Arm® RME Architecture Compliance Bare-metal

Version 3.0

User Guide

Non-Confidential

Issue 01

109412_0300_01_en

Arm® RME Architecture Compliance Bare-metal
User Guide

Release Information

Document history

Issue	Date	Confidentiality	Change
0300-01	22 January 2025	Non-Confidential	EAC release
0200-01	15 November 2024	Non-Confidential	Beta release for issue B
0100-02	9 August 2024	Non-Confidential	First internal release
0100-01	13 December 2023	Non-Confidential	First release for v1.0
0700-01	6 November 2023	Non-Confidential	First release

Proprietary Notice

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.

Feedback on content

Information about how to give feedback on the content.

If you have comments on content then send an e-mail to support-rme-acs@arm.com. Give:

- The title Arm® RME Architecture Compliance Bare-metal User Guide.
- The number 109412_0300_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.

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

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

Note

This information is important and needs your attention.

Tip

This information might help you perform a task in an easier, better, or faster way.

Remember

This information reminds you of something important relating to the current content.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® RME Architecture Compliance User Guide	108005	Non-Confidential
Arm® RME Architecture Compliance Validation Methodology	108004	Non-Confidential

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com.

1.3 Other information

See the Arm® website for other relevant information.

- Arm® Developer.
- Arm® Documentation.

- Technical Support.
- Arm® Glossary.

2. Overview to RME ACS

This chapter provides an overview on Arm®Realm Management Extension (RME) ACS, the ACS design, and steps to customize the bare-metal code.

2.1 Abbreviations

The following table lists the abbreviations used in this document.

Table 2-1: Abbreviations and expansions

Abbreviation	Expansion
ACS	Architecture Compliance Suite
DA	Device Assignment
DMA	Direct Memory Access
DPT	Device Permission Table
ECAM	Enhanced Configuration Access Mechanism
GIC	Generic Interrupt Controller
IDE	Integrity and Data Encryption
IORT	Input Output Remapping Table
IOVIRT	Input Output Virtualization
ITS	Interrupt Translation Service
KM	Key Management
MPIDR	Multiprocessor ID Register
MSI	Message-Signaled Interrupt
PAL	Platform Abstraction Layer
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PMU	Performance Monitoring Unit
RAS	Reliability, Availability, and Serviceability
RC	Root Complex
RP	Root Port
RME	Realm Management Extension
RMM	Realm Management Monitor
RMSD	Realm Management Security Domain
MMU	Memory Management Unit
SoC	System on Chip
SMC	Secure Monitor Call
SMMU	System Memory Management Unit
TEE	Trusted Execution Environment

Abbreviation	Expansion
TDI TEE	Device Interface Trusted Execution Environment
TDISP TEE	Device Interface Security Protocol Trusted Execution Environment
UART	Universal Asynchronous Receiver and Transmitter
UEFI	Unified Extensible Firmware Interface
VAL	Validation Abstraction Layer

2.2 RME ACS

The RME architecture defines the set of hardware features and properties that are required to comply with the Arm CCA architecture. The Arm Confidential Compute Architecture (Arm CCA) enables the construction of protected execution environments called Realms. Realms allow lower-privileged software, such as application or a Virtual Machine to protect its content and execution from attacks by higher-privileged software, such as an OS or a hypervisor.

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 RME.

For more information on Arm® RME ACS, see the README.

2.3 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 example 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:

- GIC
- SMMU
- RME
- DA System

2.4 Boot framework

The bootwrapper is a simple implementation of a boot loader to boot up the system and transition to the ACS where specific set of tests are run.

The bootwrapper initializes the hardware and loads the ACS into the memory, allowing the system to start up, independent of UEFI and execute ACS tests automatically. This further reduces porting complexity for the partners and provides them with off-the-shelf system Init code.

2.4.1 Boot process and boot flow

The boot process is the sequence of operations that occurs when a computer system is powered-on or restarted, allowing it to transition from a power-off state to an operational state, where the operating system can run.

A boot loader, also known as a boot manager, is a piece of software responsible for initiating the boot process and loading the operating system into memory. Boot loader is located in firmware or a dedicated boot partition and is executed when the system is powered-on or restarted.

