Arm® SBSA Architecture Compliance

Revision: r2p0

Validation Methodology

arm

101544_0200_04_en

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

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

Arm® SBSA Architecture Compliance Validation Methodology

Appendix B Revisions

Preface

This preface introduces the *Arm® SBSA Architecture Compliance Validation Methodology*.

It contains the following:

About this book

This book describes the architecture compliance validation methodology for Arm® SBSA architecture.

Product revision status

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

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

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

Intended audience

This book is written for engineers who are designing or verifying an implementation of the Arm® Server Base System Architecture.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an introduction to Arm SBSA Architecture Compliance Suite.

Chapter 2 Execution model and flow control
This chapter describes the execution model and the flow control used for SBSA ACS.

Chapter 3 Platform Abstraction Layer
This chapter provides an overview of PAL API and its categories.

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

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

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

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

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

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

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

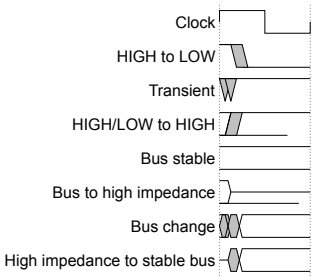

Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

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

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

Lowercase n

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

Additional reading

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

Arm publications

- *Arm® Server Base System Architecture Specification* (ARM-DEN-0029 Version 6.0).
- *Arm® Server Base Boot Requirements* (ARM-DEN-0044B).
- *Arm® Architecture Reference Manual ARMv8, for Armv8-A architecture profile* (ARM DDI 0487F.a (ID021920)).

Other publications
None.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

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

- The title *Arm SBSA Architecture Compliance Validation Methodology*.
- The number 101544_0200_04_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Chapter 1
Introduction

This chapter provides an introduction to Arm SBSA Architecture Compliance Suite.

It contains the following sections:

1.1 Abbreviations

The following table lists the abbreviations used in this document.

Table 1-1 Abbreviations and expansions

Abbreviation	Expansion
ACPI	Advanced Configuration and Power Interface
ELx	Exception Level x (where x can be 0 to 3)
GIC	Generic Interrupt Controller
NIST STS	National Institute of Standards and Technology Statistical Test Suite
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
UART	Universal Asynchronous Receiver and Transmitter
UEFI	Unified Extensible Firmware Interface
VAL	Validation Abstraction Layer

1.2 Server Base System Architecture ACS

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

The primary goal is to ensure enough standard system architecture to enable a suitably built single OS image to run on all hardware that is 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.

1.3 Compliance tests

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

The following table describes the compliance test components.

Table 1-2 Compliance test components

Components	Description
PE	Tests to verify PE compliance.
GIC	Tests to verify GIC compliance.
Timer	Tests to verify PE timers and system timers compliance.
Watchdog	Tests to verify watchdog timer compliance.
PCIe	Tests to verify PCIe subsystem compliance.
Peripherals	Tests to verify USB, SATA, and UART compliance.
Power states	Tests to verify system power states compliance.
SMMU	Tests to verify SMMU subsystem compliance.
Secure	Tests to verify Secure hardware.
Exerciser	Tests to verify PCIe subsystem with a custom stimulus generator.
NIST	Tests to determine the suitability of a generator for a cryptographic application.

1.4 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)

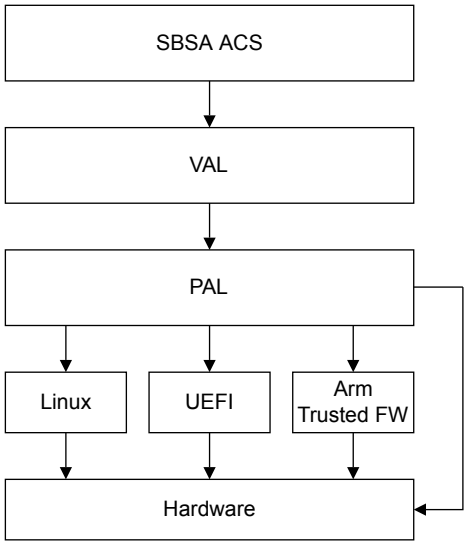

Figure 1-1 Layered software stack

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

Table 1-3 Compliance test layers

Layer	Description
Test suite	Collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL.
VAL	Provides a uniform view of all the underlying hardware and test infrastructure to the test suite.
PAL	Is a C-based, Arm-defined API 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.

