Arm® SBSA Architecture Compliance

Revision: r2p0

Validation Methodology

arm

101544_0200_02_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	RELv1.0
D	19 January 2018	Non-Confidential	Alpha release for RELv2.0
E	11 May 2018	Non-Confidential	RELv2.0
0200-01	27 December 2018	Non-Confidential	RELv2.1. The document now follows a new numbering format.
0200-02	26 April 2019	Non-Confidential	RELv2.2

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

http://www.arm.com

Contents

Arm® SBSA Architecture Compliance Validation Methodology

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
Read this chapter for an introduction to the SBSA ACS.

Chapter 2 Execution model and flow control
Read this chapter for a description of the execution model and the flow control used for SBSA ACS.

Chapter 3 Platform Abstraction Layer
Read this chapter for an overview of PAL API and its categories.

Appendix A 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.

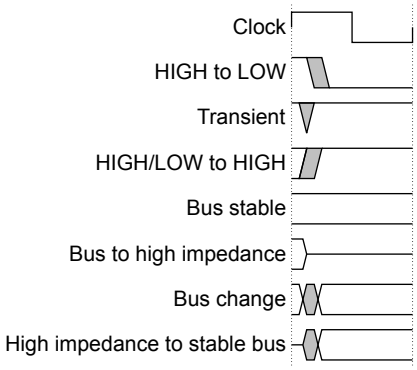

Figure 1 Key to timing diagram conventions

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 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 3.0).
- *Arm® Server Base Boot Requirements* (ARM-DEN-0044B).
- *Arm® Architecture Reference Manual ARMv8, for Armv8-A architecture profile* (ARM DDI 0487).

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 *errata@arm.com*. Give:

- The title *Arm SBSA Architecture Compliance Validation Methodology*.
- The number 101544_0200_02_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

Read this chapter for an introduction to the SBSA ACS.

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*
LPI	*Locality-specific Peripheral Interrupt*
PAL	*Platform Abstraction Layer*
PCIe	*Peripheral Component Interconnect Express*
PE	*Processing Element*
PPI	*Private Peripheral Interrupt*
PSCI	*Power State Coordination Interface*
SBSA	*Server Base Systems Architecture*
SMC	*Secure Monitor Call*
SoC	Sytem on Chip
SPI	*Shared Peripheral Interrupt*
UART	*Universal Asynchronous Receiver and Transmitter*
UEFI	*Unified Extensible Firmware Interface*
VAL	*Validation Abstraction Layer*
XSDT	*eXtended System Description Table*

1.2 Server Base System Architecture

Server Base System Architecture (SBSA) specification specifies hardware system architecture that is based on the 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.

Note

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 *System on Chip* (SoC). The stimulus is used in verifying the compliance of PCIe functions like IO coherency, snoop behavior, address translation, PASID transactions, MSI and legacy interrupt behavior.

1.4 Layered software stack

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

The constituents of the layered stack are:

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

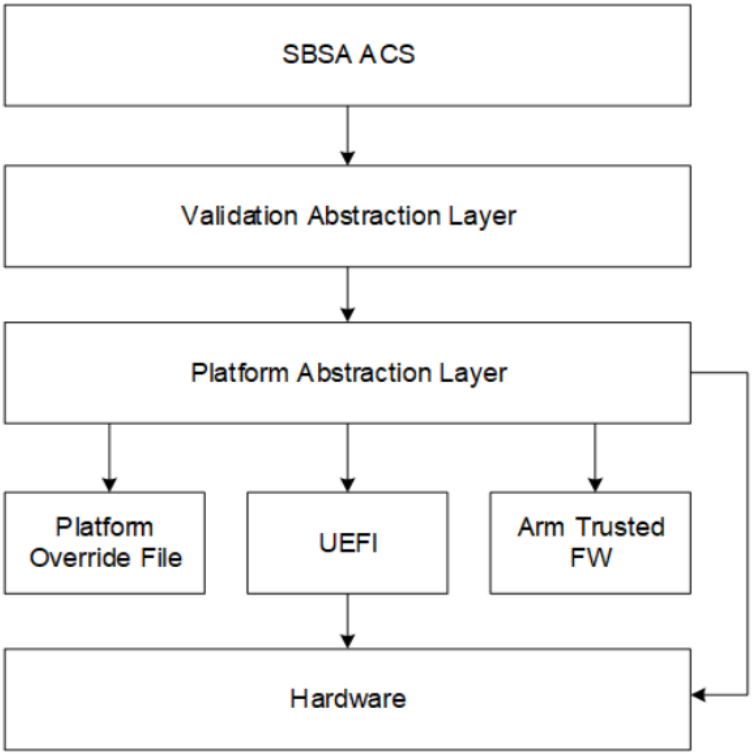

Figure 1-1 Layered software stack

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.

