# Arm® SBSA Architecture Compliance

## Validation Methodology

**Non-Confidential – DEV2.0**



PJDOC-2042731200-3437

# Arm® SBSA Architecture Compliance
## Validation Methodology



### Release Information

**Document History**

| Date | Confidentiality | Change |
|---|---|---|
| 30 Nov 2016 | Non-Confidential | Release for alpha |
| 31 Mar 2017 | Non-Confidential | Release for beta |
| 13 Jul 2017 | Non-Confidential | Release for REL v1.0 |
| 19 Jan 2018 | Non-Confidential | Alpha release for REL v2.0 |
| 11 May 2018 | Non-Confidential | Release for DEV2.0 |

### 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

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 sections:

# About this book

This book describes the Architecture Compliance Validation Methodology for the 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 specifying, designing, or verifying an implementation of the Arm® SBSA architecture.

## Using this book

This book is organized into the following chapters:

***Chapter 1 Introduction***

Read this for an overview of SBSA validation methodology.

***Chapter 2 Execution model and flow control***

Read this for details on the execution control mode and flow control scheme used by the compliance and for details on the different test platform variants and ideas on implementing the PAL layer.

***Chapter 3 Platform Abstraction Layer***

Read this for details on the PAL APIs and their categories.

### 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.

monospace

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

*monospace italic*

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

**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.

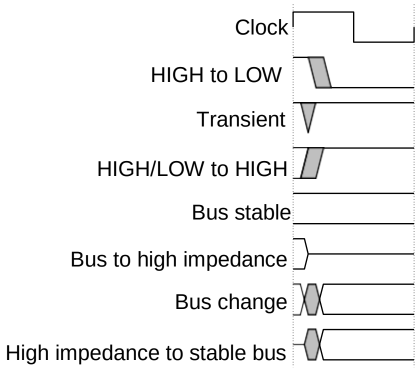

**Figure 1 Key to timing diagram conventions**

### 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

- *Server Base System Architecture (ARM-DEN-0029 Version 3.0)*
- *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 support-enterprise-acs@arm.com.

Give:

- The title *Arm® SBSA Architecture Compliance Suite Validation Methodology*.
- The number PJDOC-2042731200-3437
- 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 introduces the Arm® SBSA Architecture Compliance Suite.

It contains the following sections:

## 1.1 Overview of SBSA

*Server Base Systems 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 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 SBSA 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 described behavior of the SBSA Architecture Specification.

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

## 1.2 Acronyms and terminology

The following acronyms are used in this document.

**Table 1-1 Acronyms and expansions**

| Acronym | Expansion |
|---|---|
| ACPI | Advanced Configuration and Power Interface |
| ELx | Exception Level ‘x’ where x can be 0 to 5 |
| 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 |
| SPI | Shared Peripheral Interrupt |
| UART | Universal Asynchronous Receiver and Transmitter |
| UEFI | Unified Extensible Firmware Interface |
| VAL | Validation Abstraction Layer |
| XSDT | eXtended System Description Table |

## 1.3 Compliance tests

SBSA compliance tests are self-checking, portable C-based tests with directed stimulus. The following table describes the components of the compliance test with the description.

**Table 1-2 Components of the compliance test**

| Components | Description |
|---|---|
| PE | Tests to verify PE compliance |
| GIC | Tests to verify GIC compliance |
| Timers | Tests to verify PE timers and System timers compliance |
| Watchdog | Tests to verify Watchdog 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 |

## 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
- VAL
- PAL

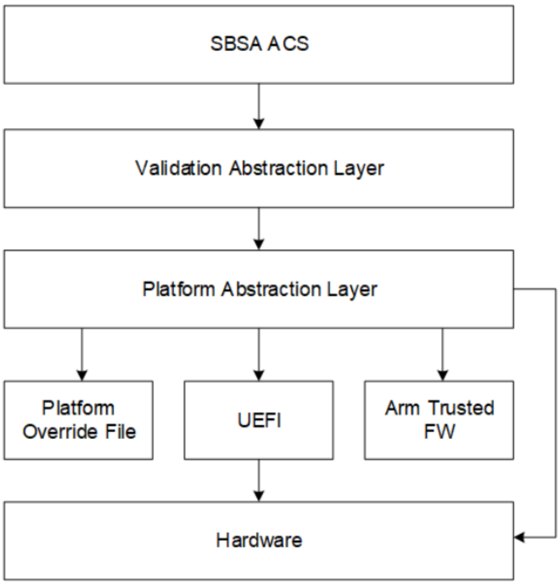


**Figure 1-1 Layered software stack for compliance suite**

The following table describes the different layers of compliance test.