This section contains the following subsections:

- *1.4.1 Compliance test software stack with UEFI application* on page 1-14.
- *1.4.2 Compliance test software stack with Linux application* on page 1-15.
- *1.4.3 Coding guidelines* on page 1-15.

1.4.1 Compliance test software stack with UEFI application

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

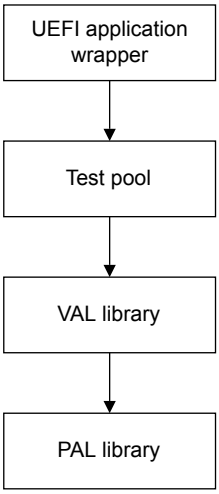

Figure 1-2 UEFI shell application

1.4.2 Compliance test software stack with Linux application

The following figure shows the compliance test software stack with Linux application as an example.

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.

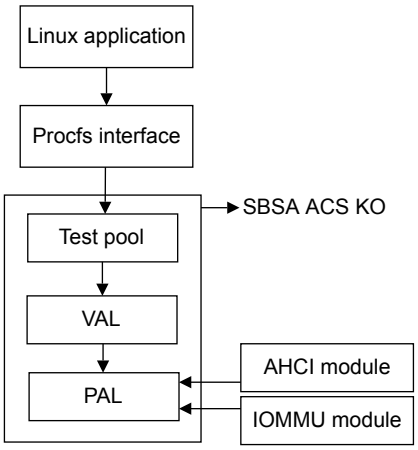

Figure 1-3 Linux application

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 together built 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*.

1.4.3 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 might require changes if there are architectural changes impacting multiple platforms.

1.5 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.

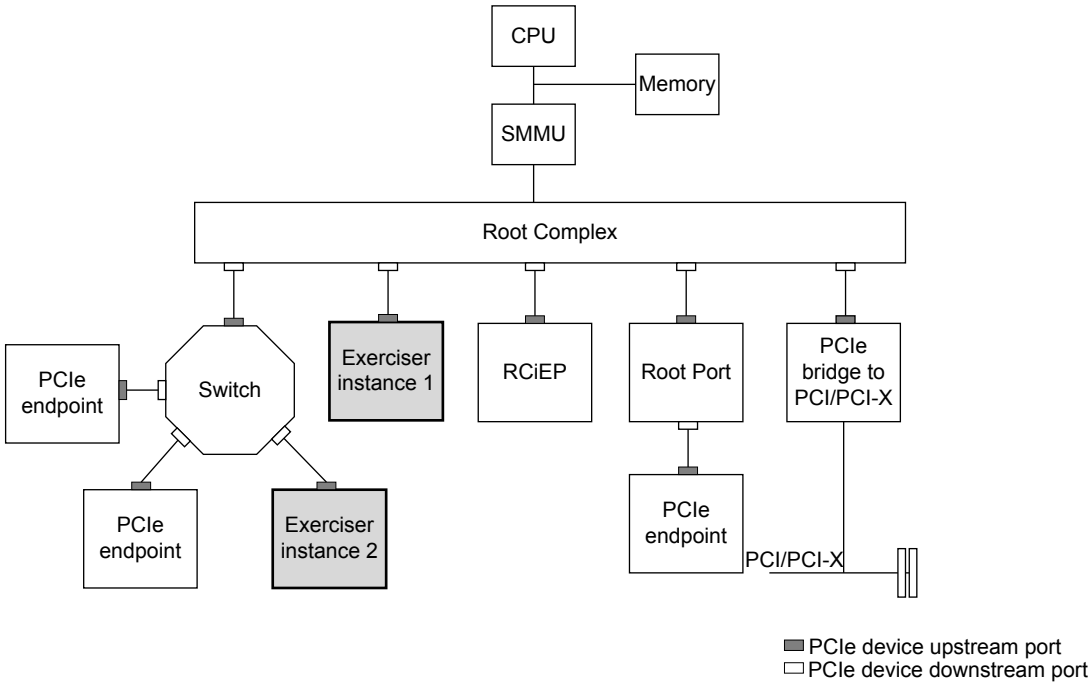

Figure 1-4 Exerciser in a SoC

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 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 instantiated. 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-specific. This can be done by providing a set of BAR-mapped registers as shown in the following figure, and writing specific values to them to trigger the necessary operations.

