Arm® SBSA ACS Bare-metal

Version 1.0

User Guide

arm

102311_0100_02_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

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

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*.

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

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 section provides an introduction to Arm SBSA describing the ACS architecture and the directory structure.

Chapter 2 SoC emulation environment

This chapter provides details of the parameters that can be customized in `platform_name/include/platform_override_fvp.h` as per the actuals with respect to PE and PCIe on an SoC emulation environment.

Chapter 3 DMA tests

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

Chapter 4 SMMU and device tests

This chapter provides an overview on SMMU and device tests. Additionally, it describes the parameters for 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 that can be customized.

Chapter 5 GIC tests

This chapter describes the parameters for Generic Interrupt Controller (GIC) specific test that can be customized.

Chapter 6 Timer tests

This chapter describes the parameters for timer tests, and timer information that can be customized.

Chapter 7 Watchdog timer tests

This chapter describes the parameters for the number of watchdog timer tests, and watchdog information that can be customized.

Chapter 8 Porting requirements

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

Chapter 9 SBSA ACS flow

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

Appendix A Revisions

This section describes the technical changes made in 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-enterprise-acs@arm.com*. Give:

- The title *Arm SBSA ACS Bare-metal User Guide*.
- The number 102311_0100_02_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Chapter 1
Introduction to SBSA

This section provides an introduction to Arm SBSA describing the ACS architecture and the directory structure.

It contains the following sections:

1.1 SBSA ACS

This chapter describes about the Arm SBSA ACS.

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

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

Arm SBSA ACS can be downloaded from *https://github.com/ARM-software/sbsa-acs*.

1.2 ACS design

This section describes the layered architecture in the ACS design.

ACS is designed in a layered architecture that consists as follows:

1. Platform Adaptation 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 in 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.
2. Validation Adaptation Layer (VAL) is a layer which provides an abstraction over PAL and does not change based on platform. This layer calls the PAL layer to achieve a certain functionality, for example:

   ```
   val_pcie_read_cfg -> pal_pcie_read_cfg
   ```
3. Test pool is a layer which contains a list of test cases implemented for each component.
4. Application is the top-level application which allocates memory for component-specific tables and executes the test case for each component.

The ACS test components are classified as follows:

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

1.3 Steps to customize bare-metal code

This section describes the steps to customize bare-metal code.

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

1. Create a directory under the `<local_path>/sbsa-acs/platform/pal_baremetal/` folder.

   ```
   mkdir platform_name
   ```

2. Copy the reference code from `<local_path>/sbsa-acs/platform/pal_baremetal/FVP/` folder to the folder created, platform_name.

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

3. Port all the required APIs mentioned in *Chapter 8 Porting requirements* on page 8-31.
4. Modify the file `platform_name/include/platform_override_fvp.h` with platform-specific information. For sample implementation, refer Chapter 2 to Chapter 7.

This section contains the following subsection:

- *1.3.1 Directory structure* on page 1-13.

1.3.1 Directory structure

This section describes about the bare-metal layer.

A brief description about the bare-metal layer is as follows:

`pal_baremetal` contains the bare-metal implementation for each test component specified as follows:

Table 1-1 Bare-metal component

Component	File
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
SoC emulation environment

This chapter provides details of the parameters that can be customized in `platform_name/include/platform_override_fvp.h` as per the actuals with respect to PE and PCIe on an SoC emulation environment.

It contains the following section:

2.1 SoC emulation environment

This section describes the execution of the SBSA ACS on a full chip 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 and many more to the test logic. In UEFI bases systems, all the static information is present in UEFI tables. PAL implementation that is based on UEFI, reads this information from the tables and populates the PAL data structures. For a bare-metal system, this information must be supplied in table format which can be read by the PAL API implementation. The PAL implementation uses generated header file for populating data structures.

This section contains the following subsections:

2.1.1 Number of PEs

This section describes the number of Processing Elements (PEs) in the design.

The following is a header file representation of the PEs in the design.

```
#define PLATFORM_OVERRIDE_PE_CNT        0x8
```

For example: If the PE count is equal to 1, then use the following code:

```
#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
```

2.1.2 PE-specific information

This section describes the number of PE-specific information.

Tests contain comparison of MPIDR values with actual values read from register. Such interrupts are generated for the performance monitoring interrupt lines and tested.