**Table 1-3 Layers of compliance test**

| 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 layer. |
| VAL | Provides a uniform view of all the underlying hardware and test infrastructure to the test suite. |
| PAL | Abstracts features whose implementation varies from one target system to another. The PAL is a C-based API defined by Arm 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 bare-metal abstraction. |

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

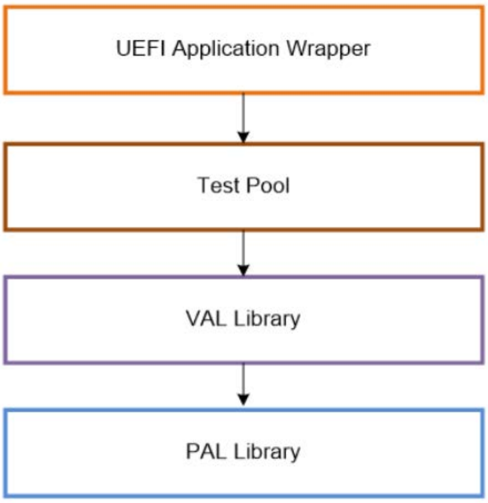


**Figure 1-2 Compliance test software-stack with UEFI shell application**

Figure 1-3 shows the compliance test software stack with Linux Application as an example.

The stack is spread across user-mode and kernel-mode space. The Linux command-line application running in 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.

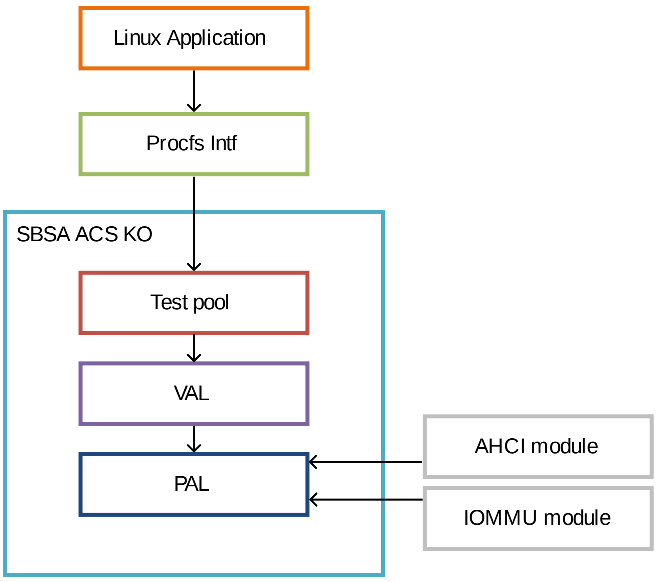


**Figure 1-3 Compliance test software stack with Linux application**

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 kernel and user mode, 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 more information, 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 layer APIs.
- VAL layer APIs might call PAL layer APIs depending on the functionality requested.
- A test does not directly interface with PAL layer 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 layer.
- The VAL layer 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.

