Arm® SBSA ACS Bare-metal

Version 3.1

User Guide

arm

102311_0301_01_en

Arm® SBSA ACS Bare-metal

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-01	25 November 2020	Non-Confidential	First release
0100-02	25 March 2021	Non-Confidential	Second release
0100-03	11 May 2021	Non-Confidential	Third release
0301-01	27 September 2021	Non-Confidential	Fourth release

Non-Confidential Proprietary Notice

(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

developer.arm.com

Inclusive language commitment

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

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

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

Contents
Arm® SBSA ACS Bare-metal User Guide

Preface

This preface introduces the *Arm® SBSA ACS Bare-metal User Guide*.

It contains the following:

- *About this book* on page 6.

About this book

This document provides information on the SBSA ACS Bare-metal.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to SBSA

This chapter provides an overview on Arm SBSA ACS, the ACS design, and steps to customize the bare-metal code.

Chapter 2 Execution of SBSA ACS

This chapter provides information on the execution of the SBSA ACS on a full-chip SoC emulation environment.

Chapter 3 Porting requirements

This chapter provides information on different PAL APIs in PE, GIC, timer, IOVIRT, PCIe, SMMU, peripheral, DMA, exerciser, and other miscellaneous APIs.

Chapter 4 SBSA ACS flow

This chapter provides an overview of the SBSA ACS flow diagram and SBSA test example flow.

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.

Additional reading

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

Arm publications

- *Arm SBSA Architecture Compliance Test Scenario* (PJDOC-2042731200-3439)
- *Arm SBSA Architecture Compliance User Guide* (101547)
- *Arm SBSA Architecture Compliance Validation Methodology* (101544)

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 ACS Bare-metal User Guide*.
- The number 102311_0301_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.

Other information

- *Arm® Developer*.
- *Arm® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction to SBSA

This chapter provides an overview on Arm SBSA ACS, the ACS design, and steps to customize the bare-metal code.

It contains the following sections:

1.1 Abbreviations

The following table lists the abbreviations used in this document.

Table 1-1 Abbreviations and expansions

Abbreviation	Expansion
ACS	Architecture Compliance Suite
DMA	Direct Memory Access
ECAM	Enhanced Configuration Access Mechanism
IORT	Input Output Remapping Table
IOVIRT	Input Output Virtualization
GIC	Generic Interrupt Controller
ITS	Interrupt Translation Service
MPIDR	Multiprocessor ID Register
MSI	Message-Signaled Interrupt
PAL	Platform Abstraction Layer
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PMU	Performance Monitoring Unit
RC	Root Complex
RP	Root Port
SBSA	Server Base System Architecture
SoC	System on Chip
SMC	Secure Monitor Call
SMMU	System Memory Management Unit
UART	Universal Asynchronous Receiver and Transmitter
UEFI	Unified Extensible Firmware Interface
VAL	Validation Abstraction Layer

1.2 SBSA ACS

Arm specifies a hardware system architecture which is based on Arm 64-bit architecture that server system software such as operating systems, hypervisors, and firmware can rely on. This ensures standard system architecture to enable a suitably built single OS image to run on all the hardware compliant with this specification.

Arm provides a test suite named Architecture Compliance Suite (ACS) which contains self-checking portable C-based test cases to verify the compliance of hardware platforms to Server Base System Architecture (SBSA).

For more information on Arm SBSA ACS, see the *README*.

1.3 ACS design

The ACS is designed in a layered architecture that consists of the following components:

- Platform Abstraction Layer (PAL) is a C-based, Arm-defined API that you can implement. It abstracts features whose implementation varies from one target system to another. Each test platform requires a PAL implementation of its own. PAL APIs are meant for the compliance test to reach or use other abstractions in the test platform such as the UEFI infrastructure and bare-metal abstraction.
  - For each component, PAL implementation must populate a data structure which involves supplying SoC-specific information such as base addresses, IRQ numbers, capabilities of PE, PCIe, RC, SMMU, DMA, and others.
  - PAL also uses client drivers underneath to retrieve certain device-specific information and to configure the devices.
- Validation Abstraction Layer (VAL) provides an abstraction over PAL and does not change based on the platform. This layer uses PAL layer to achieve a certain functionality. The following eaxmple achieves read memory functionality.

```
val_pcie_read_cfg -> pal_pcie_read_cfg
```

- Test pool is a layer which contains a list of test cases implemented for each component.
- Application is the top-level layer which allocates memory for component-specific tables and executes the test cases for each component.

The ACS test components are classified as follows:

- PE
- GIC
- PCIe
- Exerciser
- I/O virtualization
- SMMU
- Timer
- Watchdog
- Power-wakeup semantics
- Peripherals
- Memory

1.4 Steps to customize bare-metal code

The following are the steps to customize bare-metal code for different platforms.

Note

The `pal_baremetal` reference code is located in *pal_baremetal*.

1. Create a directory under the `pal_baremetal` folder.

```
mkdir <platform_name>
```

2. Copy the reference code from `pal_baremetal/FVP` folder to `<platform_name>`.

```
cp -r FVP/ platform_name/
```

3. Port all the required APIs. For more details on the list of APIs, see the *Porting requirements* on page 3-19.
4. Modify the file `platform_name/include/platform_override_fvp.h` with platform-specific information. For more details on sample implementation, see the *Execution of SBSA ACS* on page 2-13.