The cold boot path in this implementation of TF-A depends on the execution state. For AArch64, it is divided into five steps (in the order of execution):

- Boot Loader stage 1 (BL1) - AP Trusted ROM
- Boot Loader stage 2 (BL2) - Trusted Boot Firmware
- Boot Loader stage 3-1 (BL31) - EL3 Run time Software
- Boot Loader stage 3-2 (BL32) - Secure-EL1 Payload (optional)
- Boot Loader stage 3-3 (BL33) - Non-trusted Firmware

2.4.2 Boot framework for Bare-metal

With the introduction of bootwrapper, the UEFI layer is bypassed in the ACS boot flow. RME ACS with bootwrapper runs as non-trusted firmware at BL33.

The following figures show the overview of System boot flow and ACS boot flow in a system environment.

Figure 2-1: System boot flow

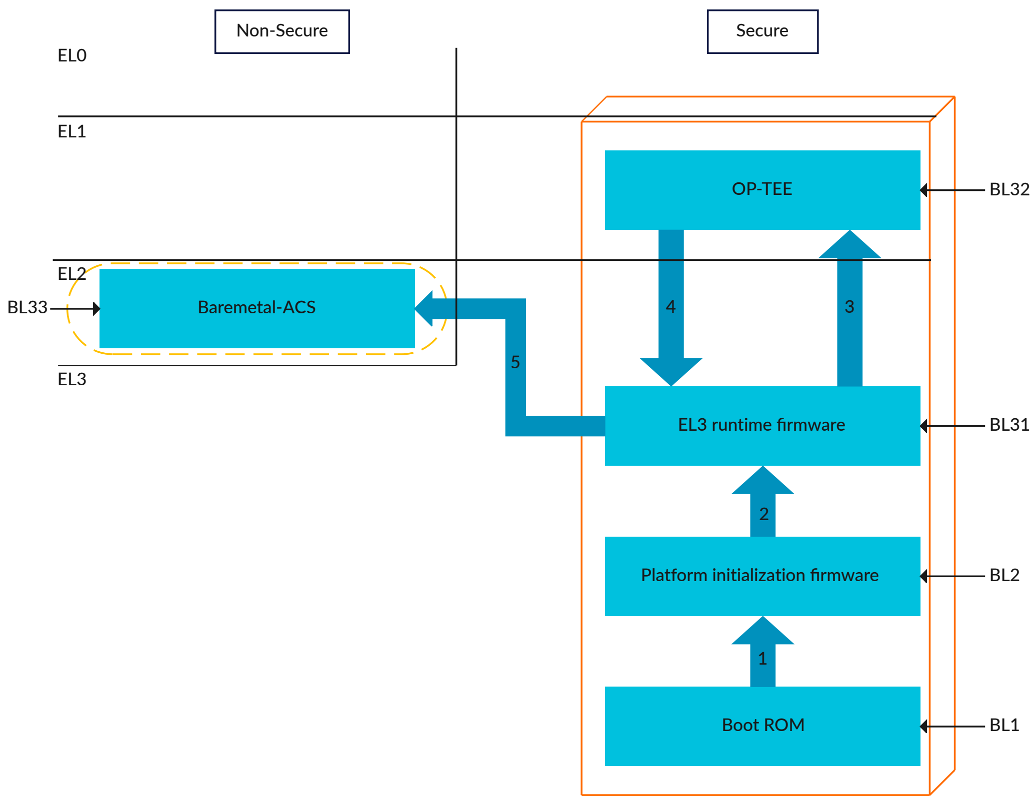
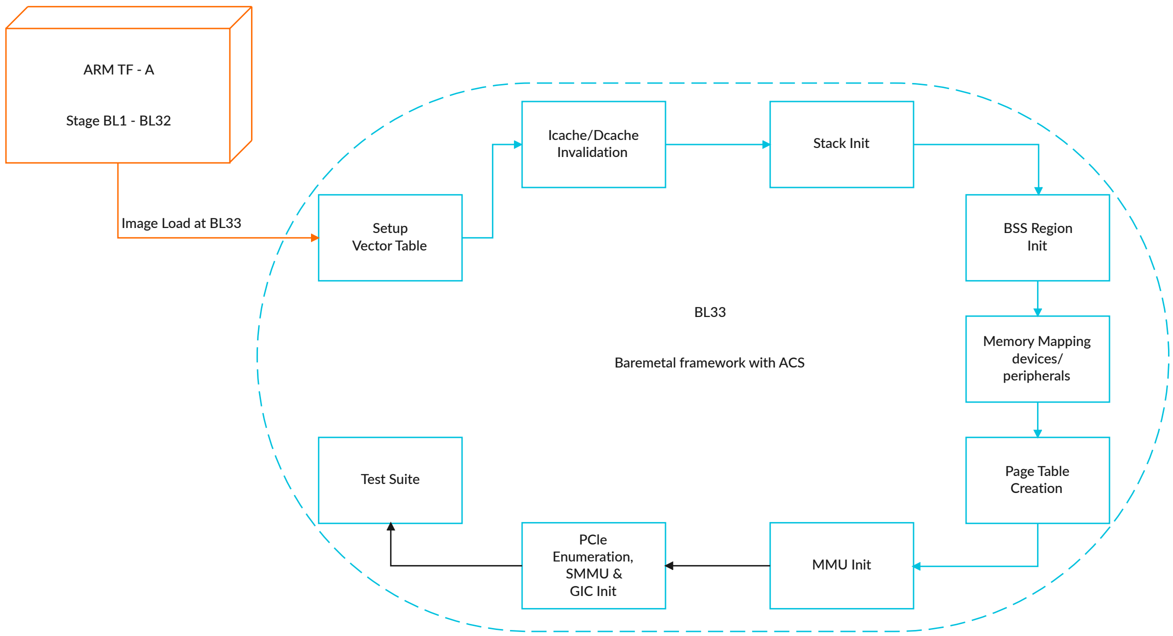