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

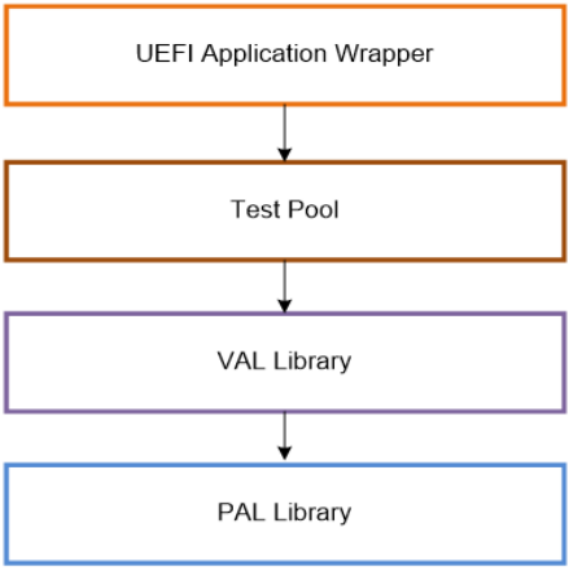

Figure 1-2 Compliance test software stack with UEFI shell application

Figure 1-3 shows the compliance test software stack with the 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.

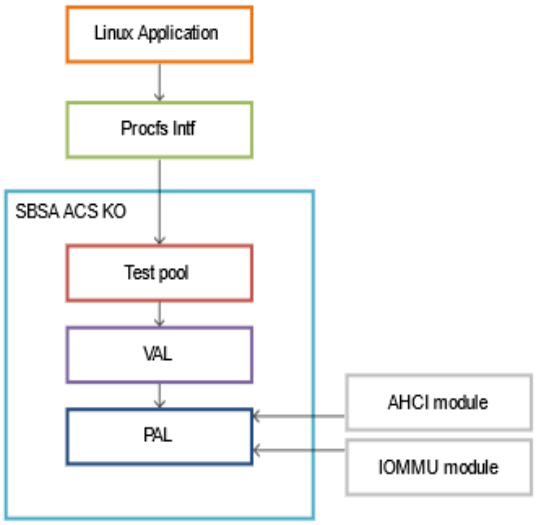

Figure 1-3 Compliance test software stack with Linux application

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 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*.

Coding guidelines

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

- All the tests call VAL APIs.
- VAL APIs might call PAL APIs depending on the functionality requested.
- 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 Test platform abstraction

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

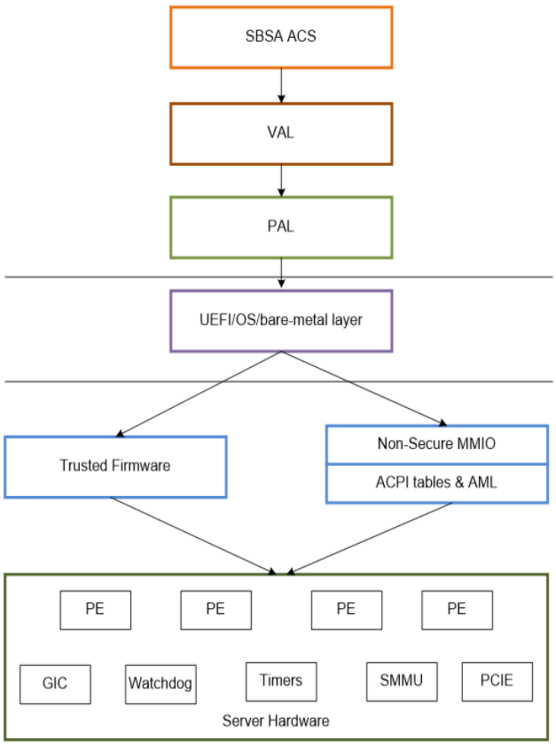

Figure 1-4 Test platform abstraction

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

Read this chapter for a description of 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.

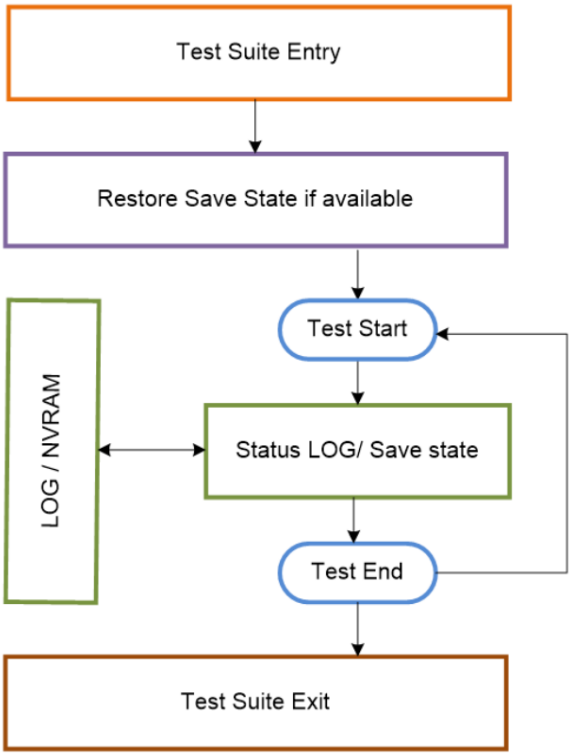

Figure 2-1 Execution model and flow control

The process that is followed for the flow control is:

1. The execution environment, like the UEFI shell, invokes the test entry point.
2. The test checks if there is a saved state. In UEFI, this might be provided by UEFI variables (Not implemented in this release).
3. If a saved state is found, then restore the saved state (Not implemented in this release).
4. Start the test iteration loop.
5. Save the state and report status during the test execution as required.
6. Reboot or put the system to sleep as required.
7. 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.

