit transfers 1: supports 64-bit transfers
get_dma_coherent	uint32_t pal_pcie_get_dma_coherent(uint32_t bus, uint32_t dev, uint32_t fn);	Returns if the PCIe device supports coherent DMA. 0: DMA coherence is not supported 1: DMA coherence is supported 2: PCIe device is not found
is_pcie	uint32_t pal_peripheral_is_pcie(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Checks if PCIe device is PCI Express capable. 0: Not PCIe capable 1: PCIe capable

---

Note

This data structure captures the information about USB, SATA, and UART controllers. Also, information about all the PCIe devices present in the system is saved. This includes information such as PCIe bus, device, function, the BAR addresses, the IRQ map, and the MSI vector list if MSI is enabled.

```
/**
@brief Summary of Peripherals in the system
**/
typedef struct {
uint32_t num_usb;    ///< Number of USB Controllers
uint32_t num_sata;   ///< Number of SATA Controllers
uint32_t num_uart;   ///< Number of UART Controllers
uint32_t num_all;    ///< Number of all PCI Controllers}
PERIPHERAL_INFO_HDR;
/**
@brief Instance of peripheral info
**/
typedef struct {
PER_INFO_TYPE_e type; ///< PER_INFO_TYPE
uint32_t bdf;         ///< Bus Device Function
uint64_t base0;       ///< Base Address of the controller
uint64_t base1;       ///< Base Address of the controller
```

```
uint32_t irq;         ///< IRQ to install an ISR
uint32_t flags;
uint32_t msi;        ///< MSI Enabled
uint32_t msix;       ///< MSIX Enabled
uint32_t max_pasids;
}PERIPHERAL_INFO_BLOCK;
```

4.2.9 DMA APIs

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

Table 4-9: 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**.
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**.