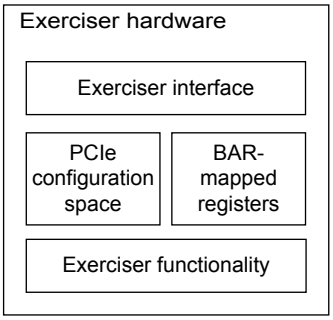

Figure 1-5 Reference implementation of exerciser hardware

1.5.1 Compliance test software stack for exerciser with UEFI shell application

The following figure shows the 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.

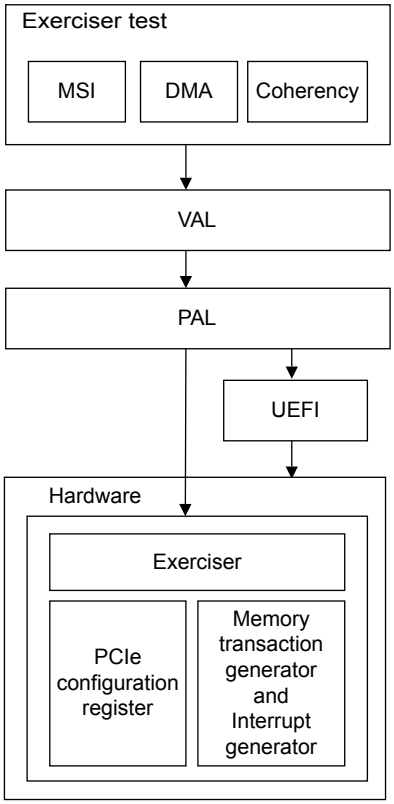

Figure 1-6 Exerciser with UEFI shell application

1.6 Test platform abstraction

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

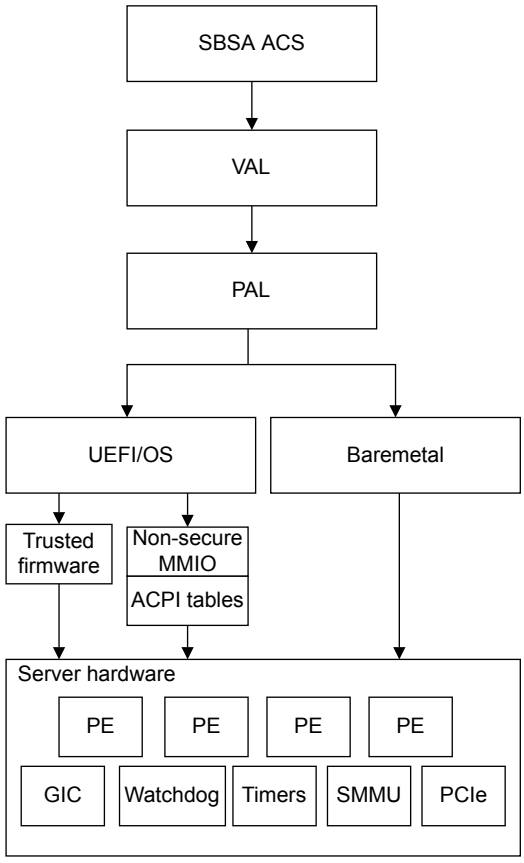

Figure 1-7 Test platform abstraction

The following table describes the SBSA abstraction terms.

Table 1-4 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 on a per platform basis.
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.

Chapter 2
Execution model and flow control

This chapter describes the execution model and the flow control used for SBSA ACS.

It contains the following sections:

2.1 Execution model and flow control

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

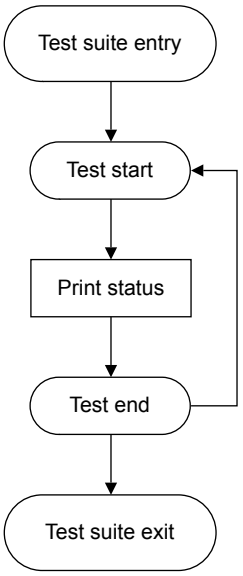

Figure 2-1 Execution model and flow control

The process that is followed for the flow control is:

1. The execution environment, like 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.

2.2 Test build and execution flow

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

This section contains the following subsections:

2.2.1 Source code directory

The following figure shows the source code directory for the SBSA ACS.