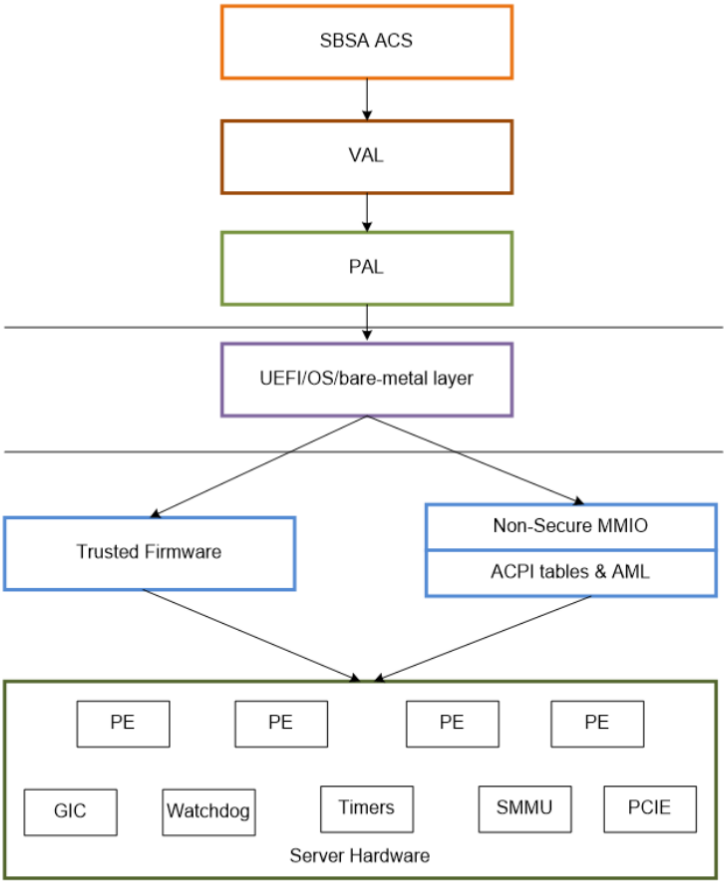


**Figure 1-4 Test platform abstraction**

The following table describes the SBSA abstraction terms with description.

**Table 1-4 Abstraction terms and descriptions**

| Abstraction | Description |
|---|---|
| UEFI /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 which is visible to all the PE and test peripherals. |
| Hardware | PE and controllers which 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 Architecture Compliance Suite.

It contains the following sections:

## 2.1 Execution model and flow control

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

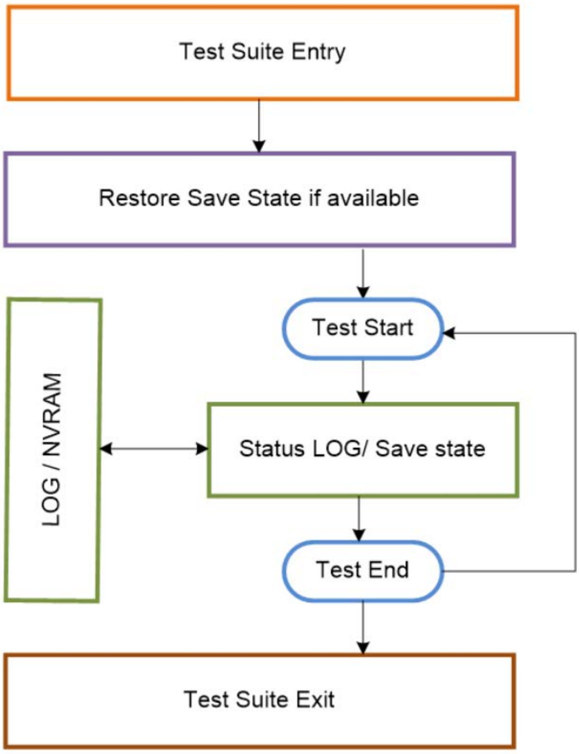


**Figure 2-1 Execution model and flow control**

The process that is followed for the flow control is as described:

1. The execution environment, for example, 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 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

### 2.2.1 Prerequisites

To build the SBSA compliance suite as a UEFI Shell application, a UEFI EDK2 source tree is required.

To build the SBSA Architecture Compliance Suite kernel module, Linux Kernel tree version 4.10 or above is required.

For more details, see [https://github.com/ARM-software/sbsa-acs/blob/master/README.md](https://github.com/ARM-software/sbsa-acs/blob/master/README.md)

### 2.2.2 Source code directory

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

```
/sbsa
├── /platform
│     ├── /pal_uefi
│     └── /secure_fw
│           └── /arm-tf
├── /val
├── /test_pool
│     ├── /gic
│     ├── /pe
│     ├── /timer_wd
│     ├── /pcie
│     ├── /io_virt
│     ├── /peripherals
│     ├── /power_wakeup
│     └── /secure
├── /uefi_app
├── /linux_app
├── /docs
└── /tools
      └── /scripts
```

**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 |
| arm-tf | Arm trusted firmware example 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 test suite |
| linux_app | Linux command-line executable source code |
| docs | Documentation |
| scripts | Scripts written for this suite. |

### 2.2.3 Test build - UEFI

#### UEFI Shell Application

The build steps for the compliance suite to be compiled as a UEFI shell application are at https://github.com/ARM-software/sbsa-acs/blob/master/README.md

#### 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 Architecture compliance suite.

The steps to port the reference implementation and build EL3 firmware are beyond the scope of this document.

### 2.2.4 Test build – OS-based tests

#### Linux Application

The build steps for the Linux application driven compliance suite are detailed within the User Guide at https://github.com/ARM-software/sbsa-acs/tree/master/docs.

