

# Arm® Base System Architecture Compliance

Revision: r0p0

## Validation Methodology



# Arm® Base System Architecture Compliance

## Validation Methodology

Copyright © 2021 Arm Limited or its affiliates. All rights reserved.

## Release Information

## Document History

Issue	Date	Confidentiality	Change
0000-01	12 May 2021	Non-Confidential	Alpha release for RELv0.5

## Non-Confidential Proprietary Notice

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

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

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 for an Alpha product, that is a product under development.

### **Web Address**

[developer.arm.com](https://developer.arm.com)

### **Progressive terminology commitment**

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

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

If you find offensive terms in this document, please contact [terms@arm.com](mailto:terms@arm.com).

# Contents

## Arm® Base System Architecture Compliance Validation Methodology

### **Preface**

About this book .....	7
Feedback .....	9

### **Chapter 1**

#### **Introduction to BSA**

1.1	Abbreviations .....	1-11
1.2	Introduction to BSA ACS .....	1-12
1.3	Compliance tests .....	1-13
1.4	Layered software stack .....	1-14
1.5	Exerciser .....	1-17
1.6	Compliance test software stack for exerciser with UEFI shell application .....	1-19
1.7	Test platform abstraction .....	1-20

### **Chapter 2**

#### **Execution model and flow control**

2.1	Execution model and flow control .....	2-22
2.2	Test build and execution flow .....	2-23

### **Chapter 3**

#### **Platform Abstraction Layer**

3.1	Overview of PAL API .....	3-26
3.2	API definitions .....	3-27

A.1    *Revisions* ..... Appx-A-54

# Preface

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

It contains the following:

- *About this book* on page 7.
- *Feedback* on page 9.

## About this book

This book describes Architecture Compliance Validation Methodology for Arm® Base System Architecture.

## Using this book

This book is organized into the following chapters:

### **Chapter 1 Introduction to BSA**

This chapter provides an overview on Arm BSA Architecture Compliance Validation Methodology.

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

This chapter provides an overview of the execution model and the flow control used for BSA ACS.

### **Chapter 3 Platform Abstraction Layer**

This chapter provides 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.

`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.

#### Other information

- [\*Arm® Developer\*](#).
- [\*Arm® Documentation\*](#).
- [\*Technical Support\*](#).
- [\*Arm® Glossary\*](#).



## 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-systemready-accs@arm.com](mailto:support-systemready-accs@arm.com). Give:

- The title *Arm Base System Architecture Compliance Validation Methodology*.
- The number 102503\_0000\_01\_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

————— **Note** —————

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

---

# Chapter 1

## Introduction to BSA

This chapter provides an overview on Arm BSA Architecture Compliance Validation Methodology.

It contains the following sections:

- *1.1 Abbreviations* on page 1-11.
- *1.2 Introduction to BSA ACS* on page 1-12.
- *1.3 Compliance tests* on page 1-13.
- *1.4 Layered software stack* on page 1-14.
- *1.5 Exerciser* on page 1-17.
- *1.6 Compliance test software stack for exerciser with UEFI shell application* on page 1-19.
- *1.7 Test platform abstraction* on page 1-20.

## 1.1 Abbreviations

The section lists the abbreviations used in this document.

**Table 1-1 Abbreviations and expansions**

Abbreviation	Expansion
ACS	Architecture Compliance Suite
ACPI	Advanced Configuration and Power Interface
BSA	Base System Architecture
DT	Device Tree
ELx	Exception Level x (where x can be 0 to 3)
GIC	Generic Interrupt Controller
ITS	Interrupt Translation Service
LPI	Locality-specific Peripheral Interrupt
MSI	Message-Signaled Interrupt
PAL	Platform Abstraction Layer
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PSCI	Power State Coordination Interface
RCiEP	Root Complex integrated EndPoint
RCEC	Root Complex Event Collector
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 Introduction to BSA ACS

This section provides information on BSA ACS and its features.

*Base System Architecture (BSA) Architecture Compliance Suite (ACS)* 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 BSA architecture that is described in the *Arm® Base System Architecture Specification* defines the behavior of an abstract machine, referred to as a BSA system. Implementations compliant with the BSA architecture must conform to the behavior described in the specification.

The 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

This section provides information on BSA compliance tests.

BSA compliance tests are self-checking and 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 and Wakeup	Tests to verify system power states compliance.
SMMU	Tests to verify SMMU subsystem compliance.
Exerciser	Tests to verify PCIe subsystem with a custom stimulus generator.

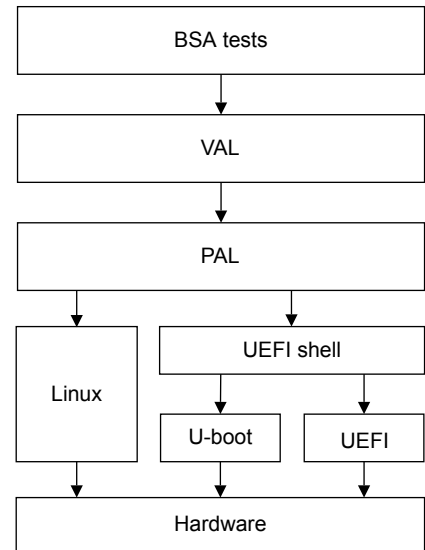
## 1.4 Layered software stack

This section provides information on compliance tests that use the layered software stack.

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

1. Test suite
2. Validation Abstraction Layer (VAL)
3. Platform Abstraction Layer (PAL)

The following figure illustrates different layers in layered software stack.