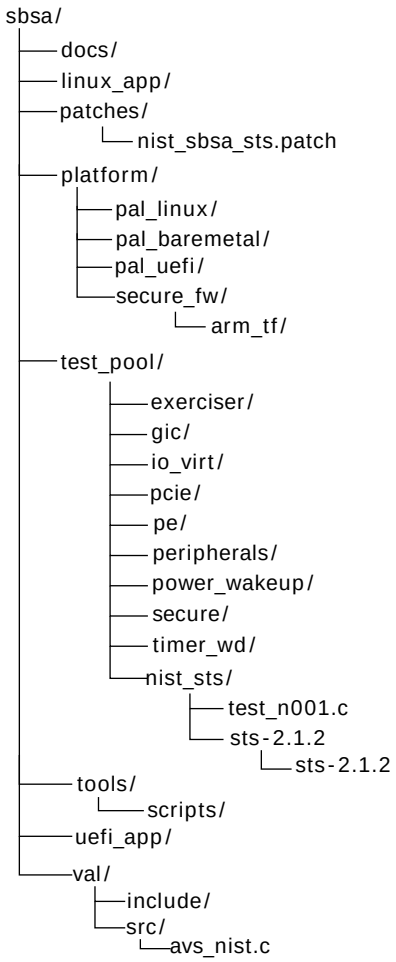

Figure 2-2 SBSA ACS directory structure

The following table describes all the directories.

Table 2-1 SBSA ACS directory structure description

Directory name	Description
`pal_uefi`	Platform code targeting UEFI implementation.
`pal_baremetal`	Example PAL bare-metal reference code.
`arm_tf`	Example of Arm Trusted Firmware code which must be integrated into the EL3 Secure firmware to run Secure tests.
`val`	Common code that is used by the tests. Makes calls to PAL as necessary.

Table 2-1 SBSA ACS directory structure description (continued)

Directory name	Description
`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.
`scripts`	Scripts written for this suite.
`patches`	Contains the SBSA NIST Statistical Test Suite (STS) patch.

2.2.2 Building the tests

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

Prerequisites

- To build SBSA ACS as a UEFI Shell application, a UEFI EDK2 source tree is required.
- To build the SBSA ACS kernel module, Linux kernel tree version 4.10 or above is required.

For details, see the *README*.

Test build for UEFI

The build steps for the compliance suite to be compiled as a UEFI shell application are available in the *README*. To execute the Secure tests, the EL3 firmware directory from the `platform/secure_sw` must be integrated into the platform-specific EL3 code base. As a reference implementation, the example code that is based on Arm Trusted Firmware is included as part of the ACS. The steps to port the reference implementation and build EL3 firmware are beyond the scope of this document.

Test build for OS-based tests

The build steps for the Linux application-driven compliance suite and SBSA ACS kernel module, which is a dependency for the SBSA ACS Linux application, are available in the *Arm® SBSA User Guide*.

Chapter 3
Platform Abstraction Layer

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

It contains the following sections:

3.1 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*

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.

3.2 PAL API definitions

The PAL API interface 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`.

This section contains the following subsections:

3.2.1 API naming convention

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

Table 3-1 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`

3.2.2 PE APIs

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

Table 3-2 PE APIs and their descriptions

API name	Function prototype	Description
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. For related definitions, see *Note*.
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 desired address.
get_esr	uint64_t pal_pe_get_esr(void *context);	Returns the exception syndrome from exception handler.
data_cache_ops_by_va	void pal_pe_data_cache_ops_by_va(uint64_t addr, uint32_t type);	Performs cache maintenance operation on an address.
get_far	uint64_t pal_pe_get_far(void *context);	Returns the FAR from exception handler.
install_esr	uint32_t pal_pe_install_esr(uint32_t exception_type, void (*esr)(uint64_t, void *));	Abstracts the exception handler installation steps. The input arguments are exception type and function pointer of the handler that has to be called when the exception of the given type occurs. It returns zero on success and nonzero 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;
```

3.2.3 GIC APIs

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

Table 3-3 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 subsystem 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
`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 info entry structure can hold information for any of the four types of GIC components. The four types of entries are:

```
typedef enum {
ENTRY_TYPE_CPUIF = 0x1000,
ENTRY_TYPE_GICD,
ENTRY_TYPE_GICRD,
ENTRY_TYPE_GICITS
}GIC_INFO_TYPE_e;
```

In addition to the type, each entry contains the base address of the component.