#### 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

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

It contains the following sections:

## 3.1 PAL API

The PAL is a C-based API defined by Arm 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 could also be bare-metal code.

The reference PAL implementation for UEFI is located at https://github.com/ARM-software/sbsa-acs/tree/master/platform/pal_uefi

The reference PAL implementation for Linux is located at http://www.linux-arm.org/git?p=linux-acs.git;a=tree;f=sbsa-acs-drv/files/platform/pal_linux;h=b313763e4aed7b17963659a856773a00f28c02da;hb=HEAD

## 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`.

### 3.2.1 API Naming convention

The PAL API interface *<module>* name is mapped as showing in the following table.

**Table 3-1 Module 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 |
| Test infrastructure | Print, mem, mmio etc. |

### 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.<br>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 the 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 that processing of interrupt is complete by writing to 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

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

**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 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 would read 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 config space read of a device identified by bdf (bus, device, function). Used only in Peripheral tests, 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.<br>0 indicates a fail since the mapping is not unique.<br>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 given SMMU node has unique context bank interrupt ids.<br>O 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 sub-system of the SOC.

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

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

### 3.2.8 SMMU

These functions abstract information 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.<br>0 is returned on address since address belongs to the device.<br>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 SMMU v3, this value can be read from the IDR1 register.<br><br>0 is returned when PASID support is not detected. Non-zero is returned if Max Pasid value supported for the input SMMU |

### 3.2.9 Peripheral

These functions abstract information 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. One possible way of returning this information is to query the _PRT method of the device ACPI namespace. The following are the return values:<br>0 – Success. irq_map successfully retrieved in irq_map buffer.<br>1 – Unable to access the PCI bridge device of the input PCI device.<br>2 – Unable to fetch the ACPI _PRT handle.<br>3 – Unable to access the ACPI _PRT object.<br>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.<br>1 – Device is behind SMMU.<br>0 - Device is not behind SMMU or SMMU is in bypass mode. |
| get_root_port | uint32_t pal_pcie_get_root_port_bdf(uint32_t *seg, uint32_t *bus, uint32_t *dev, uint32_t *func); | Returns the Bus-device-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.<br>0 – Success.<br>1 – Input BDF device cannot be found.<br>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.<br>0 – Error – Could not determine device structures<br>1– Normal PCIe Device.<br>2– PCIe Host Bridge.<br>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.<br>0– Snoop capability disabled.<br>1– Snoop capability enabled.<br>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.<br>0 – DMA capability not supported.<br>1 – DMA capability supported.<br>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.<br>0 – Does not support 64bit transfers.<br>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.<br>0 – DMA coherence not supported.<br>1 – DMA coherence supported.<br>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.<br>0–PCie not capable.<br>1– PCIe capable. |

**Note**

This data structure captures the information about USB, SATA and UART controllers. In addition to this 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 if MSI is enabled, the MSI vector list.

```
/**
  @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 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.<br>dma_target_buf is the target physical address in the memory where the DMA data is to be written.<br>0 – Success.<br>Implementation defined – On error, the status is a non-zero value which is implementation defined. |
| start_to_device | uint32_t pal_dma_start_to_device(void *dma_source_buf, uint32_t length, void *host, void *target, uint32_t timeout); | Abstracts the functionality of performing a DMA operation to the device from DDR memory.<br>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.<br>0 – success<br>Implementation defined - On error, the status is a non-zero value which is implementation defined. |
| | uint64_t pal_dma_mem_alloc(void **buffer, uint32_t length, void *dev, uint32_t flags); | Allocates contiguous memory for DMA operations.<br>Supported values for flags are<br>1 – DMA_COHERENT<br>2 – DMA_NOT_COHERENT<br>dev is a void pointer which can be used by the PAL layer to get the context of the request. This is same value that is returned by PAL during info table creation.<br>0 – Success.<br>Implementation defined – On error, the status is a non-zero value which is implementation defined. |

| | | |
|---|---|---|
| | 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. |
| dma_mem_get_attr | int pal_dma_mem_get_attrs(void *buf, uint32_t *attr, uint32_t *sh) | Returns the memory and share-ability attributes of the input address. The attributes are returned as per the MAIR definition in the Arm ARM VMSA section.<br><br>0 –Success.<br>Non-zero – Error, ignore the attribute and share-ability 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, targets connected to the port etc.
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;
```