Figure 2-2: ACS Boot framework flow



2.5 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/FVP` 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.
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 RME ACS.

2.5.1 Test components

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

Table 2-2: Bare-metal components

Components	Files
DA	`pal_tdisp_spdm.c`
Exerciser	`pal_exerciser.c`
GIC	`pal_gic.c`
PCIe	`pal_pcie.c, pal_enumeration.c`
RAS	`pal_ras.c`
SMMU	`pal_smmu.c`
Legacy System	`pal_misc.c`

Note

PAL implementation requires porting when the underlying platform design changes.

3. Execution of RME ACS

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

3.1 SoC emulation environment

Executing RME 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.

3.1.1 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
```

Header file representation:

```
typedef struct {
uint32_t num_of_pe;
} PE_INFO_HDR;

/**
@brief structure instance for PE entry
**/
typedef struct {
uint32_t pe_num; ///< PE Index
uint32_t attr; ///< PE attributes
uint64_t mpidr; ///< PE MPIDR
uint32_t pmu_gsiv; ///< PMU Interrupt ID
} PE_INFO_ENTRY;

typedef struct {
PE_INFO_HDR header;
PE_INFO_ENTRY pe_info[];
} PE_INFO_TABLE;
```

3.1.1.1 MMU Configuration

This section provides information on the MMU for the PE MMU.

The parameters required for the PE MMU are populated as follows:

```
#define PLATFORM_PAGE_SIZE      0x1000
#define PLATFORM_OVERRIDE_MMU_PGT_IAS    48
#define PLATFORM_OVERRIDE_MMU_PGT_OAS    48
```

3.1.2 PCIe

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