```
typedef struct {
uint32_t type;
uint64_t base;
}GIC_INFO_ENTRY;
```

3.2.4 Timer APIs

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

Table 3-4 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`.

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 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;
```

3.2.5 PCIe APIs

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

Table 3-5 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.
`read_cfg`	`uint32_t pal_pcie_read_cfg(uint32_t bdf, uint32_t offset, uint32_t *data);`	Abstracts the configuration space read of a device identified by BDF (Bus, Device, and Function). This is used only in peripheral tests and need not be implemented in Linux. It returns success or failure.
`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.
`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 address `dev`: PCI device address `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. `bus`: PCI bus address `dev`: PCI device address `fn`: PCI function number
`p2p_support`	`uint32_t pal_pcie_p2p_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Checks the PCIe device P2P support. `seg`: PCI segment number `bdf`: PCI Bus, Device, and Function Returns 1 if P2P feature is not supported and 0 if P2P feature is supported.

Table 3-5 PCIe APIs and their descriptions (continued)

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
`multifunction_support`	`uint32_t pal_pcie_multifunction_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);`	Checks the PCIe multifunction support. `bdf`: PCIe Bus, Device, and Function Returns 1 if multifunction feature is not supported and 0 if multifunction feature is supported.

`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.

Table 3-5 PCIe APIs and their descriptions (continued)

API name	Function prototype	Description
read_config_byte	void pal_pci_read_config_byte(uint32_t bdf, uint8_t offset, uint8_t *val);	Reads 1 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 1 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
read_msi_vector	void pal_pci_read_msi_vector (struct pci_dev *dev, struct msi_desc *entry, PERIPHERAL_VECTOR_BLOCK *vector);	Reads the MSI capability structure in PCIe configuration space. dev: PCI device structure entry: MSI description table vector: MSI controllers information structure

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 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;
```

3.2.6 IO-Virt APIs

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

Table 3-6 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 subsystem 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);	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 subsystem 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;
IOVIRT_BLOCK blocks[];
}IOVIRT_INFO_TABLE;
```

3.2.7 SMMU APIs

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

Table 3-7 SMMU APIs and their descriptions

API name	Function prototype	Description
check_device_iova	uint32_t pal_smmu_check_device_iova(void *port, uint64_t dma_addr);	Checks if the input DMA address belongs to the input device. This can be done by keeping track of the DMA addresses generated by the device using the start and stop monitor calls defined below or by reading the IOVA table of the device and looking for the input address. 0 is returned if address belongs to the device. Nonzero is returned if there are IMPLEMENTATION DEFINED error values.
device_start_monitor_iova	void pal_smmu_device_start_monitor_iova(void *port);	A hook to start the process of saving DMA addresses being used by the input device. It is used by the test to indicate the upcoming DMA transfers to be recorded and the test queries for the address through the check_device_iova call.
device_stop_monitor_iova	void pal_smmu_device_stop_monitor_iova(void *port);	Stops the recording of the DMA addresses being used by the input port.
max_pasids	uint32_t pal_smmu_max_pasids(uint64_t smmu_base);	Returns the maximum PASID value supported by the SMMU controller. For SMMUv3, this value can be read from the IDR1 register. 0 is returned when PASID support is not detected. Nonzero is returned if maximum PASID value supported for the input SMMU.
pa2iova	uint64 pal_smmu_pa2iova(uint64 SmmuBase, unit64 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.

Table 3-7 SMMU APIs and their descriptions (continued)

API name	Function prototype	Description
`smmu_disable`	`uint32 pal_smmu_disable(uint64 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_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`.
`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.

3.2.8 Peripheral APIs

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

Table 3-8 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`.
`get_legacy_irq_map`	`uint32_t pal_pcie_get_legacy_irq_map(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. 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 5: 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 checking 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

Table 3-8 Peripheral APIs and their descriptions (continued)

API name	Function prototype	Description
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
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 disabled 1: snoop capability enabled 2: PCIe device 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 not supported 1: DMA capability supported 2: PCIe device 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

Table 3-8 Peripheral APIs and their descriptions (continued)

API name	Function prototype	Description
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 not supported 1: DMA coherence supported 2: PCIe device not found
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.
is_pcie	uint32_t pal_peripheral_is_pcie(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Checks if PCI device is PCI Express capable. 0: Not PCIe capable 1: PCIe capable
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 needs to be filled. Returns none.