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*.

Source code directory

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

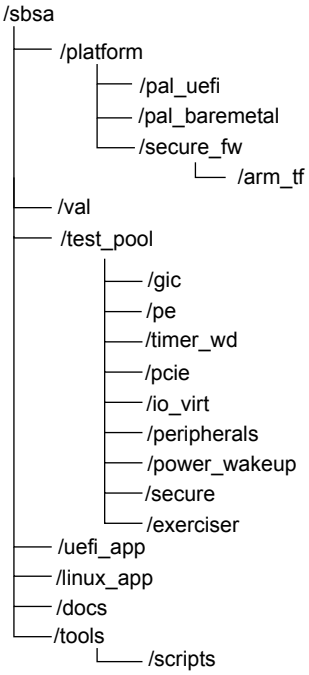

Figure 2-2 SBSA ACS directory structure

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 needed.
`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.

Test build for UEFI

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

EL3 Firmware

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 are detailed within the *User Guide*.

Linux kernel module

The build steps for the SBSA ACS kernel module, which is a dependency for the SBSA ACS Linux application, are also part of the user guide.

Chapter 3
Platform Abstraction Layer

Read this chapter for 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 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`
Test infrastructure	`print`, `mem`, `mmio`

3.2.2 PE

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 the 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 the appropriate parameters.
execute_payload	void pal_pe_execute_payload(ARM_SMC_ARGS *args);	Abstracts the PE wakeup and execute functionality. Ideally, this function should call 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 the exception type and function pointer of the handler to be called when the exception of the given type occurs. It returns zero on success and non-zero on failure.

Note

Each PE info 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

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 on the GIC subsystem and fills the 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.

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

This API provides the information and functionality required by the test suite that accesses features of local and system timers.

Table 3-4 Timer API and its description

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 info_table with the information on the available local and system timers in the system.

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 the information 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;
```

3.2.5 Watchdog

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

Table 3-5 Watchdog APIs and their descriptions

API name	Function prototype	Description
`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 fill the `info_table`.

Note

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;
```

3.2.6 PCIe

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

Table 3-6 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 fill 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, function). This is used only in peripheral tests and need not be implemented in Linux. It returns Success/Failure.
get_mcfg_ecam	uint64_t pal_pcie_get_mcfg_ecam();	Returns the PCI ECAM address from the ACPI MCFG Table address. It returns the PCI ECAM 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.

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;
```

3.2.7 IO-Virt

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

Table 3-7 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.

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;
```

3.2.8 SMMU

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

Table 3-8 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 on address since address belongs to the device. Non-zero is returned if there are implementation defined error values.
`device_start_monitor_iova`	`void pal_smmu_device_start_monitor_iova(void *port);`	A hook to start the process of saving DMA addresses being used by the input device. It is used by the test to indicate the upcoming DMA transfers need to be recorded and the test will query 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.

3.2.9 Peripheral

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

Table 3-9 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 fill 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 3-9 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 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 – The 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.

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

API name	Function prototype	Description
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 64bit transfers. 1 – Supports 64bit 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 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 – PCIe not capable. 1 – PCIe capable.

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;
```

3.2.10 DMA

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

Table 3-10 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` is the 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 non-zero value which is implementation 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.

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

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 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. Non-zero – Error, ignore the attribute and shareability parameters.

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;
```

3.2.11 Exerciser

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

Table 3-11 Exerciser APIs and 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.

Table 3-11 Exerciser APIs and descriptions (continued)

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 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.
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.

Table 3-11 Exerciser APIs and descriptions (continued)

API Name	Function prototype	Description
ops	`uint32_t pal_exerciser_ops(EXERCISER_OPS ops, uint64_t param, uint32_t instance)`	Abstracts the steps to implement the requested operation on the PCIe stimulus generation hardware. Following are the supported operations: 1 – START_DMA, 2 – GENERATE_MSI 3 – GENERATE_L_INTR 4 – MEM_READ 5 – MEM_WRITE 6 – CLEAR_INTR 7 – PASID_TLP_START 8 – PASID_TLP_STOP 9 – 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

Appendix A
Revisions

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

It contains the following section:

- *A.1 Revisions* on page Appx-A-38.

A.1 Revisions

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

Change	Location	Affects
Added information about exerciser.	See the following sections: • *1.3 Compliance tests* on page 1-12 • *2.2 Test build and execution flow* on page 2-19 • *3.2.1 API naming convention* on page 3-23 • *3.2.11 Exerciser* on page 3-34	All revisions

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

Change	Location	Affects
Added a note about exerciser.	See *1.3 Compliance tests* on page 1-12.	All revisions
Added `pal_baremetal` folder to the directory structure.	See *2.2 Test build and execution flow* on page 2-19.	All revisions
Added a note about PAL bare-metal reference code.	See *3.1 Overview of PAL API* on page 3-22.	All revisions