```
#define PLATFORM_OVERRIDE_PCIE_ECAM0_HB_COUNT 1
#define PLATFORM_OVERRIDE_PCIE_ECAM0_SEG_NUM            0x0
#define PLATFORM_OVERRIDE_PCIE_ECAM0_START_BUS_NUM      0x0
#define PLATFORM_OVERRIDE_PCIE_ECAM0_END_BUS_NUM        0x8
#define PLATFORM_OVERRIDE_PCIE_ECAM0_EP_BAR64           0x4000100000
#define PLATFORM_OVERRIDE_PCIE_ECAM0_RP_BAR64           0x4000000000
#define PLATFORM_OVERRIDE_PCIE_ECAM0_EP_NPBAR32         0x60000000
#define PLATFORM_OVERRIDE_PCIE_ECAM0_EP_PBAR32          0x60600000
#define PLATFORM_OVERRIDE_PCIE_ECAM0_RP_BAR32           0x60850000
```

PLATFORM_OVERRIDE_PCIE_ECAMx_EP_BAR64:

The address required for 64-bit Prefetchable Memory Base for an PCIe End Point.

PLATFORM_OVERRIDE_PCIE_ECAMx_RP_BAR64:

The address required for 64-bit Prefetchable Memory Base for PCIe Type 1 devices.

PLATFORM_OVERRIDE_PCIE_ECAMx_EP_NPBAR32:

The address required for 32-bit Non-Prefetchable Memory Base for an PCIe End Point.

PLATFORM_OVERRIDE_PCIE_ECAMx_EP_PBAR32:

The address required for 32-bit Prefetchable Memory Base for an PCIe End Point.

PLATFORM_OVERRIDE_PCIE_ECAMx_RP_BAR32:

The address required for 32-bit Memory Base for a PCIe Type 1 devices.

Parameters required for the PCIe enumeration for a platform is populated as follows:

PLATFORM_OVERRIDE_NUM_ECAM:

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

PLATFORM_MAX_HB_COUNT:

Represents the maximum number of Host bridges 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
```

Header file representation:

```
typedef struct {
uint64_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;

typedef struct {
uint32_t num_entries;
PCIE_INFO_BLOCK block[];
} PCIE_INFO_TABLE;
```

3.1.2.1 PCIE device hierarchy table

This hierarchy table is used to obtain platform-specific support such as DMA, P2P and so on.

Parameters to be populated for each PCIe device is as follows:

```

PLATFORM_PCIE_DEVx_CLASSCODE     0x6040000
PLATFORM_PCIE_DEVx_VENDOR_ID     0x13B5
PLATFORM_PCIE_DEVx_DEV_ID        0xDEF
PLATFORM_PCIE_DEVx_BUS_NUM       0
PLATFORM_PCIE_DEVx_DEV_NUM       1
PLATFORM_PCIE_DEVx_FUNC_NUM      0
PLATFORM_PCIE_DEVx_SEG_NUM       0
PLATFORM_PCIE_DEVx_DMA_SUPPORT   0
PLATFORM_PCIE_DEVx_DMA_COHERENT  0
PLATFORM_PCIE_DEVx_P2P_SUPPORT   1
PLATFORM_PCIE_DEVx_DMA_64BIT     0
PLATFORM_PCIE_DEVx_BEHIND_SMMU   1
PLATFORM_PCIE_DEVx_ATC_SUPPORT   0
```

Header file representation:

```
typedef struct {
uint64_t class_code;
uint32_t device_id;
uint32_t vendor_id;
uint32_t bus;
uint32_t dev;
uint32_t func;
uint32_t seg;
uint32_t dma_support;
uint32_t dma_coherent;
uint32_t p2p_support;
uint32_t dma_64bit;
```

```
uint32_t behind_smmu;
uint32_t atc_present;
PERIPHERAL_IRQ_MAP irq_map;
} PCIE_READ_BLOCK;
```

3.1.3 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, Named component, PMCG, ITS blocks, I/O virtualization node-specific information, SMMU node-specific information, RC-specific information, and I/O virtual address mapping.

3.1.3.1 Number of IOVIRT Nodes

Parameters to be filled are:

```
#define IORT_NODE_COUNT 0x13
```

IORT_NODE_COUNT:

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

3.1.3.2 Number of SMMUs

Parameters to be filled are:

```
#define IOVIRT_SMMUV3_COUNT 5
```

```
#define IOVIRT_SMMUV2_COUNT 0
```

SMMU_COUNT:

Represents the number of SMMUs in the system.

3.1.3.3 Number of RCs

Parameters to be filled are:

```
#define RC_COUNT 0x1
```

RC_COUNT:

Represents the number of RCs present in the system.