#pe_mpidr
: MPIDR register value represents the PE hierarchy (cluster, core).

pe_index
: the PE number.

#pe_performanceinterruptgsiv
: Performance monitoring interrupt number for each core.

Header file representation:

```
#define PLATFORM_OVERRIDE_PE_CNT        0x8
```

For example: If the PE count is equal to 1, then use the following code:

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

2.1.3 Number of PCIe root ports

This section describes the number of Peripheral Component Interconnect express (PCIe) root ports.

Header file representation:

```
#define PLATFORM_OVERRIDE_NUM_ECAM 1
```

For example:

```
#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
```

2.1.4 PCIe root port information

This section describes the PCIe root port information with an example.

Following information corresponding to the root port is described as follows:

- Enhanced Configuration Access Mechanism (ECAM) base address: ECAM maps PCIe configuration space to memory address. The memory address to the current configuration space must be provided here.
- Start bus number: starting bus number on this segment.
- End bus number: ending bus number on this segment
- Segment number

For example:

```
#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
```

2.1.5 PCIe peripherals

This section provides information on the number of PCIe peripheral counts.

Header file representation:

```
#define PLATFORM_PCIE_P2P_NOT_SUPPORTED  1

#define PLATFORM_PERIPHERAL_COUNT        2
#define PERIPHERAL0_DMA_SUPPORT          1
#define PERIPHERAL0_DMA_COHERENT         1
#define PERIPHERAL0_P2P_SUPPORT          1
#define PERIPHERAL0_DMA_64BIT            0
#define PERIPHERAL0_BEHIND_SMMU          1
#define PERIPHERAL0_ATC_SUPPORT          0
```

Chapter 3
DMA tests

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

It contains the following sections:

3.1 Number of DMA controllers

This section provides data for the number of DMA controllers.

Header file representation:

```
#define PLATFORM_OVERRIDE_DMA_CNT   0
```

3.2 DMA master attributes

This section provides data for the DMA master 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;
```

Chapter 4
SMMU and device tests

This chapter provides an overview on SMMU and device tests. Additionally, it describes the parameters for 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 that can be customized.

It contains the following section:

4.1 SMMU and device tests

This chapter provides an overview on SMMU and the device tests.

Additionally, this chapter 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, I/O virtual address mapping.

This section contains the following subsections:

4.1.1 Number of IOVIRT Nodes

This section provides data for the number of IOVIRT nodes.

Header file representation:

```
#define IORT_NODE_COUNT 0x3
```

4.1.2 Number of SMMUs

This section provides data for the System Memory Management Unit (SMMU) count.

Header file representation:

```
#define SMMU_COUNT 0x1
```

4.1.3 Number of RC

This section provides details on the number of Root Complex (RC) count.

Header file representation:

```
#define RC_COUNT 0x1
```

4.1.4 Number of PMCG

This section provides details on the number of Performance Monitor Counter Groups (PMCG) in the system.

Header file representation:

```
#define PMCG_COUNT 0x1
```

4.1.5 Number of ITS blocks

This section provides details on the number of Interrupt Translation Service (ITS) blocks in GIC.

Header file representation:

```
#define IOVIRT_ITS_COUNT 0x1
```

4.1.6 I/O virtualization node-specific information

This section provides details on 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;
```

4.1.7 SMMU node-specific information

This section describes the 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;
```

4.1.8 RC-specific information

This section provides data on the 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;
```

4.1.9 I/O virtual address mapping

This section provides data on the 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;
```

Chapter 5
GIC tests

This chapter describes the parameters for Generic Interrupt Controller (GIC) specific test that can be customized.

It contains the following section:

5.1 GIC-specific tests

This section provides details on GIC-specific tests.

Header file representation:

```
#define PLATFORM_OVERRIDE_GICITS_COUNT       0x1
#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_GICC_BASE          0x30000000
#define PLATFORM_OVERRIDE_GICD_BASE          0x30000000
#define PLATFORM_OVERRIDE_GICRD_BASE         0x300C0000
#define PLATFORM_OVERRIDE_GICITS_BASE        0x30040000
#define PLATFORM_OVERRIDE_GICITS_ID          0
#define PLATFORM_OVERRIDE_GICIRD_LENGTH      (0x20000*8)
```

Chapter 6
Timer tests

This chapter describes the parameters for timer tests, and timer information that can be customized.

It contains the following sections:

6.1 Timer Tests

This section provides information on timer tests.

Header file representation:

```
#define PLATFORM_OVERRIDE_TIMER_COUNT 0x2
```

6.2 Timer information

This section provides information on the timers present in the system.

Header file representation:

```
#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
#define PLATFORM_OVERRIDE_PLATFORM_TIMER_COUNT  0x2
```

Chapter 7
Watchdog timer tests

This chapter describes the parameters for the number of watchdog timer tests, and watchdog information that can be customized.

It contains the following sections:

7.1 Number of watchdog timers

This section provides information on the number of watchdog timers present in the system.

Header file representation:

```
#define PLATFORM_OVERRIDE_WD_TIMER_COUNT 2
```

7.2 Watchdog information

This section provides information on the number of watchdog timers present in the system.

The following information about each of the watchdog timers is present on 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;
```

Chapter 8
Porting requirements

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

It contains the following sections:

8.1 PAL

This section provides information on different APIs in PAL.

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 test.

Table 8-1 PAL

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_taddr, 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

Note

The following points describes the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.2 GIC

This section provides information on different types of APIs in GIC.

The following table is the list of different APIs in GIC:

Table 8-2 GIC

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_tpal_gic_set_intr_trigger(uint32_tint_idINTR_TRIGGER_INFO_TYPE_etrigger_type);	Platform-specific

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.3 Timer

This section provides information on different types of APIs in timer.

The following table is the list of different APIs in timer:

Table 8-3 Timer

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

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.4 IOVIRT

This section provides information on different types of APIs in IOVIRT.

The following table is the list of different APIs in IOVIRT:

Table 8-4 IOVIRT

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_trc_seg_num);`	Yes

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.5 PCIe

This section provides information on different APIs in PCIe.

The following table is the list of different APIs in PCIe:

Table 8-5 PCIe

API name	Function prototype	Implementation
create_info_table	void pal_pcie_create_info_table(PCIE_INFO_TA BLE*PcieTable);	Yes
read_cfg	uint32_t pal_pcie_read_cfg(uint32_tbdf, uint32_t offset, uint32_t *data);	Yes
get_msi_vectors	uint32_t pal_get_msi_vectors(uint32_t seg,uint32_t bus, uint32_t dev, uint32_tfn, 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_tdev, uint32_tfn);	Yes
get_pcie_type	uint32_t pal_pcie_get_pcie_type(uint32_t seg,uint32_t bus, uint32_t dev, uint32_tfn);	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
multifunction_support	uint32_t pal_pcie_multifunction_support(uint32_t seg, uint32_t bus, uint32_t dev,uint32_t fn);	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

Table 8-5 PCIe (continued)

API name	Function prototype	Implementation
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_toffset, uint32_t *value)	Yes
pci_cfg_write	void pal_pci_cfg_write(uint32_t bus,uint32_t dev, uint32_t func, uint32_toffset, 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_tbus, uint32_t sec_bus)	Yes
get_bdf	uint32_t pal_pcie_get_bdf(uint32_tClassCode, uint32_t StartBdf)	Yes
increment_bus_dev	uint32_t pal_increment_bus_dev(uint32_tStartBdf)	Yes
get_base	uint64_t pal_pcie_get_base(uint32_t bdf,uint32_t bar_index)	Yes
bdf_to_dev	void *pal_pci_bdf_to_dev(uint32_t bdf)	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

Table 8-5 PCIe (continued)

API name	Function prototype	Implementation
get_snoop_bit	uint32_t pal_pcie_get_snoop_bit(uint32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
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(uint3 2_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
Is_devicedma_64bit	uint32_t pal_pcie_is_devicedma_64bit(uin t32_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes
get_legacy_irq_map	uint32_t pal_pcie_get_legacy_irq_map(uin t32_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(uint 32_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(uint3 2_t seg, uint32_t bus, uint32_t dev, uint32_t fn);	Yes

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.6 SMMU

This section provides information on different APIs in SMMU.

The following table is the list of different APIs in SMMU:

Table 8-6 SMMU

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_tsmmu_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_tsmmu_base, uint32_t pasid);	Platform-specific

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.7 Peripheral

This section provides information on different APIs in peripheral.

