Arm® SDEI Architecture Compliance

Validation Methodology

Non-Confidential – v1.0

Copyright © 2017, 2018 Arm Limited or its affiliates. All rights reserved.

PJDOC-2042731200-3102

Arm® SDEI Architecture Compliance

Validation Methodology

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
-	5 October 2017	Non-Confidential	Alpha
-	31 January 2018	Non-Confidential	Beta
-	16 May 2018	Non-Confidential	No changes for v1.0 release

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

Copyright © 2017, 2018, Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to. Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents
Arm® SDEI Architecture Compliance – Validation Methodology

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

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

It contains the following sections:

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this book

This book describes the Architecture Compliance Validation Methodology for the Arm SDEI platform design document.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is written for engineers who are specifying, designing, or verifying an implementation of the Arm SDEI architecture.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

Read this for an overview of SDEI compliance validation methodology.

Chapter 2 Execution model and flow control

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

Chapter 3 Validation Abstraction Layer

Read this for details about VAL and the VAL APIs.

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.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

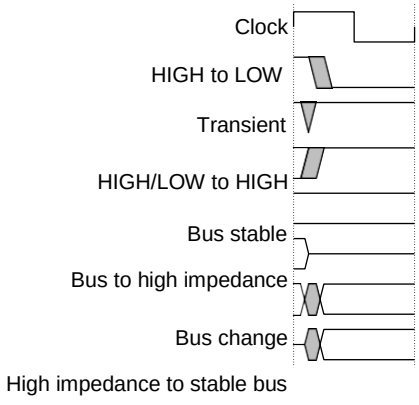

Figure 1 Key to timing diagram conventions

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name denotes an active-LOW signal.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional reading

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

Arm publications

- *Software Delegated Exception Interface* (ARM DEN0054A).
- *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile* (ARM DDI 0487).

Other publications

- http://www.uefi.org/specifications
  - ACPI Specification Version 6.1
  - UEFI Specification Version 2.6
  - UEFI Platform Initialization Specification Version 1.4 (Volume 2)

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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® SDEI ACS validation methodology*.
- The number PJDOC-2042731200-3102.
- 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.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 1
Introduction

This chapter provides an overview of the SDEI compliance validation methodology.

It contains the following sections:

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.1 About SDEI

Software Delegated Exception Interface (SDEI) provides a mechanism for registering and servicing system events from system firmware. This document defines a standard interface that is vendor-neutral, interoperable, and software portable. The interface is offered by a higher Exception level to a lower Exception level, in other words, by a Secure platform firmware to hypervisor or hypervisor to OS or both. System events are high priority events that must be serviced immediately by an OS or hypervisor. These events are often orthogonal to normal OS operation and can be handled even when the OS is executing within its own critical section with interrupts masked. System events can be provided to support:

- Platform error handling (RAS)
- Software watchdog timer
- Sample based profiling
- Kernel debugger

The goal of the compliance suite is to test the SDEI dispatcher for compliance against the SDEI platform design document.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.2 Acronyms

This section lists the acronyms that are used in this document.

Table 1-1 Acronyms and their definitions

Acronym	Expanded definition
ACPI	Advanced Configuration and Power Interface
ELx	Exception Level ‘x’ where x can be 0, 1, 2, or 3
GIC	Generic Interrupt Controller
PE	Processing Element
PPI	Private Peripheral Interrupt
PSCI	Power State Coordination Interface
SMC	Secure Monitor Call
SPI	Shared Peripheral Interrupt
UEFI	Unified Extensible Firmware Interface

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.3 Compliance test

The SDEI compliance suite is a set of targeted C-based tests that can be compiled to run at either a UEFI Shell or as a Linux driver.

The SDEI dispatcher can be implemented either at EL3 or EL2, or both, depending on the platform configuration. Therefore, SDEI compliance of the dispatcher must be tested from EL2 (either UEFI Shell or Hypervisor) and EL1 (from a Guest OS).

- To check the compliance of the dispatcher running at EL3, the compliance suite must be built to run at EL2 from a UEFI Shell.
- To test the compliance of the dispatcher running at EL2, the compliance suite must be built as Linux driver and run from EL1.

The same set of tests is executed from EL2 and EL1. So, it is imperative that the tests do not make any assumption on the Exception level that they are running at.