3.1.3.4 Number of PMCGs

Parameters to be filled are:

```
#define PMCG_COUNT 0x1
```

PMCG_COUNT:

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

3.1.3.5 Number of named components

Parameters to be filled are:

```
#define IOVIRT_NAMED_COMPONENT_COUNT 2
```

IOVIRT_NAMED_COMPONENT_COUNT

Represents the number of named components present in the system.

3.1.3.6 Number of ITS blocks

Parameters to be filled are:

```
#define IOVIRT_ITS_COUNT 0x1
```

IOVIRT_ITS_COUNT:

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

3.1.3.7 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;

typedef union {
char name[MAX_NAMED_COMP_LENGTH];
IOVIRT_RC_INFO_BLOCK rc;
IOVIRT_PMCG_INFO_BLOCK pmcg;
uint32_t its_count;
SMMU_INFO_BLOCK smmu;
}NODE_DATA;
```

3.1.3.8 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.

3.1.3.9 Root Complex node 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;
```

3.1.3.10 PMCG node-specific information

Header file representation:

```
typedef struct {
uint64_t base;
uint32_t overflow_gsiv;
uint32_t node_ref;
} IOVIRT_PMCG_INFO_BLOCK;
```

3.1.3.11 Named component node specific information

Header file representation:

```
typedef struct {
uint64_t smmu_base; /* SMMU base to which component is attached, else NULL */
uint32_t cca; /* Cache Coherency Attribute */
char name[MAX_NAMED_COMP_LENGTH]; /* Device object name */
} IOVIRT_NAMED_COMP_INFO_BLOCK;
```

Named component specific information on Coresight components

Header file representation

```
typedef struct {
char identifier[MAX_CS_COMP_LENGTH]; // Hardware ID for Coresight ARM
 implementations
```

```
char dev_name[MAX_CS_COMP_LENGTH]; // Device name of Coresight components
} PLATFORM_OVERRIDE_CORESIGHT_COMP_INFO_BLOCK;

typedef struct {
PLATFORM_OVERRIDE_CORESIGHT_COMP_INFO_BLOCK component[CS_COMPONENT_COUNT];
} PLATFORM_OVERRIDE_CS_COMP_NODE_DATA;
```

3.1.3.12 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;
```

3.1.4 GIC

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

GIC-specific tests

Parameters to be filled are:

```
#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)
```

Header file representation:

```
typedef struct {
uint32_t gic_version;
uint32_t num_gicc;
uint32_t num_gicd;
uint32_t num_gicrd;
uint32_t num_gicits;
uint32_t num_gich;
uint32_t num_msiframes;
uint32_t gicc_type;
uint32_t gicd_type;
uint32_t gicrd_type;
```

```
uint32_t gicrd_length;
uint32_t gicits_type;
uint64_t gicc_base[PLATFORM_OVERRIDE_GICC_COUNT];
uint64_t gicd_base[PLATFORM_OVERRIDE_GICD_COUNT];
uint64_t gicrd_base[PLATFORM_OVERRIDE_GICRD_COUNT];
uint64_t gicits_base[PLATFORM_OVERRIDE_GICITS_COUNT];
uint64_t gicits_id[PLATFORM_OVERRIDE_GICITS_COUNT];
uint64_t gich_base[PLATFORM_OVERRIDE_GICH_COUNT];
uint64_t gicmsiframe_base[PLATFORM_OVERRIDE_GICMSIFRAME_COUNT];
uint64_t gicmsiframe_id[PLATFORM_OVERRIDE_GICMSIFRAME_COUNT];
uint32_t gicmsiframe_flags[PLATFORM_OVERRIDE_GICMSIFRAME_COUNT];
uint32_t gicmsiframe_spi_count[PLATFORM_OVERRIDE_GICMSIFRAME_COUNT];
uint32_t gicmsiframe_spi_base[PLATFORM_OVERRIDE_GICMSIFRAME_COUNT];
} PLATFORM_OVERRIDE_GIC_INFO_TABLE;
```

3.1.5 Timer

This section provides the parameters for timer-specific tests.

3.1.5.1 Timer information

Parameters to be filled are:

```
#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
```

Header file representation:

```
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_gsiv;
uint32_t ns_el1_timer_gsiv;
uint32_t el2_timer_gsiv;
uint32_t virtual_timer_flag;
uint32_t virtual_timer_gsiv;
uint32_t el2_virt_timer_gsiv;
uint32_t num_platform_timer;
uint32_t num_watchdog;
uint32_t sys_timer_status;
}TIMER_INFO_HDR;

typedef struct {
uint32_t type;
uint32_t timer_count;
uint64_t block_cntl_base;
uint8_t frame_num[8];
uint64_t GtCntBase[8];
uint64_t GtCntEl0Base[8];
uint32_t gsiv[8];
uint32_t virt_gsiv[8];
uint32_t flags[8];
}TIMER_INFO_GTBLOCK;
```

```
typedef struct {
TIMER_INFO_HDR header;
TIMER_INFO_GTBLOCK gt_info[];
}TIMER_INFO_TABLE;
```

3.1.6 Bare-metal boot requirements

This section provides information on the Bare-metal boot requirements of the system.

The following system-specific definitions must be filled to load bootable image.

Parameters to be filled are:

```
#define PLATFORM_OVERRIDE_WORLD_IMAGE_SIZE  0x4000000
#define PLARFORM_MEMORY_POOL_SIZE           (250 * 100000)
#define PLATFORM_SHARED_MEMORY_REGION       0x100000
#define PLATFORM_NORMAL_WORLD_IMAGE_BASE    0xE0000000
```

For more information on how to run RME ACS with bootwrapper code, see the README.

Note

PLATFORM_NORMAL_WORLD_IMAGE_BASE is the entry point to BL33.

4. Porting requirements

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

4.1 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.

4.1.1 PE

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

Table 4-1: 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
get_num	`uint32_t pal_pe_get_num();`	Yes

API name	Function prototype	Implementation
psci_get_conduit	`uint32_t pal_psci_get_conduit(void)`	Platform-specific

4.1.2 GIC

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

Table 4-2: 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

4.1.3 Timer

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

Table 4-3: 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
get_counter_frequency	`uint64_t pal_timer_get_counter_frequency(void);`	Yes

4.1.4 IOVIRT

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

Table 4-4: 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

API name	Function prototype	Implementation
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

4.1.5 PCIe

The following table lists the different types APIs in PCIe.

Table 4-5: 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
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