This section contains the following subsection:

- *1.4.1 Test components* on page 1-12.

1.4.1 Test components

The following table lists the bare-metal components for each test implementation.

Table 1-2 Bare-metal components

Components	Files
PE	`pal_pe.c`
GIC	`pal_gic.c`
PCIe	`pal_pcie.c, pal_pcie_enumeration.c`
Exerciser	`pal_exerciser.c`
IOVIRT	`pal_iovirt.c`
SMMU	`pal_smmu.c`
Timer and Watchdog	`pal_timer_wd.c`
Peripherals (UART and Memory)	`pal_peripherals.c`
DMA	`pal_dma.c`
Miscellaneous	`pal_misc.c`

Note

PAL implementation requires porting when the underlying platform design changes.

Chapter 2
Execution of SBSA ACS

This chapter provides information on the execution of the SBSA ACS on a full-chip SoC emulation environment.

It contains the following section:

2.1 SoC emulation environment

Executing SBSA ACS on a full-chip emulation environment requires implementation of PAL. This involves providing a collection of SoC-specific information such as capabilities, base addresses, IRQ numbers to the test logic.

In Unified Extensible Firmware Interface (UEFI) base systems, all the static information is present in UEFI tables. The PAL implementation which is based on UEFI, uses the generated header file for populating data structures. For a bare-metal system, this information must be supplied in a tabular format which becomes easy for PAL API implementation.

This section contains the following subsections:

2.1.1 PE

This section provides information on the number of PEs in the system.

PE-specific information

Tests contain comparison of Multiprocessor ID Register (MPIDR) values with actual values read from register. Such interrupts are generated for the Performance Monitoring Unit (PMU) lines and tested.

PLATFORM_OVERRIDE_PEx_MPIDR:
MPIDR register value represents the xth PE hierarchy (cluster, core).

PLATFORM_OVERRIDE_PEx_INDEX:
Represents the xth PE.

PLATFORM_OVERRIDE_PEx_PMU_GSIV:
PMU interrupt number for xth PE.

A platform with eight PEs is populated as follows:

```
#define PLATFORM_OVERRIDE_PE_CNT        0x8

#define PLATFORM_OVERRIDE_PE0_INDEX     0x0
#define PLATFORM_OVERRIDE_PE0_MPIDR     0x0
#define PLATFORM_OVERRIDE_PE0_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE1_INDEX     0x1
#define PLATFORM_OVERRIDE_PE1_MPIDR     0x100
#define PLATFORM_OVERRIDE_PE1_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE2_INDEX     0x2
#define PLATFORM_OVERRIDE_PE2_MPIDR     0x200
#define PLATFORM_OVERRIDE_PE2_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE3_INDEX     0x3
#define PLATFORM_OVERRIDE_PE3_MPIDR     0x300
#define PLATFORM_OVERRIDE_PE3_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE4_INDEX     0x4
#define PLATFORM_OVERRIDE_PE4_MPIDR     0x10000
#define PLATFORM_OVERRIDE_PE4_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE5_INDEX     0x5
#define PLATFORM_OVERRIDE_PE5_MPIDR     0x10100
#define PLATFORM_OVERRIDE_PE5_PMU_GSIV  0x17

#define PLATFORM_OVERRIDE_PE6_INDEX     0x6
#define PLATFORM_OVERRIDE_PE6_MPIDR     0x10200
#define PLATFORM_OVERRIDE_PE6_PMU_GSIV  0x17
```

```
#define PLATFORM_OVERRIDE_PE7_INDEX     0x7
#define PLATFORM_OVERRIDE_PE7_MPIDR     0x10300
#define PLATFORM_OVERRIDE_PE7_PMU_GSIV  0x17
```

2.1.2 PCIe

This section provides information on the number of Peripheral Component Interconnect express (PCIe) root ports.

PLATFORM_OVERRIDE_NUM_ECAM:
Represents the number of Enhanced Configuration Access Mechanism (ECAM) regions in the system.

PLATFORM_OVERRIDE_PCIE_ECAM_BASE_ADDR_x:
ECAM base address: ECAM maps PCIe configuration space to a memory address. The memory address to the current configuration space must be provided here.

PLATFORM_OVERRIDE_PCIE_SEGMENT_GRP_NUM_x:
Segment number of the xth ECAM region.

PLATFORM_OVERRIDE_PCIE_START_BUS_NUM_x:
Starting bus number of the xth ECAM region.

PLATFORM_OVERRIDE_PCIE_END_BUS_NUM_x:
Ending bus number of the xth ECAM region.

A platform with one ECAM region is populated as follows:

```
/* PCIE platform config parameters */
#define PLATFORM_OVERRIDE_NUM_ECAM               1

/* Platform config parameters for ECAM_0 */
#define PLATFORM_OVERRIDE_PCIE_ECAM_BASE_ADDR_0   0x60000000
#define PLATFORM_OVERRIDE_PCIE_SEGMENT_GRP_NUM_0  0x0
#define PLATFORM_OVERRIDE_PCIE_START_BUS_NUM_0    0x0
#define PLATFORM_OVERRIDE_PCIE_END_BUS_NUM_0      0xFF
```

2.1.3 DMA

This section provides the configuration options for Direct Memory Access (DMA) controller-based tests. Additionally, it describes the parameters for the number of DMA bus Requesters, and DMA Requester attributes that can be customized.