**Figure 1-1 Layered software stack**

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

**Table 1-3 Compliance test layers**

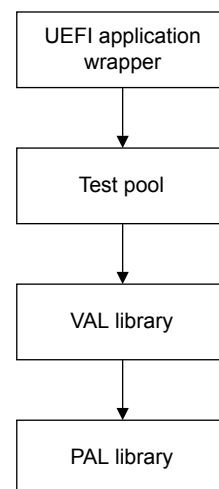
Layer	Description
Test suite	Collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL.
VAL	Provides a uniform view of all the underlying hardware and test infrastructure to the test suite.
PAL	Is a C-based, Arm-defined API 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 intended for the compliance test to reach or use other abstractions in the test platform such as the UEFI infrastructure, and U-boot infrastructure.

This section contains the following subsections:

- [1.4.1 Compliance test software stack with UEFI shell 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 shell application

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

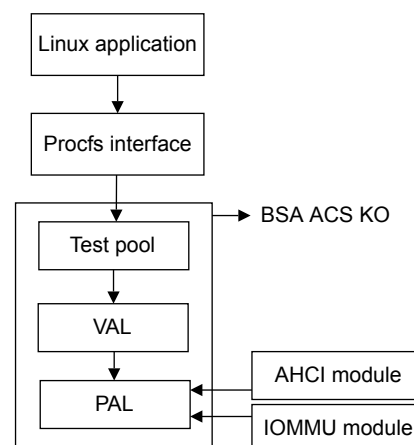


**Figure 1-2 UEFI shell application**

### 1.4.2 Compliance test software stack with Linux application

The stack is spread across user mode and kernel mode space. The Linux command-line application running in the user mode space and the kernel module communicate using a `procfs` interface.

The test pool, VAL, and PAL layers are built as a kernel module. The following figure illustrates the compliance test software stack with Linux application as an example.



**Figure 1-3 Linux application**

The BSA 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 BSA ACS kernel module. A separate patch file is provided to patch the drivers appropriately to export the required information. For more information on Linux application, see the *Arm® BSA Architecture Compliance User Guide*.

### 1.4.3 Coding guidelines

The coding guidelines followed for the implementation of the test suite are described as follows.

1. All the tests call VAL APIs.
2. VAL APIs might call PAL APIs depending on the requested functionality.
3. A test does not directly interface with PAL functions.

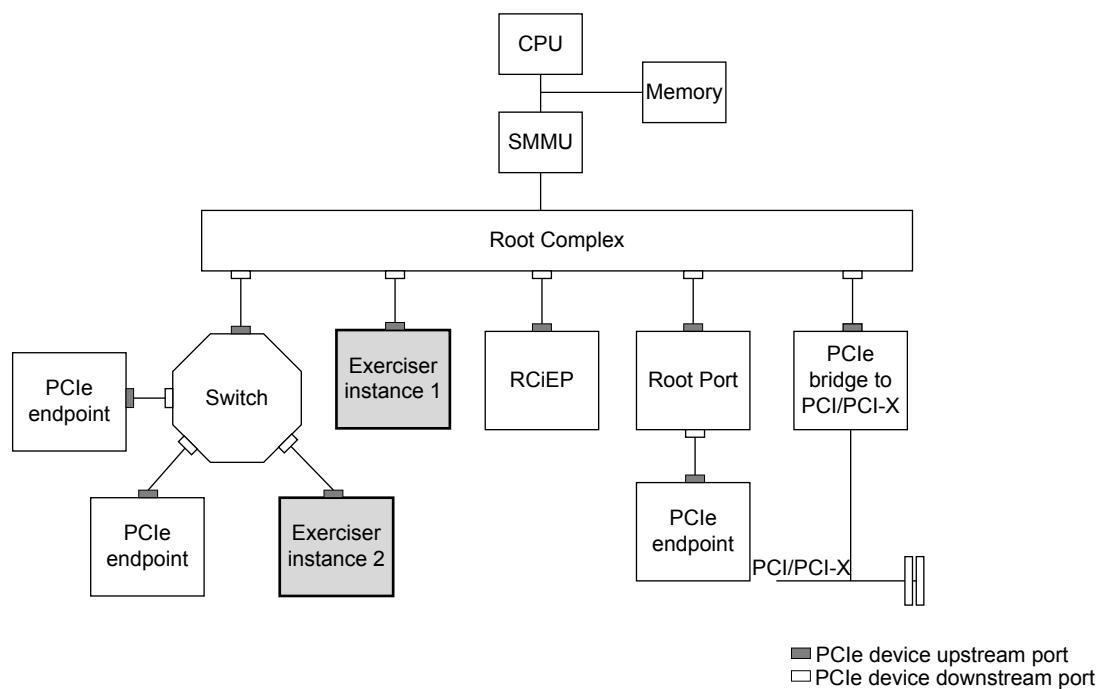
4. The test layer does not need any code modifications when porting from one platform to another.
5. All the platform porting changes are limited to PAL.
6. The VAL might require changes if there are architectural changes impacting multiple platforms.



## 1.5 Exerciser

This section provides information on Exerciser and how it behaves in SoC.

Exerciser is a PCIe endpoint device that can be programmed to generate custom stimuli for verifying the BSA compliance of PCIe IP integration into an Arm System on Chip (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 illustrates the PCIe hierarchy consisting of various endpoints, switches, and bridges.



**Figure 1-4 Exerciser in a SoC**

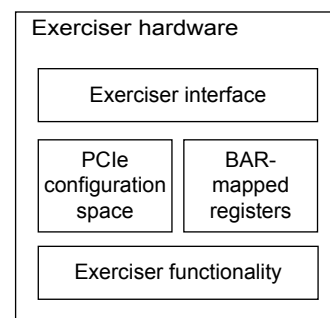
Root Complex integrated EndPoint (RCiEP) and Root Complex Event Collector (RCEC) are endpoints connected directly to Root Complex. PCIe endpoints are connected either to the Root Port or downstream ports. Bridges are used to connect PCI devices into PCIe hierarchy while switches are used to connect multiple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

The figure illustrates 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 DEFINED*. 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.