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 (uint32_t class_code, uint32_t start_bdf);`	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 seg, 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 seg, uint32_t bus, uint32_t dev, uint32_t func, uint32_t offset, uint32_t data)`	Yes
program_bar_reg	`pal_pcie_program_bar_reg(uint32_t seg, uint32_t bus, 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_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

API name	Function prototype	Implementation
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
check_device_valid	`uint32_t pal_pcie_check_device_valid(uint32_t bdf);`	Platform-specific
mem_get_offset	`uint32_t pal_pcie_mem_get_offset(uint32_t type);`	Yes
bar_mem_read	`uint32_t pal_pcie_bar_mem_read(uint32_t Bdf, uint64_t address, uint32_t *data);`	Yes
bar_mem_write	`uint32_t pal_pcie_bar_mem_write(uint32_t Bdf, uint64_t address, uint32_t data);`	Yes

4.1.6 SMMU

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

Table 4-6: 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
pa2iova	`uint64_t pal_smmu_pa2iova(uint64_t smmu_base, unit64_t pa);`	Platform-specific
smmu_disable	`uint32_t pal_smmu_disable(uint64_t smmu_base);`	Platform-specific
create_pasid_entry	`uint32_t pal_smmu_create_pasid_entry(uint64_t smmu_base, uint32_t pasid);`	Platform-specific

API name	Function prototype	Implementation
get_device_path	`uint32_t pal_get_device_path(const char *hid, char hid_path[][MAX_NAMED_COMP_LENGTH]);`	Yes
is_etr_behind_catu	`uint32_t pal_smmu_is_etr_behind_catu(char *etr_path);`	Platform-specific

4.1.7 Peripheral

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

Table 4-7: 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

4.1.8 Exerciser

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

Table 4-8: 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
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

API name	Function prototype	Implementation
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
device_lock	`uint32_t pal_device_lock(uint32_t bdf)`	Platform-specific
device_unlock	`uint32_t pal_device_unlock(uint32_t bdf)`	Platform-specific

4.1.9 Memory map

The following table lists the different types of APIs required for Memory Map.

Table 4-9: Memory map APIs and their details

API name	Function prototype	Implementation
add_mmap	`void pal_mmu_add_mmap(void);`	Platform-specific
get_mmap_list	`void *pal_mmu_get_mmap_list(void);`	Platform-specific