The following table is the list of different APIs in peripheral:

Table 8-7 Peripheral

API name	Function prototype	Implementation
`create_info_table`	`void pal_peripheral_create_info_table(PERIPHERAL_INFO_TABLE *per_info_table);`	Yes
`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
`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_get_unpopulated_addr`	`uint64_t pal_memory_get_unpopulated_addr(uint64_t *addr, uint32_t instance)`	Platform-specific

Note

The following describes the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.8 DMA

This section provides information on different APIs in DMA.

The following table is the list of different APIs in DMA:

Table 8-8 DMA

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_tlength, 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

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.9 Exerciser

This section provides information on different APIs in exerciser.

The following table is the list of different APIs in exerciser:

Table 8-9 Exerciser

API name	Function prototype	Implementation
pal_exerciser_get_ecsr_base	uint64_t pal_exerciser_get_ecsr_base(uint32_t Bdf,uint32_t BarIndex)	Platform-specific
pal_exerciser_get_pcie_config_offset	uint64_t pal_exerciser_get_pcie_config_offset(uint32_t Bdf)	Platform-specific
pal_exerciser_start_dma_direction	uint32_t pal_exerciser_start_dma_direction(uint64_t Base, EXERCISER_DMA_ATTRDirection)	Platform-specific
pal_exerciser_find_pcie_capability	uint32_t pal_exerciser_find_pcie_capability(uint32_t ID, uint32_t Bdf, uint32_t Value,uint32_t *Offset)	Platform-specific
pal_exerciser_set_param	uint32_t pal_exerciser_set_param(EXERCISER_PARAM_TYPE type, uint64_t value1, uint64_tvalue2, uint32_t bdf);	Platform-specific
pal_exerciser_get_param	uint32_t pal_exerciser_get_param(EXERCISER_PARAM_TYPE type, uint64_t *value1, uint64_t*value2, uint32_t bdf);	Platform-specific
pal_exerciser_set_state	uint32_t pal_exerciser_set_state(EXERCISER_STATEstate, uint64_t *value, uint32_t bdf);	Platform-specific
pal_exerciser_get_state	uint32_t pal_exerciser_get_state(EXERCISER_STATE*state, uint32_t bdf);	Platform-specific
pal_exerciser_ops	uint32_t pal_exerciser_ops(EXERCISER_OPS ops,uint64_t param, uint32_t instance);	Platform-specific
pal_exerciser_get_data	uint32_t pal_exerciser_get_data(EXERCISER_DATA_TYPE type, exerciser_data_t *data, uint32_tbdf, uint64_t ecam);	Platform-specific

Note

The following points describe the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

8.10 Miscellaneous

This section provides information on different APIs in miscellaneous.

The following table lists different APIs in miscellaneous:

Table 8-10 Miscellaneous

API name	Function prototype	Implementation
pal_mmio_read8	uint8_t pal_mmio_read8(uint64_t addr);	Yes
pal_mmio_read16	uint16_t pal_mmio_read16(uint64_t addr);	Yes
pal_mmio_read	uint32_t pal_mmio_read(uint64_t addr);	Yes
pal_mmio_read64	uint64_t pal_mmio_read64(uint64_t addr);	Yes
pal_mmio_write8	void pal_mmio_write8(uint64_t addr,uint8_t data);	Yes
pal_mmio_write16	void pal_mmio_write16(uint64_t addr,uint16_t data);	Yes
pal_mmio_write	void pal_mmio_write(uint64_t addr,uint32_t data);	Yes
pal_mmio_write64	void pal_mmio_write64(uint64_t addr,uint64_t data);	Yes
pal_print	void pal_print(char8_t *string, uint64_tdata);	Platform-specific
pal_print_raw	void pal_print_raw(uint64_t addr, char*string, uint64_t data)	Yes
pal_mem_free	void pal_mem_free(void *buffer);	Platform-specific
pal_mem_compare	int pal_mem_compare(void *src,void *dest, uint32_t len);	Yes
pal_mem_set	void pal_mem_set(void *buf, uint32_t size,uint8_t value);	Yes
pal_mem_allocate_shared	void pal_mem_allocate_shared(uint32_tnum_pe, uint32_t sizeofentry);	Yes
pal_mem_get_shared_addr	uint64_t pal_mem_get_shared_addr(void);	Yes
pal_mem_free_shared	void pal_mem_free_shared(void);	Yes
pal_mem_alloc	void *pal_mem_alloc(uint32_t size);	Platform-specific
pal_mem_virt_to_phys	void *pal_mem_virt_to_phys(void *va);	Platform-specific