Number of DMA controllers

Header file representation:

```
#define PLATFORM_OVERRIDE_DMA_CNT 0
```

PLATFORM_OVERRIDE_DMA_CNT:
Represents the number of DMA controllers in the system.

DMA Requester attributes

Header file representation:

```
typedef struct {
DMA_INFO_TYPE_e type;
void            *target;   ///< The actual info stored in these pointers is implementation-
specific.
void            *port;
void            *host;
uint32_t        flags;
}DMA_INFO_BLOCK;
```

2.1.4 SMMU and device tests

This section provides an overview on SMMU and the device tests. It also provides information on the number of IOVIRT nodes, SMMUs, RC, PMCG, ITS blocks, I/O virtualization node-specific

information, SMMU node-specific information, RC-specific information, and I/O virtual address mapping.

Number of IOVIRT Nodes

Header file representation:

```
#define IORT_NODE_COUNT 0x3
```

IORT_NODE_COUNT:
Represents the total number of Root Complex (RC), SMMU, ITS, PMCG, and other nodes represented in IORT structure.

Number of SMMUs

Header file representation:

```
#define SMMU_COUNT 0x1
```

SMMU_COUNT:
Represents the number of SMMUs in the system.

Number of RC

Header file representation:

```
#define RC_COUNT 0x1
```

RC_COUNT:
Represents the number of RC present in the system.

Number of PMCG

Header file representation:

```
#define PMCG_COUNT 0x1
```

PMCG_COUNT:
Represents the number of Performance Monitor Counter Groups (PMCGs) present in the system.

Number of ITS blocks

Header file representation:

```
#define IOVIRT_ITS_COUNT 0x1
```

IOVIRT_ITS_COUNT:
Represents the number of Interrupt Translation Service (ITS) nodes in the system.

I/O virtualization node-specific information

Header file representation:

```
typedef struct {
uint32_t type;
uint32_t num_data_map;
NODE_DATA data;
uint32_t flags;
NODE_DATA_MAP data_map[];
}IOVIRT_BLOCK;
```

SMMU node-specific information

Header file representation:

```
typedef struct {
uint32_t arch_major_rev;   ///< Version 1 or 2 or 3
```

```
uint64_t base;             ///< SMMU controller base address
}SMMU_INFO_BLOCK;
```

IOVIRT_SMMUV3_BASE_ADDRESS:
Represents the SMMU base address in the system.

RC-specific information

Header file representation:

```
typedef struct {
uint32_t segment;
uint32_t ats_attr;
uint32_t cca;          //Cache Coherency Attribute
uint64_t smmu_base;
}IOVIRT_RC_INFO_BLOCK;
```

I/O virtual address mapping

Header file representation:

```
typedef struct {
uint32_t input_base;
uint32_t id_count;
uint32_t output_base;
uint32_t output_ref;
}ID_MAP;
```

2.1.5 GIC

This section provides the parameters for Generic Interrupt Controller (GIC) specific test.

GIC-specific tests

Header file representation:

```
#define PLATFORM_OVERRIDE_GICD_COUNT        0x1
#define PLATFORM_OVERRIDE_GICRD_COUNT       0x1
#define PLATFORM_OVERRIDE_GICITS_COUNT      0x1
#define PLATFORM_OVERRIDE_GICH_COUNT        0x1
#define PLATFORM_OVERRIDE_GICMSIFRAME_COUNT 0x0
#define PLATFORM_OVERRIDE_GICC_TYPE         0x1000
#define PLATFORM_OVERRIDE_GICD_TYPE         0x1001
#define PLATFORM_OVERRIDE_GICC_GICRD_TYPE   0x1002
#define PLATFORM_OVERRIDE_GICR_GICRD_TYPE   0x1003
#define PLATFORM_OVERRIDE_GICITS_TYPE       0x1004
#define PLATFORM_OVERRIDE_GICMSIFRAME_TYPE  0x1005
#define PLATFORM_OVERRIDE_GICH_TYPE         0x1006
#define PLATFORM_OVERRIDE_GICC_BASE         0x30000000
#define PLATFORM_OVERRIDE_GICD_BASE         0x30000000
#define PLATFORM_OVERRIDE_GICRD_BASE        0x300C0000
#define PLATFORM_OVERRIDE_GICITS_BASE       0x30040000
#define PLATFORM_OVERRIDE_GICH_BASE         0x2C010000
#define PLATFORM_OVERRIDE_GICITS_ID         0
#define PLATFORM_OVERRIDE_GICIRD_LENGTH     (0x20000*8)
```

2.1.6 Timer

This section provides the parameters for timer-specific tests.

Timer information

Header file representation:

```
#define PLATFORM_OVERRIDE_PLATFORM_TIMER_COUNT  0x2
#define PLATFORM_OVERRIDE_S_EL1_TIMER_GSIV      0x1D
#define PLATFORM_OVERRIDE_NS_EL1_TIMER_GSIV     0x1E
#define PLATFORM_OVERRIDE_NS_EL2_TIMER_GSIV     0x1A
#define PLATFORM_OVERRIDE_VIRTUAL_TIMER_GSIV    0x1B
#define PLATFORM_OVERRIDE_EL2_VIR_TIMER_GSIV    28
```

2.1.7 Watchdog timer

This section provides the parameters for the number of watchdog timer tests and watchdog information.