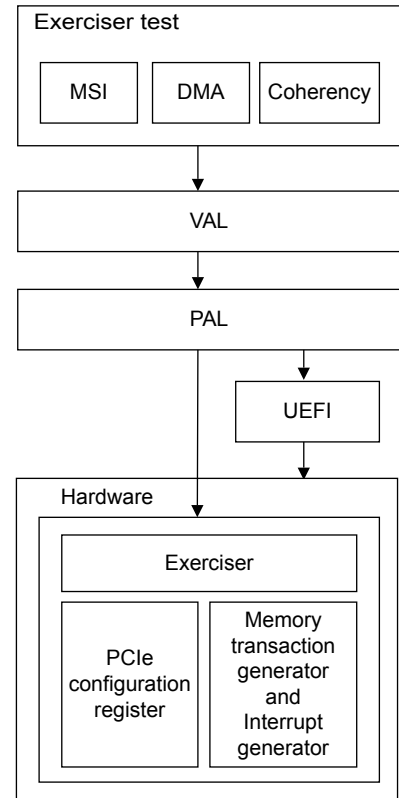
**Figure 1-5 Reference implementation of exerciser hardware**

## 1.6 Compliance test software stack for exerciser with UEFI shell application

This section provides information on exerciser with UEFI shell application.

The exerciser tests validate device interrupts (legacy interrupt and MSI-X interrupt), DMA (address translation and memory access), and coherency behavior. The exerciser PCIe configuration space is accessed using UEFI or MMIO APIs and exerciser functionality like interrupt generation and DMA transactions can be accessed using exerciser APIs.

The following figure illustrates the compliance test software stack for exerciser with UEFI shell application.

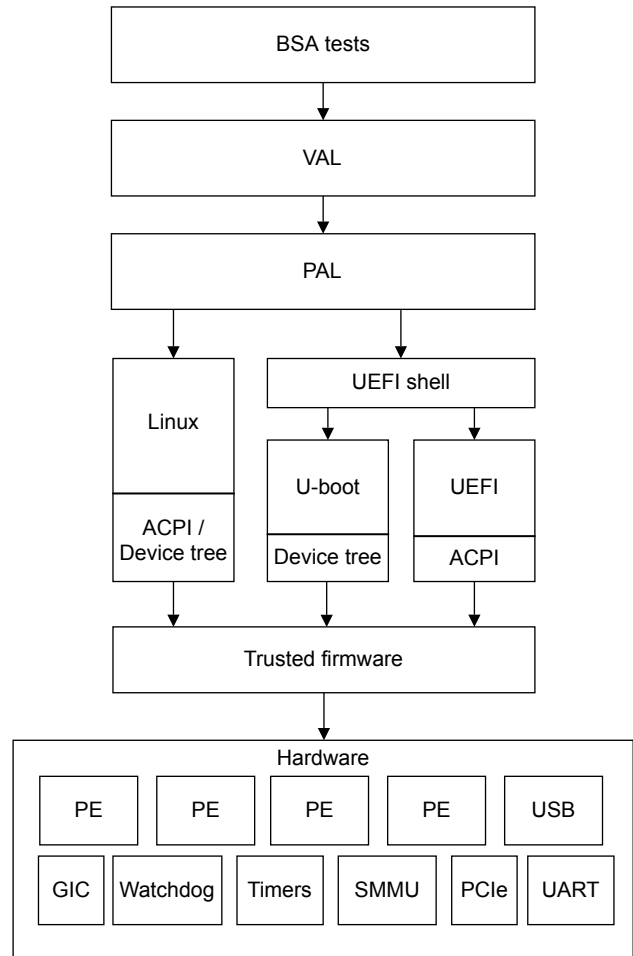


**Figure 1-6 Exerciser with UEFI shell application**

## 1.7 Test platform abstraction

This section provides information on the uses of test platform abstraction.

The following figure illustrates the compliance suite and the uses of the test platform abstraction.



**Figure 1-7 Test platform abstraction**

The following table describes the BSA abstraction terms.

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

Abstraction	Description
UEFI	UEFI Shell application 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.
Hardware	PE and controllers that are specified as part of the BSA specification.

# Chapter 2

## Execution model and flow control

This chapter provides an overview of the execution model and the flow control used for BSA ACS.

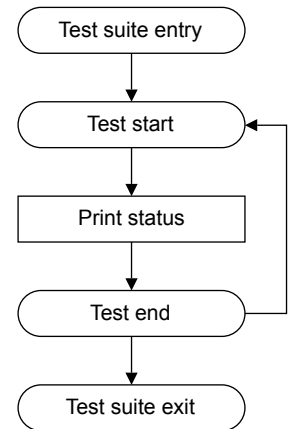
It contains the following sections:

- [2.1 Execution model and flow control on page 2-22.](#)
- [2.2 Test build and execution flow on page 2-23.](#)

## 2.1 Execution model and flow control

This section provides information on the process of the execution model and the flow control.

The following figure illustrates 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

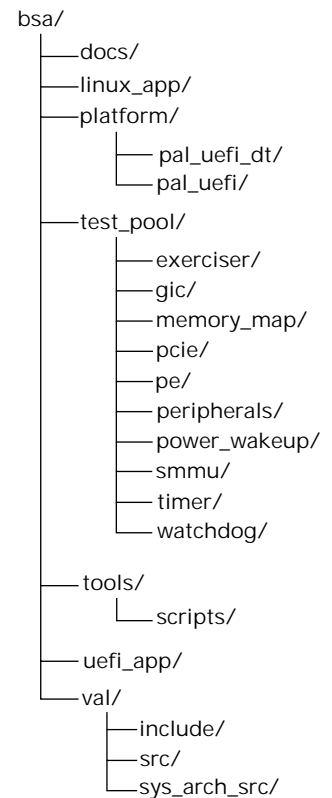
This section describes the source code directory structure and provides references for building the tests.