API name	Function prototype	Description
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;
```

4.2.10 Exerciser

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

Table 4-10: 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 parameters 3 – MSI(x) attributes 4 – DMA attributes 5 – Peer-to-Peer attributes 6 – PASID attributes 7 – P2P_ATTRIBUTES 8 – PASID_ATTRIBUTES 9 – CFG_TXN_ATTRIBUTES 10 – ATS_RES_ATTRIBUTES 11 – TRANSACTION_TYPE 12 – NUM_TRANSACTIONS value2 is an optional argument and must be ignored for some configuration parameters.
get_param	uint32_t pal_exerciser_get_param(EXERCISER_PARAM_TYPE type, uint64_t *value1, uint64_t *value2, uint32_t instance)	Returns the requested configuration parameter values through 64-bit input arguments value1 and value2. The function returns a value of 1 to indicate read success and 0 to indicate read failure.
set_state	uint32_t pal_exerciser_set_state(EXERCISER_STATE state, uint64_t *value, uint32_t instance)	Sets the state of the PCIe stimulus generation hardware. The supported states include: 1 – RESET, hardware in reset state. 2 – ON, this state is set after hardware is initialized and is ready to generate stimulus. 3 – OFF, this state is set to indicate that hardware can no longer generate stimulus. 4 – ERROR, this state is set to signal an error with hardware.

API Name	Function prototype	Description
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
get_data	uint32_t pal_exerciser_get_data(EXERCISER_DATA_TYPE type, exerciser_data_t *data, uint32_t instance)	Returns either the configuration space or the BAR space information depending on the input argument type. The argument type can take one of the following two values: 1 – EXERCISER_DATA_CFG_SPACE 2 – EXERCISER_DATA_BAR0_SPACE
is_bdf_exerciser	uint32_t pal_is_bdf_exerciser(uint32_t bdf)	Checks if the device is an exerciser. Returns 1 if device is an exerciser, else 0.
get_ecsr_base	uint64_t pal_exerciser_get_ecsr_base(uint32_t Bdf,uint32_t BarIndex)	Returns the ECSR base address of a particular BAR Index.
get_pcie_config_offset	uint64_t pal_exerciser_get_pcie_config_offset(uint32_t Bdf)	Returns the configuration address of the given bdf.
start_dma_direction	uint32_t pal_exerciser_start_dma_direction(uint64_t Base, EXERCISER_DMA_ATTRDirection)	Triggers the DMA operation.
find_pcie_capability	uint32_t pal_exerciser_find_pcie_capability(uint32_t ID, uint32_t Bdf, uint32_t Value, uint32_t *Offset)	Returns 0 if the PCI capability is found.

4.2.11 Miscellaneous APIs

Miscellaneous APIs are described in the following table.

Table 4-11: Miscellaneous APIs and their descriptions

API name	Function prototype	Description
print	void pal_print(char *string, uint64_t data);	Sends a formatted string to the output console. `string`: An ASCII string. `data`: Data for the formatted output.
print_raw	void pal_print_raw(uint64_t addr, char *string, uint64_t data);	Sends a string to the output console without using the platform print function. This function gets COMM port address and directly writes to the address character by character. `addr`: Address to be written. `string`: An ASCII string. `data`: Data for the formatted output.
strncmp	pal_strncmp uint32_t pal_strncmp (char *FirstString, char *SecondString, uint32_t Length);	Compares two strings. Returns zero if strings are identical, else a nonzero value. `FirstString`: The pointer to the first null-terminated ASCII string. `SecondString`: The pointer to the second null-terminated ASCII string. `Length`The maximum number of ASCII characters for comparison.
mmio_read	uint32 pal_mmio_read(uint64 addr);	Provides a single point of abstraction to read from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 32-bit data read from the input address
mmio_read8	pal_mmio_read8(uint64 addr);	Provides a single point of abstraction to read 8-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 8-bit data read from the input address
mmio_read16	pal_mmio_read16(uint64 addr);	Provides a single point of abstraction to read 16-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 16-bit data read from the input address

API name	Function prototype	Description
mmio_read64	pal_mmio_read64(uint64 addr);	Provides a single point of abstraction to read 64-bit data from all memory-mapped I/O addresses. `addr`: 64-bit input address `return`: 64-bit data read from the input address
mmio_write	void pal_mmio_write(unit64 addr,uint32 data);	Provides a single point of abstraction to write to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 32-bit data to write to address
mmio_write8	pal_mmio_write8(unit64 addr,uint8 data);	Provides a single point of abstraction to write 8-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 8-bit data to write to address
mmio_write16	pal_mmio_write16(unit64 addr,uint16 data);	Provides a single point of abstraction to write 16-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 16-bit data to write to address
mmio_write64	pal_mmio_write(unit64 addr,uint64 data);	Provides a single point of abstraction to write 64-bit data to all memory-mapped I/O addresses. `addr`: 64-bit input address `data`: 64-bit data to write to address
mem_free_shared	pal_mem_free_shared(void);	Frees the allocated shared memory region.
mem_get_shared_addr	pal_mem_get_shared_addr(void);	Returns the base address of the shared memory region to the VAL layer.
mem_alloc	void pal_mem_alloc(unsigned int size);	Allocates memory of the requested size. `size`: size of the memory region to be allocated Returns virtual address on success and null on failure.