Number of watchdog timers

Header file representation:

```
#define PLATFORM_OVERRIDE_WD_TIMER_COUNT 2
```

Watchdog information

The following is the list of watchdog timers is present in the system:

- Watchdog timer number
- Control base
- Refresh base
- Interrupt number
- Flags

Header file representation:

```
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;
```

2.1.8 Memory

This section provides information on the memory map in the system.

PLATFORM_OVERRIDE_MEMORY_ENTRY_COUNT:
Represents the number of memory range entries.

PLATFORM_OVERRIDE_MEMORY_ENTRYx_PHY_ADDR:
Represents the physical address of the xth memory entry.

PLATFORM_OVERRIDE_MEMORY_ENTRYx_VIRT_ADDR:
Represents the virtual address of the xth memory entry.

PLATFORM_OVERRIDE_MEMORY_ENTRYx_SIZE:
Represents the size of the xth memory entry.

PLATFORM_OVERRIDE_MEMORY_ENTRYx_TYPE:
Represents the type of the xth memory entry.

The following is an example for memory map.

```
#define PLATFORM_OVERRIDE_MEMORY_ENTRY_COUNT        0x4
#define PLATFORM_OVERRIDE_MEMORY_ENTRY0_PHY_ADDR    0xC000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY0_VIRT_ADDR   0xC000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY0_SIZE        0x4000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY0_TYPE        MEMORY_TYPE_DEVICE
#define PLATFORM_OVERRIDE_MEMORY_ENTRY1_PHY_ADDR    0x10000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY1_VIRT_ADDR   0x10000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY1_SIZE        0xC170000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY1_TYPE        MEMORY_TYPE_NOT_POPULATED
#define PLATFORM_OVERRIDE_MEMORY_ENTRY2_PHY_ADDR    0xFF600000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY2_VIRT_ADDR   0xFF600000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY2_SIZE        0x10000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY2_TYPE        MEMORY_TYPE_RESERVED
#define PLATFORM_OVERRIDE_MEMORY_ENTRY3_PHY_ADDR    0x80000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY3_VIRT_ADDR   0x80000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY3_SIZE        0x7F000000
#define PLATFORM_OVERRIDE_MEMORY_ENTRY3_TYPE        MEMORY_TYPE_NORMAL
```

Chapter 3
Porting requirements

This chapter provides information on different PAL APIs in PE, GIC, timer, IOVIRT, PCIe, SMMU, peripheral, DMA, exerciser, and other miscellaneous APIs.

It contains the following section:

3.1 PAL implementation

PAL is a C-based, Arm-defined API that you can implement. Each test platform requires a PAL implementation of its own.

The bare-metal reference code provides a reference implementation for a subset of APIs. Additional code must be implemented to match the target SoC implementation under the tests.

Note

There are two implementation types for the PAL APIs and are classified in the following tables:

- Yes: indicates that the implementation of this API is already present. Since the values are platform-specific, it must be taken from the platform configuration file.
- Platform-specific: you must implement all the APIs that are marked as platform-specific.

This section contains the following subsections:

3.1.1 PE

The following table lists the different types of APIs in PE.

Table 3-1 PE APIs and their details

API name	Function prototype	Implementation
`create_info_table`	`void pal_pe_create_info_table(PE_INFO_TABLE *PeTable);`	Yes
`call_smc`	`void pal_pe_call_smc(ARM_SMC_ARGS *args);`	Yes
`execute_payload`	`void pal_pe_execute_payload(ARM_SMC_ARGS *args);`	Yes
`update_elr`	`void pal_pe_update_elr(void *context,uint64_toffset);`	Platform-specific
`get_esr`	`uint64_t pal_pe_get_esr(void *context);`	Platform-specific
`data_cache_ops_by_va`	`void pal_pe_data_cache_ops_by_va(uint64_t addr, uint32_t type);`	Yes
`get_far`	`uint64_t pal_pe_get_far(void *context);`	Platform-specific
`install_esr`	`uint32_t pal_pe_install_esr(uint32_t exception_type, void(*esr)(uint64_t, void *));`	Platform-specific

3.1.2 GIC

The following table lists the different types of APIs in GIC.

Table 3-2 GIC APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_gic_create_info_table(GIC_INFO_TABLE* gic_info_table);	Yes
install_isr	uint32_t pal_gic_install_isr(uint32_t int_id, void(*isr)(void));	Platform-specific
end_of_interrupt	uint32_t pal_gic_end_of_interrupt(uint32_t int_id);	Platform-specific
request_irq	uint32_t pal_gic_request_irq(unsigned intirq_num, unsigned int mapped_irq_num,void *isr);	Platform-specific
free_irq	void pal_gic_free_irq(unsigned int irq_num,unsigned int mapped_irq_num);	Platform-specific
set_intr_trigger	uint32_t pal_gic_set_intr_trigger(uint32_t int_idINTR_TRIGGER_INFO_TYPE_etrigger_type);	Platform-specific

3.1.3 Timer

The following table lists the different types of APIs in timer.