Table 8-10 Miscellaneous (continued)

API name	Function prototype	Implementation
pal_mem_alloc_cacheable	void *pal_mem_alloc_cacheable(uint32_t Bdf, uint32_t Size, void **Pa);	Platform-specific
pal_mem_free_cacheable	void pal_mem_free_cacheable(uint32_t Bdf, uint32_t Size, void *Va, void *Pa);	Platform-specific
pal_mem_phys_to_virt	void *pal_mem_phys_to_virt (uint64_t Pa);	Platform-specific
pal_strncmp	uint32_t pal_strncmp(char8_t *str1, char8_t*str2, uint32_t len);	Yes
pal_memcpy	void *pal_memcpy(void *dest_buffer, void*src_buffer, uint32_t len);	Yes
pal_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 (uint32NumPages);	Platform-specific
free_pages	void pal_mem_free_pages (void *PageBase,uint32_t NumPages);	Platform-specific

Note

The following points describes the implementation type for this API:

1. Yes: This 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.
2. Platform-specific: You must implement all the APIs that are marked as platform-specific.

Chapter 9
SBSA ACS flow

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

It contains the following sections:

9.1 SBSA ACS flow diagram

This chapter provides information on the SBSA ACS flow diagram.

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

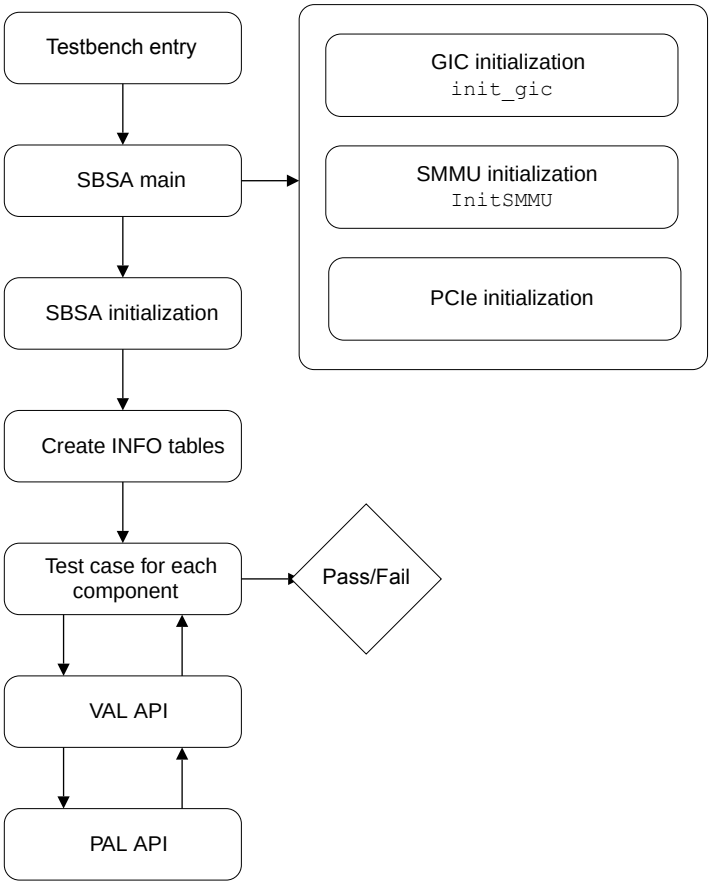

Figure 9-1 SBSA flow diagram

9.2 SBSA test example flow

This section shows the flow of a typical SBSA test.

This flow chart is for a test that checks MSI support of a PCIe device.

If the device is MSI enabled, the flag is set to MSI_ENABLED by the PAL layer. The test checks whether the device is of type endpoint and then checks whether the flags are set to MSI_ENABLED.