This section contains the following subsections:

- [2.2.1 Source code directory on page 2-23.](#)
- [2.2.2 Building the tests on page 2-24.](#)

### 2.2.1 Source code directory

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



**Figure 2-2 BSA ACS directory structure**

The following table describes all the directories in the BSA ACS directory structure.

**Table 2-1 BSA ACS directory structure description**

Directory name	Description
pal_uefi	Platform code targeting UEFI implementation.
pal_uefi_dt/	Platform code targeting U-boot / Device tree (DT) implementation.
val	Common code that is used by the tests. Makes calls to PAL as necessary.
uefi_app	UEFI application source to call into the tests entry point.
test_pool	Test case source files for the test suite.
linux_app	Linux command-line executable source code.
docs	Documentation.
scripts	Scripts written for this suite.

## 2.2.2 Building the tests

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

### Prerequisites

- IR and ES platforms run ACS as UEFI shell application. To build BSA ACS as a UEFI Shell application, a UEFI EDK2 source tree is required.
- To build the BSA ACS kernel module, Linux kernel tree version 5.10 or above is required.

For more information on building BSA ACS, see the [README](#).

### Test build for ES

The build steps for the compliance suite to be compiled as a UEFI shell application for ES environment are available in the [README](#). The steps to port the reference implementation and build EL3 firmware are beyond the scope of this document.

### Test build for IR

The build steps for the compliance suite to be compiled as a UEFI shell application for IR environment are available in the [README](#). On U-boot platforms ACS UEFI shell application runs on top of UEFI shell, which in turn runs on top of U-boot as EFI payload. The steps to build UEFI shell, porting 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 BSA ACS kernel module, which is a dependency for the BSA ACS Linux application, are available in the *Arm® BSA Architecture Compliance 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](#) on page 3-26.
- [3.2 API definitions](#) on page 3-27.

## 3.1 Overview of PAL API

This section provides an 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.

The reference PAL implementations are available in the following locations:

- [\*UEFI\*](#)
- [\*Linux\*](#)
- [\*DT\*](#)

## 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 on page 3-27](#).
- [3.2.2 PE APIs on page 3-27](#).
- [3.2.3 GIC APIs on page 3-28](#).
- [3.2.4 Timer APIs on page 3-30](#).
- [3.2.5 Watchdog APIs on page 3-31](#).
- [3.2.6 PCIe APIs on page 3-32](#).
- [3.2.7 IO-Virt APIs on page 3-35](#).
- [3.2.8 SMMU APIs on page 3-37](#).
- [3.2.9 Peripheral APIs on page 3-38](#).
- [3.2.10 DMA APIs on page 3-42](#).
- [3.2.11 Exerciser on page 3-44](#).
- [3.2.12 Miscellaneous APIs on page 3-46](#).
- [3.2.13 Device Tree APIs on page 3-50](#).

### 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

### 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
get_num	uint32_t pal_pe_get_num();	Returns the number of PEs in the system.
create_info_table	void pal_pe_create_info_table(PE_INFO_TABLE *PeTable);	Gathers information about the PEs in the system and fills the <code>info_table</code> with the relevant data.  For related definitions, see <i>Note</i> .
call_smc	void pal_pe_call_smc(ARM_SMC_ARGS *args);	Abstracts the smc instruction. The input arguments to this function are x0 to x7 registers filled in with the appropriate parameters.
execute_payload	void pal_pe_execute_payload(ARM_SMC_ARGS *args);	Abstracts the PE wakeup and execute functionality. Ideally, this function calls the PSCI_ON SMC command.
update_elr	void pal_pe_update_elr(void *context, uint64_t offset);	Updates the ELR to return from exception handler to a required address.
get_esr	uint64_t pal_pe_get_esr(void *context);	Returns the exception syndrome from exception handler.
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_gsicv;    ///< PMU Interrupt ID
    uint32_t gmain_gsicv; ///< GIC Maintenance 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	<code>void pal_gic_create_info_table(GIC_INFO_TABLE *gic_info_table);</code>	Gathers information about the GIC subsystem and fills the <code>gic_info_table</code> with the relevant data.
install_isr	<code>uint32_t pal_gic_install_isr(uint32_t int_id, void (*isr)(void));</code>	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.  <code>int_id</code> : Interrupt ID to install the ISR. <code>isr</code> : Function pointer of the ISR.
end_of_interrupt	<code>uint32_t pal_gic_end_of_interrupt(uint32_t int_id);</code>	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.  <code>int_id</code> : Interrupt id for which interrupt must be disabled.
request_irq	<code>uint32_t pal_gic_request_irq(unsigned int irq_num, unsigned int mapped_irq_num, void *isr);</code>	Registers the interrupt handler for a given IRQ.  <code>irq_num</code> : hardware IRQ number <code>mapped_irq_num</code> : mapped IRQ number. <code>isr</code> : Interrupt Service Routine that returns the status.
free_irq	<code>void pal_gic_free_irq(unsigned int irq_num, unsigned int mapped_irq_num);</code>	Frees the registered interrupt handler for a given IRQ.  <code>irq_num</code> : Hardware IRQ number <code>mapped_irq_num</code> : mapped IRQ number
set_intr_trigger	<code>uint32_t pal_gic_set_intr_trigger (uint32_t int_id, INTR_TRIGGER_INFO_TYPE_e trigger_type);</code>	Sets the trigger type to edge or level.  <code>int_id</code> : interrupt ID which must be enabled and the service routine installed for <code>trigger_type</code> : interrupt trigger type edge or level
request_msi	<code>uint32_t pal_gic_request_msi(uint32_t its_id, uint32_t device_id, uint32_t IntID, uint32_t msi_index, uint32_t *msi_addr, uint32_t *msi_data);</code>	Creates the MSI mappings for an LPI with interrupt ID <code>IntID</code> in ITS tables and assigns the <code>msi_addr</code> and <code>msi_data</code> .  <code>its_id</code> : ITS ID for the current device. <code>Device_id</code> : Device ID of the endpoint <code>IntID</code> : LPI interrupt ID <code>*msi_addr</code> : MSI address <code>*msi_data</code> : MSI data