Table 3-3 Timer APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_timer_create_info_table(TIMER_INFO_TABLE *timer_info_table);	Yes
wd_create_info_table	void pal_wd_create_info_table(WD_INFO_TABLE *wd_table);	Yes

3.1.4 IOVIRT

The following table lists the different types of APIs in IOVIRT.

Table 3-4 IOVIRT APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_iovirt_create_info_table(IOVIRT_INFO_TABLE *iovirt);	Yes
unique_rid_strid_map	uint32_t pal_iovirt_unique_rid_strid_map(uint64_t rc_block);	Yes
check_unique_ctx_initd	uint32_t pal_iovirt_check_unique_ctx_intid(uint64_t smmu_block);	Yes
get_rc_smmu_base	uint64_t pal_iovirt_get_rc_smmu_base(IOVIRT_INFO_TABLE *iovirt, uint32_t rc_seg_num, uint32_t rid);	Yes

3.1.5 PCIe

The following table lists the different types APIs in PCIe.

Table 3-5 PCIe APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_pcie_create_info_table (PCIE_INFO_TABLE *PcieTable);	Yes
read_cfg	uint32_t pal_pcie_read_cfg(uint32_t bdf, uint32_t offset, uint32_t *data);	Yes

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

API name	Function prototype	Implementation
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);	Platform-specific
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_tfn);	Yes
get_pcie_type	uint32_t pal_pcie_get_pcie_type(uint32_t seg,uint32_t bus, uint32_t dev, uint32_t fn);	Yes
p2p_support	uint32_t pal_pcie_p2p_support(void);	Yes
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);	Yes
get_bdf_wrapper	uint32_t pal_pcie_get_bdf_wrapper (uint32ClassCode, uint32 StartBdf);	Yes
bdf_to_dev	void *pal_pci_bdf_to_dev(uint32_t bdf);	Yes
pal_pcie_ecam_base	uint64_t pal_pcie_ecam_base(uint32_t seg,uint32_t bus, uint32_t dev, uint32_tfunc)	Yes
pci_cfg_read	uint32_t pal_pci_cfg_read(uint32_t bus, uint32_t dev, uint32_t func, uint32_t offset, uint32_t *value)	Yes
pci_cfg_write	void pal_pci_cfg_write(uint32_t bus, uint32_t dev, uint32_t func, uint32_t offset, uint32_t data)	Yes
program_bar_reg	void pal_pcie_program_bar_reg(uint32_tbus, uint32_t dev, uint32_t func)	Yes
enumerate_device	uint32_t pal_pcie_enumerate_device(uint32_t bus, uint32_t sec_bus)	Yes
get_bdf	uint32_t pal_pcie_get_bdf(uint32_t ClassCode, uint32_t StartBdf)	Yes
increment_bus_dev	uint32_t pal_increment_bus_dev(uint32_t StartBdf)	Yes
get_base	uint64_t pal_pcie_get_base(uint32_t bdf, uint32_t bar_index)	Yes
io_read_cfg	uint32_t pal_pcie_io_read_cfg(uint32_t Bdf, uint32_t offset, uint32_t *data) ;	Yes
io_write_cfg	void pal_pcie_io_write_cfg(uint32_t bdf, uint32_t offset, uint32_t data);	Yes
get_device_type	uint32_t pal_pcie_get_device_type(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
get_snoop_bit	uint32_t pal_pcie_get_snoop_bit(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes

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

API name	Function prototype	Implementation
is_device_behind_smmu	uint32_t pal_pcie_is_device_behind_smmu(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
get_dma_support	uint32_t pal_pcie_get_dma_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
get_dma_coherent	uint32_t pal_pcie_get_dma_coherent(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
Is_devicedma_64bit	uint32_t pal_pcie_is_devicedma_64bit(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
get_legacy_irq_map	uint32_t pal_pcie_get_legacy_irq_map(uint32_t Seg, uint32_t Bus, uint32_t Dev, uint32_t Fn, PERIPHERAL_IRQ_MAP *IrqMap);	Platform-specific
get_root_port_bdf	uint32_t pal_pcie_get_root_port_bdf(uint32_t *Seg, uint32_t *Bus, uint32_t *Dev, uint32_t *Func);	Yes
dev_p2p_support	uint32_t pal_pcie_dev_p2p_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
is_cache_present	uint32_t pal_pcie_is_cache_present(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
is_onchip_peripheral	uint32_t pal_pcie_is_onchip_peripheral(uint32_t bdf);	Platform-specific
check_device_list	uint32_t pal_pcie_check_device_list(void);	Yes
get_rp_transaction_frwd_support	uint32_t pal_pcie_get_rp_transaction_frwd_support(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn)	Platform-specific

3.1.6 SMMU

The following table lists the different types of APIs in SMMU.

Table 3-6 SMMU APIs and their details

API name	Function prototype	Implementation
check_device_iova	uint32_t pal_smmu_check_device_iova(void *port, uint64_t dma_addr);	Platform-specific
device_start_monitor_iova	void pal_smmu_device_start_monitor_iova(void *port);	Platform-specific
device_stop_monitor_iova	void pal_smmu_device_stop_monitor_iova(void *port);	Platform-specific
max_pasids	uint32_t pal_smmu_max_pasids(uint64_t smmu_base);	Yes
pa2iova	uint64_t pal_smmu_pa2iova(uint64 SmmuBase, unit64 Pa);	Platform-specific
smmu_disable	uint32_t pal_smmu_disable(uint64 SmmuBase);	Platform-specific
create_pasid_entry	uint32_t pal_smmu_create_pasid_entry(uint64_t smmu_base, uint32_t pasid);	Platform-specific

3.1.7 Peripheral

The following table lists the different types of APIs in peripheral.