Since the same set of tests can be compiled to run either as a UEFI application or as a Linux driver, the tests must not directly call any environment-specific API. All the environment-specific interaction must be contained within the Platform Abstraction Layer.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.4 Layered software stack

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

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

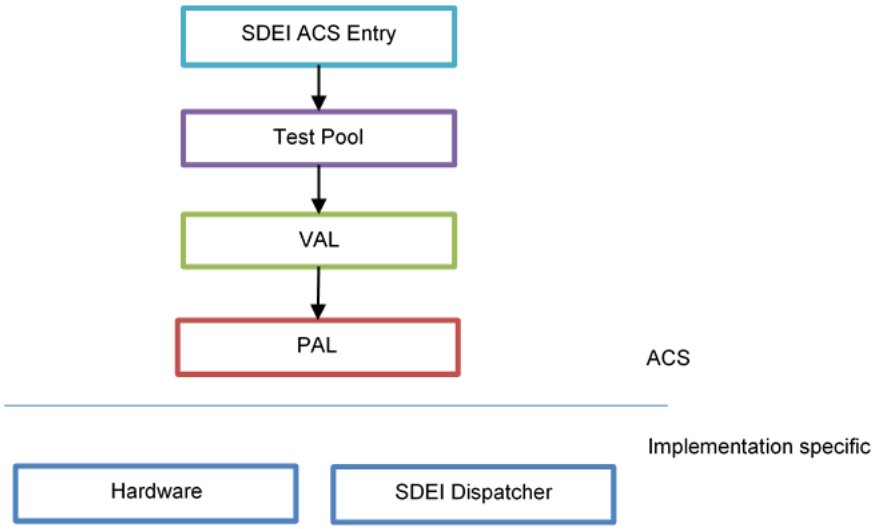

Figure 1-1 Layered software stack for compliance suite

The different layers of compliance tests are described in the following table.

Table 1-2 Different layers of compliance test

Layer	Description
Test pool	A collection of targeted tests that validate the compliance of the target system. These tests use interfaces that are provided by the VAL layer.
VAL	Provides a uniform view of all the underlying hardware and test infrastructure to the test pool.
PAL	Abstracts features whose implementation varies from one target system to another. The PAL is a C-based API that is defined by Arm and implemented by you. Each test platform requires a PAL implementation of its own. The PAL APIs are meant for the compliance test to reach or use other abstractions in the test platform such as the UEFI infrastructure and bare-metal abstraction.

Figure 1-2 shows the compliance test software stack with UEFI Shell application as an example.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

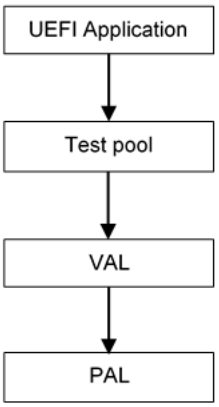

Figure 1-2 Compliance test software stack with UEFI Shell application

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

The stack is spread across user-mode and kernel-mode space. The Linux command-line application that runs in user-mode space and the Kernel module communicate using a `procfs` interface. The test pool, VAL, and PAL layers are built as a kernel module.

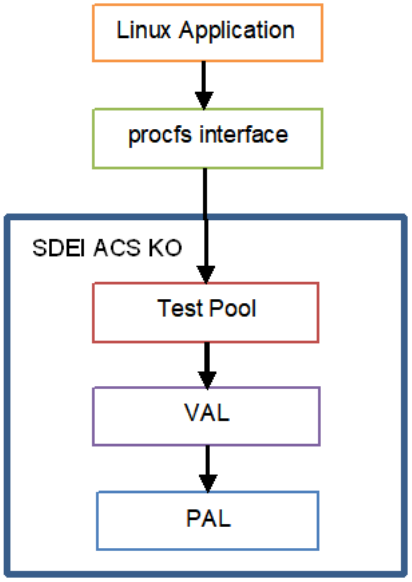

Figure 1-3 Compliance test software stack with Linux application

The SDEI command-line application initiates the tests and queries for status of the test using the standard `procfs` interface of the Linux OS. To avoid multiple data transfers between kernel and user mode, the test pool, VAL, and Linux PAL are together built as a kernel module.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Coding guidelines

The coding guidelines that are followed for the implementation of the test pool are:

- All the tests call VAL layer APIs.
- VAL layer APIs call PAL layer APIs depending on the functionality requested.
- A test does not directly interface with PAL layer functions.
- The test layer does not need any code modifications when porting from one platform to another.
- All the platform porting changes are limited to PAL layer.
- The VAL layer requires changes if there are architectural changes impacting multiple platforms.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.5 Test platform abstraction

The compliance suite defines and uses the test platform abstraction that is illustrated in figure 1-4.