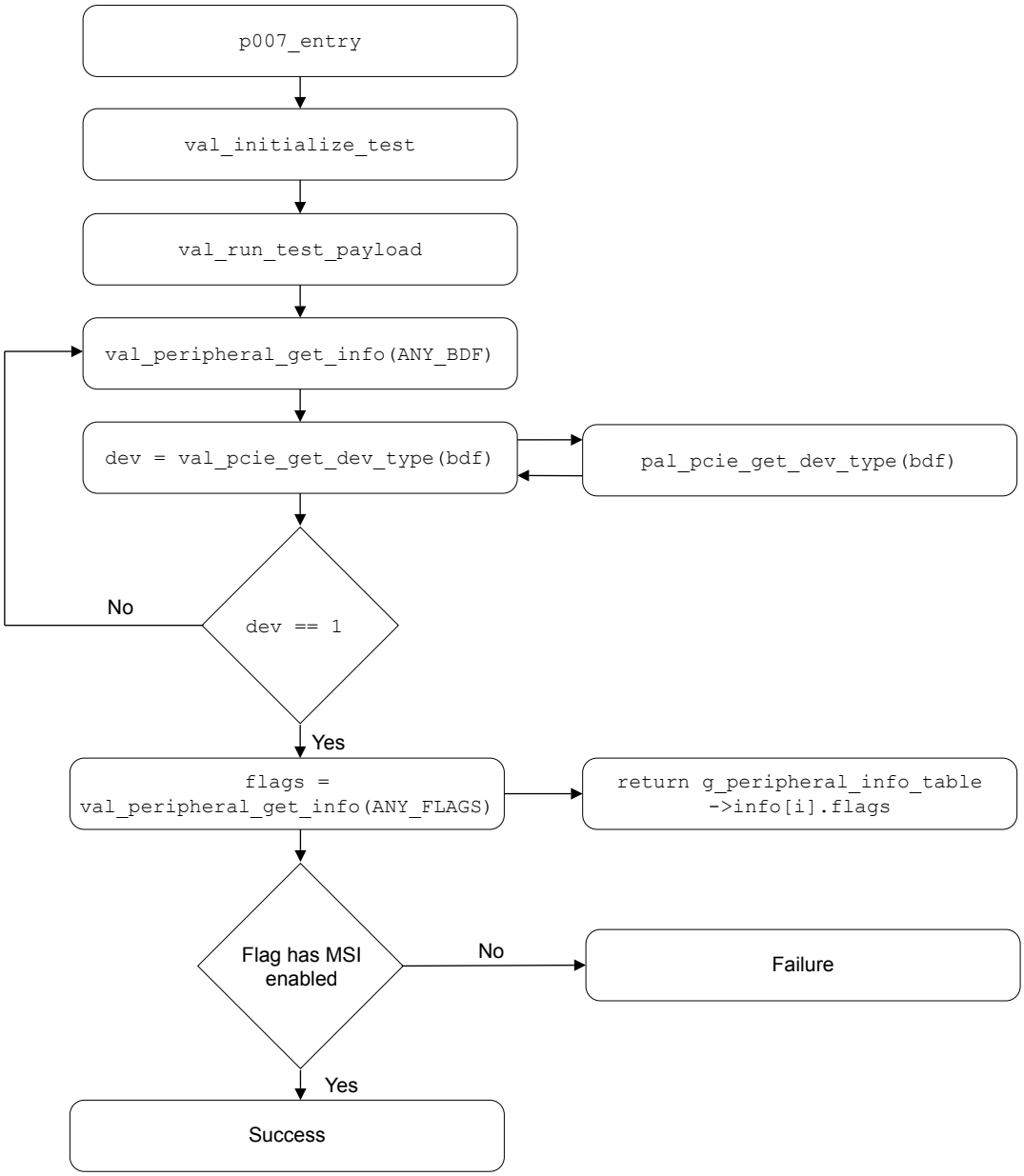

Figure 9-2 SBSA example flow diagram

Appendix A
Revisions

This section describes the technical changes made in this book.

It contains the following section:

- *A.1 Revisions* on page Appx-A-50.

A.1 Revisions

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, *1.3.1 Directory structure* on page 1-13.
Changed the node count in IOVIRT nodes.	See, *4.1.1 Number of IOVIRT Nodes* on page 4-21.
Added a few GIC-specific tests.	See, *5.1 GIC-specific tests* on page 5-24.
Removed a few APIs in GIC.	See, *8.2 GIC* on page 8-33.
Removed and added a few APIs in PCIe.	See, *8.5 PCIe* on page 8-36.
Removed the 'create_info_table' in SMMU.	See, *8.6 SMMU* on page 8-39.
Removed a few APIs from Peripheral.	See, *8.7 Peripheral* on page 8-40.
Renamed and added a few APIs in Miscellaneous.	See, *8.10 Miscellaneous* on page 8-44.