Table 3-7 Peripheral APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_peripheral_create_info_table(PERIPHERAL_INFO_TABLE *per_info_table);	Yes
is_pcie	uint32_t pal_peripheral_is_pcie(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
memory_create_info_table	void pal_memory_create_info_table(MEMORY_INFO_TABLE *memoryInfoTable);	Platform-specific
memory_ioremap	uint64_t pal_memory_ioremap(void *addr, uint32_t size, uint32_t attr);	Platform-specific
memory_unmap	void pal_memory_unmap(void *addr);	Platform-specific
memory_get_unpopulated_addr	uint64_t pal_memory_get_unpopulated_addr(uint64_t *addr, uint32_t instance)	Platform-specific

3.1.8 DMA

The following table lists the different types of APIs in DMA.

Table 3-8 DMA APIs and their details

API name	Function prototype	Implementation
create_info_table	void pal_dma_create_info_table(DMA_INFO_TABLE *dma_info_table);	Yes
start_from_device	uint32_t pal_dma_start_from_device(void *dma_target_buf, uint32_t length,void *host,void *dev);	Platform-specific
start_to_device	uint32_t pal_dma_start_to_device(void *dma_source_buf, uint32_t length, void *host, void *target, uint32_t timeout);	Platform-specific
mem_alloc	uint64_t pal_dma_mem_alloc(void *buffer, uint32_t length, void *dev, uint32_t flags);	Platform-specific
scsi_get_dma_addr	void pal_dma_scsi_get_dma_addr(void *port, void *dma_addr, uint32_t *dma_len);	Platform-specific
mem_get_attrs	int pal_dma_mem_get_attrs(void *buf, uint32_t *attr, uint32_t *sh)	Platform-specific
dma_mem_free	void pal_dma_mem_free(void *buffer, addr_tmem_dma, unsigned int length, void *port,unsigned int flags);	Platform-specific

3.1.9 Exerciser

The following table lists the different types of APIs in exerciser.

Table 3-9 Exerciser APIs and their details

API name	Function prototype	Implementation
get_ecsr_base	uint64_t pal_exerciser_get_ecsr_base(uint32_t Bdf,uint32_t BarIndex)	Platform-specific
get_pcie_config_offset	uint64_t pal_exerciser_get_pcie_config_offset(uint32_t Bdf)	Platform-specific
start_dma_direction	uint32_t pal_exerciser_start_dma_direction(uint64_t Base, EXERCISER_DMA_ATTRDirection)	Platform-specific

Table 3-9 Exerciser APIs and their details (continued)

API name	Function prototype	Implementation
find_pcie_capability	uint32_t pal_exerciser_find_pcie_capability(uint32_t ID, uint32_t Bdf, uint32_t Value, uint32_t *Offset)	Platform-specific
set_param	uint32_t pal_exerciser_set_param(EXERCISER_PARAM_TYPE type, uint64_t value1, uint64_t value2, uint32_t bdf);	Platform-specific
get_param	uint32_t pal_exerciser_get_param(EXERCISER_PARAM_TYPE type, uint64_t *value1, uint64_t *value2, uint32_t bdf);	Platform-specific
set_state	uint32_t pal_exerciser_set_state(EXERCISER_STATE state, uint64_t *value, uint32_t bdf);	Platform-specific
get_state	uint32_t pal_exerciser_get_state(EXERCISER_STATE *state, uint32_t bdf);	Platform-specific
ops	uint32_t pal_exerciser_ops(EXERCISER_OPS ops,uint64_t param, uint32_t instance);	Platform-specific
get_data	uint32_t pal_exerciser_get_data(EXERCISER_DATA_TYPE type, exerciser_data_t *data, uint32_tbdf, uint64_t ecam);	Platform-specific
is_bdf_exerciser	uint32_t pal_is_bdf_exerciser(uint32_t bdf)	Platform-specific

3.1.10 Miscellaneous

The following table lists the different types of miscellaneous PAL APIs.

Table 3-10 Miscellaneous APIs and their details

API name	Function prototype	Implementation
mmio_read8	uint8_t pal_mmio_read8(uint64_t addr);	Yes
mmio_read16	uint16_t pal_mmio_read16(uint64_t addr);	Yes
mmio_read	uint32_t pal_mmio_read(uint64_t addr);	Yes
mmio_read64	uint64_t pal_mmio_read64(uint64_t addr);	Yes
mmio_write8	void pal_mmio_write8(uint64_t addr, uint8_t data);	Yes
mmio_write16	void pal_mmio_write16(uint64_t addr, uint16_t data);	Yes
mmio_write	void pal_mmio_write(uint64_t addr, uint32_t data);	Yes
mmio_write64	void pal_mmio_write64(uint64_t addr, uint64_t data);	Yes
print	void pal_print(char8_t *string, uint64_tdata);	Platform-specific
print_raw	void pal_print_raw(uint64_t addr, char *string, uint64_t data)	Yes
mem_free	void pal_mem_free(void *buffer);	Platform-specific
mem_compare	int pal_mem_compare(void *src,void *dest, uint32_t len);	Yes
mem_set	void pal_mem_set(void *buf, uint32_t size,uint8_t value);	Yes
mem_allocate_shared	void pal_mem_allocate_shared(uint32_t num_pe, uint32_t sizeofentry);	Yes
mem_get_shared_addr	uint64_t pal_mem_get_shared_addr(void);	Yes
mem_free_shared	void pal_mem_free_shared(void);	Yes