Table 3-3 GIC APIs and their descriptions (continued)

API name	Function prototype	Description
free_msi	<code>uint32_t pal_gic_free_msi(uint32_t its_id, uint32_t device_id, uint32_t IntID, uint32_t msi_index);</code>	Deletes the MSI mappings for an LPI with interrupt ID IntID from the ITS tables.  <code>its_id</code> : ITS ID for the current device.  <code>device_id</code> : PCIe bus, device, function.  <code>IntID</code> : LPI interrupt ID  <code>msi_index</code> : MSI index in the MSI table
its_configure	<code>uint32_t pal_gic_its_configure();</code>	Configures the ITS, allocates memory for different ITS tables, LPI configuration tables, and enables the ITS.
get_max_lpi_id	<code>pal_gic_get_max_lpi_id();</code>	Returns the maximum LPI ID supported.
bsa_gic_imp	<code>uint32_t pal_bsa_gic_imp(void);</code>	Checks if GIC init and interrupt handlers ACS code is used.

**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_GICC_GICRD,
    ENTRY_TYPE_GICR_GICRD,
    ENTRY_TYPE_GICITS,
    ENTRY_TYPE_GIC_MSI_FRAME
}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;
    uint32_t entry_id; /* This entry_id is used to tell component ID */
    uint64_t length; /* This length is only used in case of Re-Distributor Range Address length */
    uint32_t flags;
    uint32_t spi_count;
    uint32_t spi_base;
}GIC_INFO_ENTRY;
```

**3.2.4 Timer APIs**

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

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 timer_info_table with information about the available local and system timers in the system.  timer_info_table: Address where the timer information must be filled.

**Note**

- This structure holds the timer-related information of the system.

```
typedef struct {
    uint32_t s_el1_timer_flag;
    uint32_t ns_el1_timer_flag;
    uint32_t el2_timer_flag;
    uint32_t el2_virt_timer_flag;
    uint32_t s_el1_timer_gsv;
    uint32_t ns_el1_timer_gsv;
    uint32_t el2_timer_gsv;
    uint32_t virtual_timer_flag;
    uint32_t virtual_timer_gsv;
    uint32_t el2_virt_timer_gsv;
    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 gsv[8];
    uint32_t virt_gsv[8];
    uint32_t flags[8];
}TIMER_INFO_GTBLOCK;
```

**3.2.5 Watchdog APIs**

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

Table 3-5 Watchdog API and its description

API name	Function prototype	Description
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.  wd_table: Address where the watchdog information must be filled.

**Note**

- This data structure holds the watchdog-related information of the system.

```
typedef struct {
    uint64_t wd_ctrl_base;    ///< Watchdog Control Register Frame
```

```
uint64_t wd_refresh_base; ///< Watchdog Refresh Register Frame
uint32_t wd_gsv;          ///< Watchdog Interrupt ID
uint32_t wd_flags;
}WD_INFO_BLOCK;
```

### 3.2.6 PCIe APIs

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 fills the PCIe info_table. Ideally, this function reads the ACPI MCFG table to retrieve the ECAM base address.  <b>PcieTable</b> : Address where the PCIe information must be filled.
io_read_cfg	uint32_t pal_pcie_io_read_cfg(uint32_t bdf, uint32_t offset, uint32_t *data);	Abstracts the configuration space read of a device identified by BDF (Bus, Device, and Function). This is used only in peripheral tests and need not be implemented in Linux. It returns success or failure.  <b>bdf</b> : PCI bus device and function. <b>offset</b> : Register offset within the device PCIe configuration space. <b>data</b> : 32-bit value at offset from ECAM base specified by BDF.
io_write_cfg	void pal_pcie_io_write_cfg(uint32_t Bdf, uint32_t offset, uint32_t data)	Abstracts the configuration space write of a device identified by BDF. Writes 32-bit data to the configuration space of the device at an offset.  <b>bdf</b> : PCI bus device and function. <b>offset</b> : Register offset within the device PCIe configuration space. <b>data</b> : 32-bit value at offset from ECAM base specified by BDF.
get_mcfg_ecam	uint64_t pal_pcie_get_mcfg_ecam();	Returns the PCI ECAM address from the ACPI MCFG table address.



**Table 3-6 PCIe APIs and their descriptions (continued)**

API name	Function prototype	Description
get_msi_vectors	uint32_t pal_get_msi_vectors(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn, PERIPHERAL_VECTOR_LIST **mvector);	Creates a list of MSI(X) vectors for a device. It returns the number of MSI(X) vectors.
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();	Checks P2P support in the PCIe hierarchy.  Returns 1 if P2P feature is not supported and 0 if it is supported.
dev_p2p_support	uint32_t pal_pcie_dev_p2p_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Checks the PCIe device P2P support. seg: PCI segment number bdf: PCI Bus, Device, and Function  Returns 1 if P2P feature is not supported and 0 if P2P feature is supported.
is_cache_present	uint32_t pal_pcie_is_cache_present (uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Checks whether the PCIe device has an <i>Address Translation Cache</i> (ATC). seg: PCI segment number bus: PCI bus address dev: PCI device address fn: PCI function number  Returns 0 if the device does not have ATC, or else 1.

**Table 3-6 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 3-6 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;
```

**3.2.7 IO-Virt APIs**

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.  iovirt: Address where the IOVIRT information must be filled.
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.  rc_block: Root complex IOVIRT block base address.
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.  smmu_block: SMMU IOVIRT block base address.
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.  rc_seg_num: Root complex segment number.