mem_calloc	void * pal_mem_calloc(uint32_t num, uint32_t Size);	Allocates requested buffer size in bytes with zeros in a contiguous memory and returns the base address of the range.
mem_allocate_shared	pal_mem_allocate_shared (uint32_t num_pe, uint32_t sizeofentry);	Allocates memory which is to be used to share data across PEs. `num_pe`: number of PEs in the system `sizeofentry`: size of memory region allocated to each PE Returns none.
mem_free	void pal_mem_free(void *buffer);	Frees the memory allocated by UEFI framework APIs. `buffer`: the base address of the memory range to be free

API name	Function prototype	Description
mem_cpy	void *pal_memcpy(void *dest_buffer, void *src_buffer, uint32_t len);: base address of the memory range to be freed	Copies a source buffer to a destination buffer and returns the destination buffer. `dest_buffer`: pointer to the destination buffer of the memory copy `src_buffer`: pointer to the source buffer of the memory copy len: number of bytes to copy from source buffer to destination buffer Returns the destination buffer.
mem_compare	uint32 pal_mem_compare(void *src, void *dest, uint32 len);	Compares the contents of the source and destination buffers. `src`: base address of the memory, source buffer to be compared `dest`: destination buffer to be compared with `len`: length of the comparison to be performed
mem_alloc_cacheable	void pal_mem_alloc_cacheable(uint32_t bdf, uint32_t size, void *pa);	Allocates cacheable memory of the requested size. `bdf`: BDF of the requesting PCIe device `size`: size of the memory region to be allocated `pa`: physical address of the allocated memory
mem_free_cacheable	void pal_mem_free_cacheable(uint32_t bdf, uint32_t size, void *va, void *pa);	Frees the cacheable memory allocated by Linux DMA Framework APIs. `bdf`: Bus, Device, and Function of the requesting PCIe device `size`: size of memory region to be freed `va`: virtual address of the memory to be freed `pa`: physical address of the memory to be freed
mem_virt_to_phys	void pal_mem_virt_to_phys(void *va);	Returns the physical address of the input virtual address. `va`: virtual address of the memory to be converted Returns the physical address.
time_delay_ms	uint64 pal_time_delay_ms (uint64 MicroSeconds);	Stalls the CPU for the specified number of microseconds. `MicroSeconds`: the minimum number of microseconds to be delayed Returns the value of the microseconds given as input.

API name	Function prototype	Description
mem_set	void pal_mem_set (void *buf, uint32 size, uint8 value);	A buffer with a known specified input value. `buf`: pointer to the buffer to fill `size`: number of bytes in the buffer to fill `value`: value to fill the buffer with
page_size	uint32_t pal_mem_page_size();	Returns the memory page size (in bytes) used by the platform.
alloc_pages	void* pal_mem_alloc_pages (uint32 NumPages);	Allocates the requested number of memory pages.
free_pages	void pal_mem_free_pages (void *PageBase, uint32_t NumPages);	Frees pages as requested.
phys_to_virt	void* pal_mem_phys_to_virt (uint64_t Pa);	Returns the VA of the input PA. `Pa`: Physical Address of the memory to be converted. Returns the VA.
target_is_bm	uint32_t pal_target_is_bm()	Checks if the system information is passed using bare-metal.
aligned_alloc	void *pal_aligned_alloc(uint32_t alignment, uint32_t size);	Allocates memory with the given alignment. `alignment`: Specifies the alignment. `size`: Requested memory allocation size. Returns pointer to the allocated memory with requested alignment.

4.2.12 NIST API

This API is used for randomness testing.

Table 4-12: NIST API and its description

API name	Function prototype	Description
generate_rng	uint32 pal_nist_generate_rng(UINT32 *rng_buffer);	Generates a 32-bit random number. `rng_buffer`: pointer to store the random data Returns success or failure.

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 11. • 3.2 Test build and execution flow on page 20. • 4.2.1 API naming convention on page 23. • 4.2.10 Exerciser on page 39.

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 11.
`pal_baremetal` folder is added to the directory structure.	See 3.2 Test build and execution flow on page 20.
Added a note about PAL bare-metal reference code.	See 4.1 Overview of PAL API on page 23.

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 14.
NIST STS information is updated in these topics.	See • 2.3 Compliance tests on page 11. • 3.2 Test build and execution flow on page 20. • 4.2 PAL API definitions on page 23.
APIs are added in all the modules.	See 4.2 PAL API definitions on page 23.
A new appendix about NIST STS is added.	See A. NIST Statistical Test Suite on page 45.

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 17.
GIC ITS PAL APIs are added to GIC APIs section.	See 4.2.3 GIC APIs on page 25.
SBSA ACS directory structure is updated.	See 3.2.1 Source code directory on page 20.

Change	Location
read_cfg and write_cfg APIs in the PCIe APIs table are updated.	See 4.2.5 PCIe APIs on page 27.
New configuration parameters are added to the Exerciser APIs set_param and ops.	See 4.2.10 Exerciser on page 39.
New APIs are added to Miscellaneous APIs section.	See 4.2.11 Miscellaneous APIs on page 40.

Table B-6: Differences between Issue 0301-01 and Issue 0302-01

Change	Location
Added new APIs	See: • 4.2.4 Timer APIs on page 26 • 4.2.5 PCIe APIs on page 27 • 4.2.11 Miscellaneous APIs on page 40
Enhancement changes	Applicable sections.