Table 3-10 Miscellaneous APIs and their details (continued)

API name	Function prototype	Implementation
mem_alloc	void *pal_mem_alloc(uint32_t size);	Platform-specific
mem_virt_to_phys	void *pal_mem_virt_to_phys(void *va);	Platform-specific
mem_alloc_cacheable	void *pal_mem_alloc_cacheable(uint32_t Bdf, uint32_t Size, void **Pa);	Platform-specific
mem_free_cacheable	void pal_mem_free_cacheable(uint32_t Bdf, uint32_t Size, void *Va, void *Pa);	Platform-specific
mem_phys_to_virt	void *pal_mem_phys_to_virt (uint64_t Pa);	Platform-specific
strncmp	uint32_t pal_strncmp(char8_t *str1, char8_t *str2, uint32_t len);	Yes
memcpy	void *pal_memcpy(void *dest_buffer, void *src_buffer, uint32_t len);	Yes
time_delay_ms	uint64_t pal_time_delay_ms(uint64_t time_ms);	Platform-specific
page_size	uint32_t pal_mem_page_size();	Platform-specific
alloc_pages	void *pal_mem_alloc_pages (uint32 NumPages);	Platform-specific
free_pages	void pal_mem_free_pages (void *PageBase, uint32_t NumPages);	Platform-specific

Chapter 4
SBSA ACS flow

This chapter provides an overview of the SBSA ACS flow diagram and SBSA test example flow.

It contains the following sections:

4.1 SBSA ACS flow diagram

The following flow diagram shows the sequence of events from initialization of devices, initialization of SBSA test data structures, and test case execution.

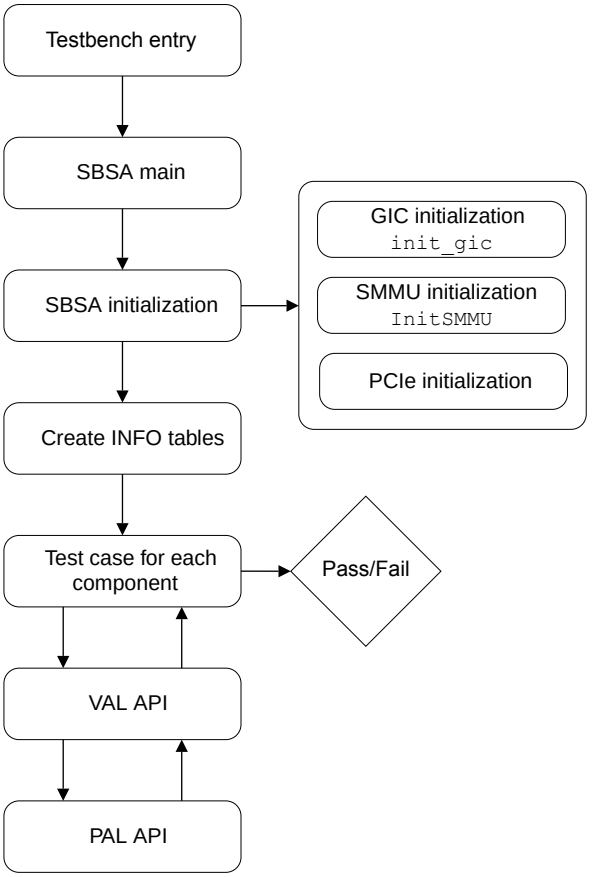

Figure 4-1 SBSA flow diagram

4.2 SBSA test example flow

If the device is Message-Signaled Interrupt (MSI) enabled, then the flag is set to `MSI_ENABLED` by the PAL layer. The test checks whether the device is of type endpoint and then checks if the flags are set to `MSI_ENABLED`.

The following flowchart shows the test that checks MSI support in a PCIe device.