**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;
    uint32_t num_pmcgs;
    IOVIRT_BLOCK blocks[];
}IOVIRT_INFO_TABLE;
```

### 3.2.8 SMMU APIs

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	<code>uint32_t pal_smmu_check_device_iova(void *port, uint64_t dma_addr);</code>	<p>Checks if the input DMA address belongs to the input device. This can be done by tracking 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.</p> <p>0 is returned if address belongs to the device. Nonzero is returned if there are IMPLEMENTATION DEFINED error values.</p> <p><b>port:</b> Device port whose domain IOVA table is checked.</p> <p><b>dma_addr:</b> DMA address which is checked.</p>
device_start_monitor_iova	<code>void pal_smmu_device_start_monitor_iova(void *port);</code>	<p>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 <code>check_device_iova</code> call.</p>
device_stop_monitor_iova	<code>void pal_smmu_device_stop_monitor_iova(void *port);</code>	<p>Stops the recording of the DMA addresses being used by the input port.</p>
max_pasids	<code>uint32_t pal_smmu_max_pasids(uint64_t smmu_base);</code>	<p>Returns the maximum PASID value supported by the SMMU controller. For SMMUv3, this value can be read from the IDR1 register.</p> <p>0 is returned when PASID support is not detected. Nonzero is returned if maximum PASID value supported for the input SMMU.</p>

**Table 3-8 SMMU APIs and their descriptions (continued)**

API name	Function prototype	Description
pa2iova	uint64 pal_smmu_pa2iova(uint64 SmmuBase, uint64 Pa);	<p>Converts physical address to I/O virtual address.</p> <p><b>SmmuBase:</b> physical address of the SMMU for conversion to virtual address.</p> <p><b>Pa:</b> physical address to use in conversion.</p> <p>Returns 0 on success and 1 on failure.</p>
smmu_disable	uint32 pal_smmu_disable(uint64 SmmuBase);	<p>Globally disables the SMMU based on input base address.</p> <p><b>SmmuBase:</b> physical address of the SMMU that must be globally disabled.</p> <p>Returns 0 for success and 1 for failure.</p>
create_info_table	void pal_smmu_create_info_table(SMMU_INFO_TABLE *smmu_info_table);	<p>Abstracts the steps to gather information about SMMUs in the system and fills the <b>info_table</b>.</p>
create_pasid_entry	uint32_t pal_smmu_create_pasid_entry(uint64_t smmu_base, uint32_t pasid);	<p>Prepares the SMMU page tables to support input PASID.</p> <p><b>smmu_base:</b> physical address of the SMMU for which PASID support is needed.</p> <p><b>pasid:</b> Process Address Space Identifier.</p> <p>Returns 0 for success and 1 for failure.</p>

### 3.2.9 Peripheral APIs

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

**Table 3-9 Peripheral APIs and their descriptions**

API name	Function prototype	Description
<code>create_info_table</code>	<code>void pal_peripheral_create_info_table(PERIPHERAL_INFO_TABLE *per_info_table);</code>	Abstracts the steps to gather information on all the peripherals present in the system and fills the information in the <code>per_info_table</code> .
<code>get_legacy_irq_map</code>	<code>uint32_t pal_pcie_get_legacy_irq_map(uint32_t bus, uint32_t dev, uint32_t fn, PERIPHERAL_IRQ_MAP *irq_map);</code>	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 <code>_PRT</code> method of the device ACPI namespace. The following are the return values:  0: success. <code>irq_map</code> successfully retrieved in <code>irq_map</code> buffer.  1: unable to access the PCI bridge device of the input PCI device  2: unable to fetch the ACPI <code>_PRT</code> handle  3: unable to access the ACPI <code>_PRT</code> object  5: legacy interrupt out of range
<code>is_device_behind_smmu</code>	<code>uint32_t pal_pcie_is_device_behind_smmu(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);</code>	Checks if a device with the input BDF is behind an SMMU. One way of checking this in Linux is to check if the <code>iommu_group</code> value of this device is nonzero.  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, 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 3-9 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.
memory_get_unpopulated_addr	uint64_t pal_memory_get_unpopulated_addr(uint64_t *addr, uint32_t instance);	Returns the address of unpopulated memory of the requested instance from <i>Grand Central Dispatch</i> (GCD) memory map.  <b>addr</b> : Address of the unpopulated memory.  <b>instance</b> : Instance of memory.  Returns 0 for success.
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 <b>MEMORY_INFO_TABLE</b> with information about memory in the system. This is achieved by parsing the UEFI memory map.  <b>peripheralInfoTable</b> : Address where the peripheral information must be filled.  Returns none.

---

**Note**

---

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

- This structure holds the peripherals-related information in the system.

```
/**
@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;
```

- This structure holds the memory-related information in the system.

```
typedef struct {
MEM_INFO_TYPE_e type;
uint64_t phy_addr;
uint64_t virt_addr;
uint64_t size;
uint64_t flags; //To Indicate Cacheability etc..
}MEM_INFO_BLOCK;

typedef struct {
uint64_t dram_base;
uint64_t dram_size;
MEM_INFO_BLOCK info[];
}MEMORY_INFO_TABLE;
```

### 3.2.10 DMA APIs

These functions abstract information that is specific to DMA operations 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 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.
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 Requester for the last transaction. It is used by the test to verify if the address used by the DMA Requester was the same as what was allocated by the test.

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

API name	Function prototype	Description
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 must be enhanced.