Note

This data structure captures the information about USB, SATA, and UART controllers. Additionally, 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;
```

3.2.9 DMA APIs

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

Table 3-9 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 nonzero 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 IMPLEMENTATION DEFINED: on error, the status is a nonzero value which is IMPLEMENTATION DEFINED.

Table 3-9 DMA APIs and their descriptions (continued)

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 nonzero 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 master for the last transaction. It is used by the test to verify if the address used by the DMA master was the same as what was 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. Nonzero: 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 need to 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;
```

3.2.10 Exerciser APIs

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

Table 3-10 Exerciser APIs and their descriptions

API Name	Function prototype	Description
`create_info_table`	`void pal_exerciser_create_info_table(EXERCISER_INFO_TABLE *exerciser_info_table);`	Abstracts the steps to gather information about all PCIe stimulus generation hardware in the system.
`get_info`	`uint32_t pal_exerciser_get_info(EXERCISER_INFO_TYPE type, uint32_t instance);`	Returns specific information of the requested instance.
`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 1: Peer-to-Peer attributes 1: PASID attributes `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.

Table 3-10 Exerciser APIs and their descriptions (continued)

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: NO_SNOOP_TLP_START
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

3.2.11 Miscellaneous APIs

Table 3-11 Miscellaneous APIs and their descriptions

API name	Function prototype	Description
mmio_read	uint32 pal_mmio_read(uint64 addr);	Provides a single point of abstraction to read from all memory-mapped I/O addresses. addr: 64-bit input address return: 32-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
mem_free_shared	pal_mem_free_shared(void);	Frees the shared memory region allocated.
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_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: base address of the memory range to be freed Returns none.
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.

Table 3-11 Miscellaneous APIs and their descriptions (continued)

API name	Function prototype	Description
mem_compare	uint32 pal_mem_compare(void *src, void *dest, uint32 len);	Compares the contents of the source and destination buffers. src: source buffer to be compared dest: destination buffer to be compared with len: length of the comparison to be performed
mem_alloc_coherent	void pal_mem_alloc_coherent(uint32_t bdf, uint32_t size, void *pa);	Allocates 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_coherent	void pal_mem_free_coherent(uint32_t bdf, uint32_t size, void *va, void *pa);	Frees the 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.
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

3.2.12 NIST API

This API is used for randomness testing.

Table 3-12 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.

It contains the following section:

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 (NIST 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. This package is obtained from *here* and downloaded automatically as part of the build process.

The updated version of the NIST STS tool for randomness testing is available *here*. 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 User Guide*. For details about NIST STS, see *https://doi.org/10.6028/NIST.SP.800-22r1a*.

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.

It contains the following section:

- *B.1 Revisions* on page Appx-B-49.

B.1 Revisions

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

Change	Location	Affects
Information about exerciser is added.	See the following sections: • *1.3 Compliance tests* on page 1-13 • *2.2 Test build and execution flow* on page 2-22 • *3.2.1 API naming convention* on page 3-26 • *3.2.10 Exerciser APIs* on page 3-41	All revisions

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

Change	Location	Affects
A note about exerciser is added.	See *1.3 Compliance tests* on page 1-13.	All revisions
`pal_baremetal` folder is added to the directory structure.	See *2.2 Test build and execution flow* on page 2-22.	All revisions
Added a note about PAL bare-metal reference code.	See *3.1 Overview of PAL API* on page 3-25.	All revisions

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

Change	Location	Affects
No technical changes.	-	-

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

Change	Location	Affects
A new section about exerciser is added.	See *1.5 Exerciser* on page 1-17.	All revisions.
NIST STS information is updated in these topics.	See • *1.3 Compliance tests* on page 1-13 • *2.2 Test build and execution flow* on page 2-22 • *3.2 PAL API definitions* on page 3-26	All revisions.
APIs are added in all the modules.	See *3.2 PAL API definitions* on page 3-26.	All revisions.
A new appendix about NIST STS is added.	See *Appendix A NIST Statistical Test Suite* on page Appx-A-46.	All revisions.