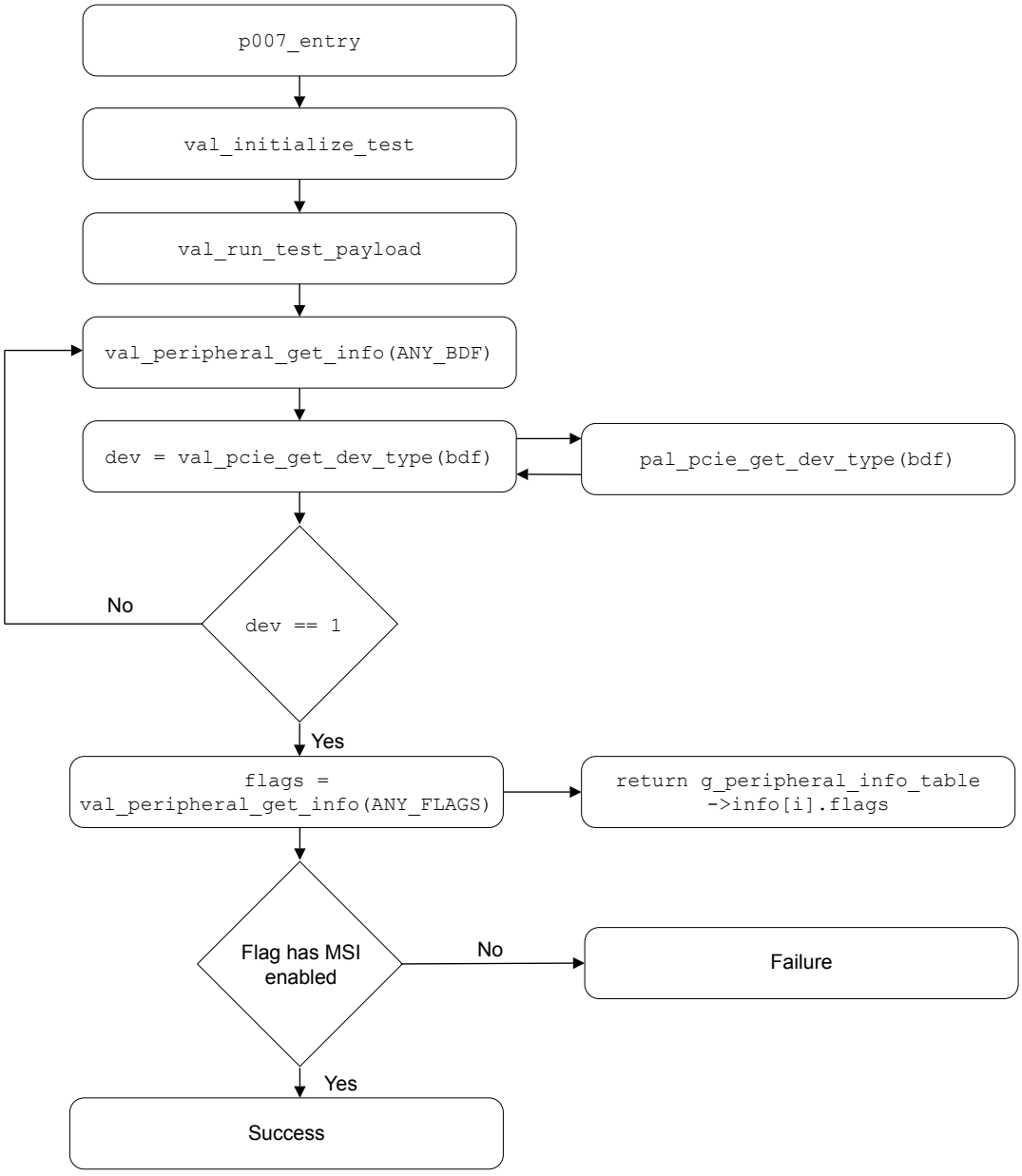

Figure 4-2 SBSA example flow diagram

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-31.

A.1 Revisions

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

Table A-1 Issue A

Change	Location
First release.	-

Table A-2 Difference between Issue A and Issue 0100-02

Change	Location
1. Changed the file name of the component Timer and Watchdog. 2. Added two more components - DMA and Miscellaneous.	See *1.4.1 Test components* on page 1-12.
Changed the node count in IOVIRT nodes.	See *Number of IOVIRT Nodes* on page 2-16.
Added `PLATFORM_OVERRIDE_GICITS_ID` and `PLATFORM_OVERRIDE_GICIRD_LENGTH` in the GIC-specific tests section.	See *2.1.5 GIC* on page 2-17.
Removed `request_msi`, `free_msi`, `its_configure`, and `get_max_lpi_id` APIs in the GIC section.	See *3.1.2 GIC* on page 3-20.
Removed `pal_pci_read_config_byte` and `pci_write_config_byte` and added 10 new APIs in the PCIe section.	See *3.1.5 PCIe* on page 3-21.
Removed the `create_info_table` API in the SMMU section.	See *3.1.6 SMMU* on page 3-23.
Removed `pal_pcie.c` and `pal_pcie_enumeration.c` APIs and added 8 new APIs in the Peripheral section.	See *3.1.7 Peripheral* on page 3-23.
Renamed `pal_mem_alloc_coherent` API to `pal_mem_alloc_cacheable` API and `pal_mem_free_coherent` API to `pal_mem_free_cacheable` API. Added `pal_mem_phys_to_virt` API in the Miscellaneous section.	See *3.1.10 Miscellaneous* on page 3-25.

Table A-3 Difference between issue 0100-02 and issue 0100-03

Change	Location
Added `get_rp_transaction_frwd_support` API in PCIe.	See *3.1.5 PCIe* on page 3-21.
Added `pal_is_bdf_exerciser` API in Exerciser.	See *3.1.9 Exerciser* on page 3-24.

Table A-4 Difference between issue 0100-03 and issue 0301-01

Change	Location
Updated the Execution of SBSA ACS	See *Chapter 2 Execution of SBSA ACS* on page 2-13.
Updated the Porting requirements	See *Chapter 3 Porting requirements* on page 3-19.
Added memory topic.	See *2.1.8 Memory* on page 2-18.
Updated the SBSA example flow diagram.	See *4.2 SBSA test example flow* on page 4-29.