```
/**
@brief DMA controllers info structure
*/
typedef enum {
    DMA_TYPE_USB = 0x2000,
    DMA_TYPE_SATA,
    DMA_TYPE_OTHER,
}DMA_INFO_TYPE_e;

typedef struct {
    DMA_INFO_TYPE_e type;
    void *target;          ///< The actual info stored in these pointers is implementation
                           specific.
    void *port;
    void *host;            ///< It will be used only by PAL. hence void.
    uint32_t flags;
}DMA_INFO_BLOCK;
```

**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
set_param	uint32_t pal_exerciser_set_param(EXERCISER_PARAM_TYPE type, uint64_t value1, uint64_t value2, uint32_t instance)	Writes the configuration parameters to the PCIe stimulus generation hardware indicated by the instance number. The supported configuration parameters include:  1 – Snoop attributes 2 – Legacy IRQ parameters 3 – MSI(x) attributes 4 – DMA attributes 5 – Peer-to-Peer attributes 6 – PASID attributes 7 – P2P_ATTRIBUTES 8 – PASID_ATTRIBUTES 9 – CFG_TXN_ATTRIBUTES 10 – ATS_RES_ATTRIBUTES 11 – TRANSACTION_TYPE 12 – NUM_TRANSACTIONS  value2 is an optional argument and must be ignored for some configuration parameters.
get_param	uint32_t pal_exerciser_get_param(EXERCISER_PARAM_TYPE type, uint64_t *value1, uint64_t *value2, uint32_t instance)	Returns the requested configuration parameter values through 64-bit input arguments value1 and value2. The function returns a value of 1 to indicate read success and 0 to indicate read failure.
set_state	uint32_t pal_exerciser_set_state(EXERCISER_STATE state, uint64_t *value, uint32_t instance)	Sets the state of the PCIe stimulus generation hardware. The supported states include:  1 – RESET, hardware in reset state. 2 – ON, this state is set after hardware is initialized and is ready to generate stimulus. 3 – OFF, this state is set to indicate that hardware can no longer generate stimulus. 4 – ERROR, this state is set to signal an error with hardware.
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)	<p>Abstracts the steps to implement the requested operation on the PCIe stimulus generation hardware. Following are the supported operations:</p> <ul style="list-style-type: none"> <li>1 – START_DMA,</li> <li>2 – GENERATE_MSI</li> <li>3 – GENERATE_L_INTR</li> <li>4 – MEM_READ</li> <li>5 – MEM_WRITE</li> <li>6 – CLEAR_INTR</li> <li>7 – PASID_TLP_START</li> <li>8 – PASID_TLP_STOP</li> <li>9 – TXN_NO_SNOOP_ENABLE</li> <li>10 – TXN_NO_SNOOP_DISABLE</li> <li>11 – START_TXN_MONITOR</li> <li>12 – STOP_TXN_MONITOR</li> <li>13 – ATS_TXN_REQ</li> </ul>
get_data	uint32_t pal_exerciser_get_data(EXERCISER_DATA_TYPE type, exerciser_data_t *data, uint32_t instance)	<p>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:</p> <ul style="list-style-type: none"> <li>1 – EXERCISER_DATA_CFG_SPACE</li> <li>2 – EXERCISER_DATA_BAR0_SPACE</li> </ul>

### 3.2.12 Miscellaneous APIs

Miscellaneous APIs of print, mem, mmio, and others are described in the following table.

**Table 3-12 Miscellaneous APIs and their descriptions**

API name	Function prototype	Description
print	void pal_print(char *string, uint64_t data);	Sends a formatted string to the output console. <b>string:</b> An ASCII string. <b>data:</b> Data for the formatted output.
print_raw	void pal_print_raw(uint64_t addr, char *string, uint64_t data);	Sends a string to the output console without using the platform print function. This function gets COMM port address and directly writes to the address character by character. <b>addr:</b> Address to be written. <b>string:</b> An ASCII string. <b>data</b> : Data for the formatted output.
strncmp	pal_strncmp uint32_t pal_strncmp (char *FirstString, char *SecondString, uint32_t Length);	Compares two strings. Returns zero if strings are identical, or else a nonzero value. <b>FirstString:</b> The pointer to the first null-terminated ASCII string. <b>SecondString:</b> The pointer to the second null-terminated ASCII string. <b>Length</b> The maximum number of ASCII characters for comparison.
mmio_read	uint32 pal_mmio_read(uint64 addr);	Provides a single point of abstraction to read from all memory-mapped I/O addresses. <b>addr:</b> 64-bit input address <b>return:</b> 32-bit data read from the input address
mmio_read8	pal_mmio_read8(uint64 addr);	Provides a single point of abstraction to read 8-bit data from all memory-mapped I/O addresses. <b>addr:</b> 64-bit input address <b>return:</b> 8-bit data read from the input address
mmio_read16	pal_mmio_read16(uint64 addr);	Provides a single point of abstraction to read 16-bit data from all memory-mapped I/O addresses. <b>addr:</b> 64-bit input address <b>return:</b> 16-bit data read from the input address
mmio_read64	pal_mmio_read64(uint64 addr);	Provides a single point of abstraction to read 64-bit data from all memory-mapped I/O addresses. <b>addr:</b> 64-bit input address <b>return:</b> 64-bit data read from the input address

**Table 3-12 Miscellaneous APIs and their descriptions (continued)**

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



**Table 3-12 Miscellaneous APIs and their descriptions (continued)**

API name	Function prototype	Description
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.  <b>dest_buffer</b> : pointer to the destination buffer of the memory copy  <b>src_buffer</b> : pointer to the source buffer of the memory copy  <b>len</b> : number of bytes to copy from source buffer to destination buffer  Returns the destination buffer.