get_mapping_count	`uint32_t pal_mmu_get_mapping_count(void);`	Platform-specific

4.1.10 Miscellaneous

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

Table 4-10: 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

API name	Function prototype	Implementation
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
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
mem_calloc	`void *pal_mem_calloc(uint32_t num, uint32_t Size);`	Platform-specific
aligned_alloc	`void *pal_aligned_alloc( uint32_t alignment, uint32_t size );`	Platform-specific
mem_free_aligned	`void pal_mem_free_aligned(void *buffer);`	Platform-specific
strncpy	`void *pal_strncpy(void *DestinationStr, const void *SourceStr,uint32_t Length);`	Yes
uart_pl011_putc	`void pal_driver_uart_pl011_putc(int c);`	Yes

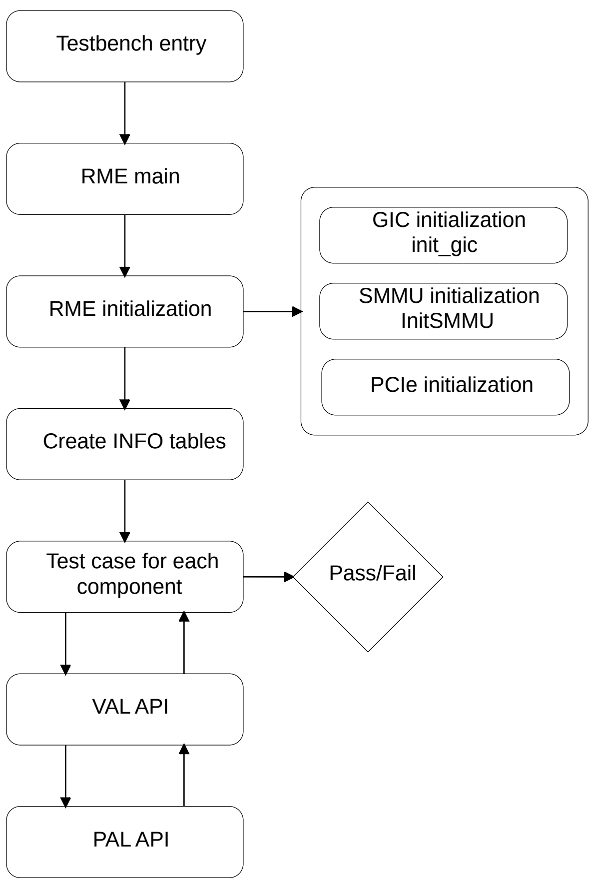
5. RME ACS flow

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

5.1 RME ACS flow diagram

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

Figure 5-1: RME flow diagram

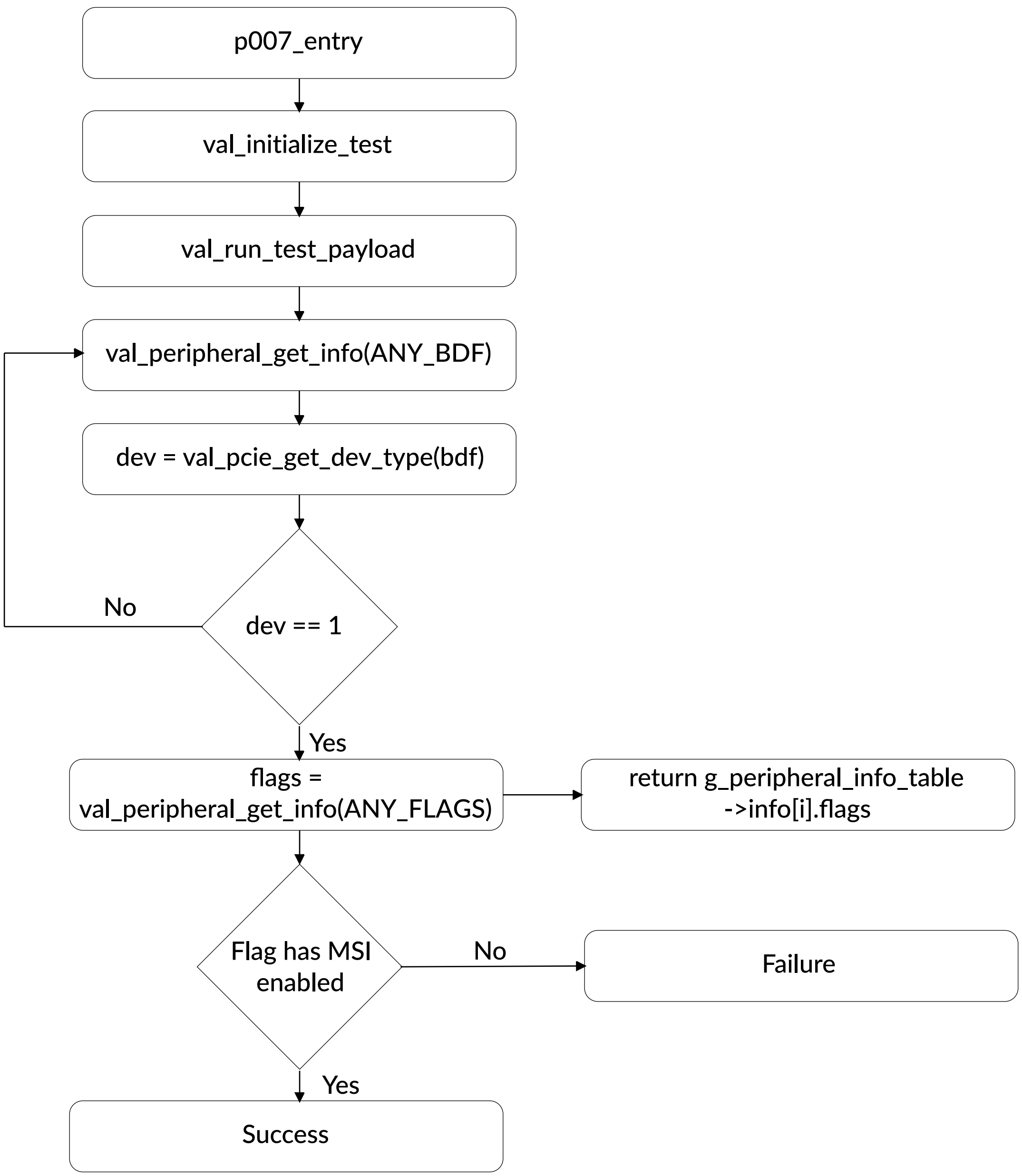


5.2 RME test example flow

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 5-2: RME example flow diagram



Appendix A Revisions

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

A.1 Revisions

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

Table A-1: Issue 0007-01

Change	Location
First release.	-

Table A-2: Difference between Issue 0007-01 and Issue 0100-01

Change	Location
No technical changes	-

Table A-3: Difference between Issue 0100-01 and Issue 0100-02

Change	Location
No technical changes	-

Table A-4: Difference between Issue 0100-02 and Issue 0200-01

Change	Location
Added new abbreviation.	See, 2.1 Abbreviations on page 10.
Added new APIs in the exerciser APIs and their details table.	See, 4.1.8 Exerciser on page 30.

Table A-5: Difference between Issue 0200-01 and Issue 0300-01

Change	Location
Added new abbreviations.	See, 2.1 Abbreviations on page 10.
Added new sections.	See, • 2.4 Boot framework on page 12 • 3.1.1.1 MMU Configuration on page 17 • 3.1.2 PCIe on page 17 • 3.1.6 Bare-metal boot requirements on page 25 • 4.1.9 Memory map on page 31
Updated functional prototype for few APIs in PCIe table.	See, 4.1.5 PCIe on page 28
Added new test components.	See, 2.5.1 Test components on page 14
Added new APIs in the exerciser and miscellaneous APIs and their details table.	See, • 4.1.8 Exerciser on page 30 • 4.1.10 Miscellaneous on page 31