mem_compare	uint32 pal_mem_compare(void *src, void *dest, uint32 len);	Compares the contents of the source and destination buffers.  <b>src</b> : source buffer to be compared  <b>dest</b> : destination buffer to be compared with  <b>len</b> : length of the comparison to be performed
mem_alloc_cacheable	void pal_mem_alloc_cacheable(uint32_t bdf, uint32_t size, void *pa);	Allocates cacheable memory of the requested size.  <b>bdf</b> : BDF of the requesting PCIe device  <b>size</b> : size of the memory region to be allocated  <b>pa</b> : physical address of the allocated memory
mem_free_cacheable	void pal_mem_free_cacheable(uint32_t bdf, uint32_t size, void *va, void *pa);	Frees the cacheable memory allocated by Linux DMA Framework APIs.  <b>bdf</b> : Bus, Device, and Function of the requesting PCIe device  <b>size</b> : size of memory region to be freed  <b>va</b> : virtual address of the memory to be freed  <b>pa</b> : 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.  <b>va</b> : 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.  <b>MicroSeconds</b> : the minimum number of microseconds to be delayed  Returns the value of the microseconds given as input.

Table 3-12 Miscellaneous APIs and their descriptions (continued)

API name	Function prototype	Description
mem_set	void pal_mem_set (void *buf, uint32 size, uint8 value);	A buffer with a known specified input value. buf: pointer to the buffer to fill size: number of bytes in the buffer to fill value: value to fill the buffer with
page_size	uint32_t pal_mem_page_size();	Returns the memory page size (in bytes) used by the platform.
alloc_pages	void* pal_mem_alloc_pages (uint32 NumPages);	Allocates the requested number of memory pages.
free_pages	void pal_mem_free_pages (void *PageBase, uint32_t NumPages);	Frees pages as requested.
phys_to_virt	void* pal_mem_phys_to_virt (uint64_t Pa);	Returns the VA of the input PA. Pa: Physical Address of the memory to be converted. Returns the VA.

### 3.2.13 Device Tree APIs

These functions abstract information that is specific to Device Tree (DT) operations in the system.

For IR systems, the platform depends on DT. The following APIs are for parsing the DT and extract the necessary information.

Table 3-13 Device Tree APIs and their descriptions

API name	Function prototype	Description
PE	VOID pal_pe_info_table_pmu_gsv_dt (PE_INFO_TABLE *PeTable)	This API fills in the PE_INFO_TABLE with information about PMU in the system. This is achieved by parsing the DT. PeTable: Address where the PMU information must be filled.
PE	VOID pal_pe_create_info_table_dt(PE_INFO_TABLE *PeTable)	This API fills in the PE_INFO_Table with information about the PEs in the system. This is achieved by parsing the DT blob. PeTable: Address where the PE information must be filled.

**Table 3-13 Device Tree APIs and their descriptions (continued)**

API name	Function prototype	Description
PE	VOID pal_pe_info_table_gmaint_gsisv_dt(PE_INFO_TABLE *PeTable)	This API fills in the PE_INFO_TABLE with information about GIC maintenance interrupt in the system. This is achieved by parsing the DT.  <b>PeTable:</b> Address where the information must be filled.
GIC	VOID pal_gic_create_info_table_dt(GIC_INFO_TABLE *GicTable)	This API fills in the GIC_INFO Table with information about the GIC in the system. This is achieved by parsing the DT blob.  <b>PeTable:</b> Address where the GIC information must be filled.
Timer	VOID pal_timer_create_info_table_dt(TIMER_INFO_TABLE *TimerTable)	This API fills in the TIMER_INFO_Table with information about the timer in the system. This is achieved by parsing the DT blob.  <b>TimerTable:</b> Address where the timer information must be filled.
Watchdog	VOID pal_wd_create_info_table_dt(WD_INFO_TABLE *WdTable)	This API fills in the WD_INFO_Table with information about the WDs in the system. This is achieved by parsing the DT blob.  <b>WdTable:</b> Address where the WD information must be filled.
Peripheral	VOID pal_peripheral_usb_create_info_table_dt(PERIPHERAL_INFO_TABLE *peripheralInfoTable)	This API fills in the PERIPHERAL_INFO_TABLE with information about USB in the system. This is achieved by parsing the DT.  <b>peripheralInfoTable:</b> Address where the peripheral information must be filled.

**Table 3-13 Device Tree APIs and their descriptions (continued)**

API name	Function prototype	Description
Peripheral	VOID pal_peripheral_sata_create_info_table_dt(PERIPHERAL_INFO_TABLE *peripheralInfoTable)	This API fills in the PERIPHERAL_INFO_TABLE with information about SATA in the system. This is achieved by parsing the DT.  peripheralInfoTable: Address where the peripheral information must be filled.
Peripheral	VOID pal_peripheral_uart_create_info_table_dt(PERIPHERAL_INFO_TABLE *peripheralInfoTable)	This API fills in the PERIPHERAL_INFO_TABLE with information about UART in the system. This is achieved by parsing the DT.  peripheralInfoTable: Address where the peripheral information must be filled.
IOVIRT	VOID pal_iovirt_create_info_table_dt(IOVIRT_INFO_TABLE *IoVirtTable)	Parses DT_SMMU_Table and populates the local IOVIRT table.  IoVirtTable: Address where the IOVIRT information must be filled
PCIe	VOID pal_pcie_create_info_table_dt(PCIE_INFO_TABLE *PcieTable)	This API fills in the PCIE_INFO_Table with information about the PCIe's in the system. This is achieved by parsing the DT blob.  PcieTable: Address where the PcieTable information must be filled.

# 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-54.](#)

## A.1 Revisions

This section consists of all the technical changes between different versions of this document.

**Table A-1 Issue 0000\_01**

Change	Location
First release	-