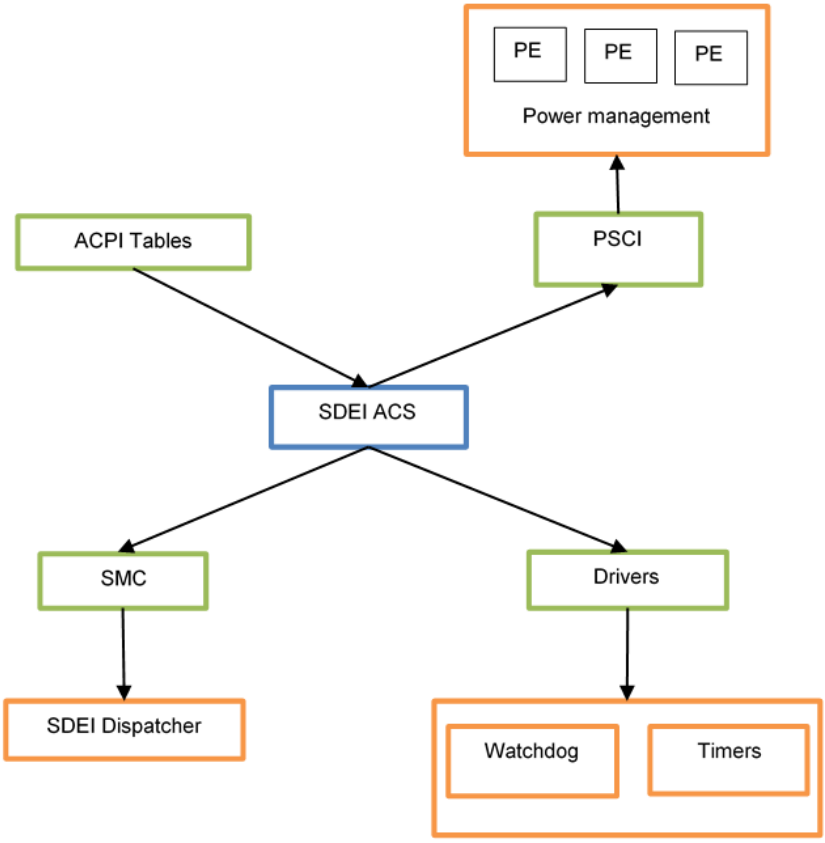

Figure 1-4 Test platform abstraction

Table 1-3 describes the SDEI abstraction terms.

Table 1-3 SDEI abstraction terms

Abstraction	Description
Drivers	Layer that abstracts the access to hardware enabling the compliance suite to generate events and interrupts.
SMC	Arm Architecture call to communicate with EL3 firmware.
ACPI tables	Interface layer that provides platform-specific information, removing the need for the test pool to be ported on a per platform basis.
PSCI	Arm specification that abstracts the power management implementation of PEs.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2
Execution model and flow control

This chapter provides details about the execution control mode and flow control scheme used by the compliance, different test platform variants, and ideas for implementing the PAL layer.

It contains the following sections:

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1 Execution model and flow control

Figure 2-1 illustrates the execution model of the compliance suite and the flow control used.

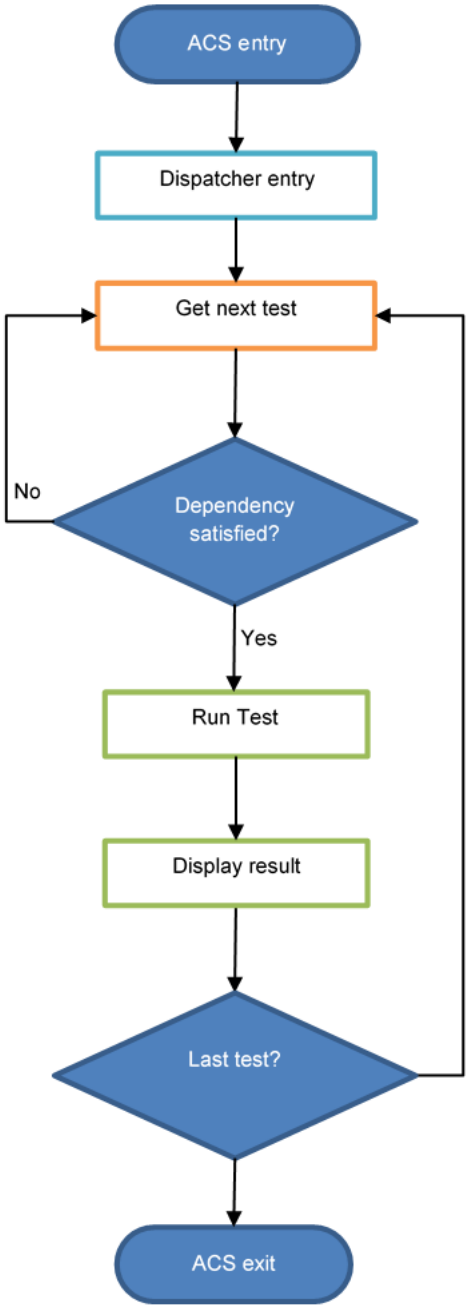

Figure 2-1 Execution model and flow control

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1. The execution environment (for example, UEFI Shell) invokes the test entry point.
2. Start the Test iteration loop.
3. If test has dependency, check if its resolved. If not, skip the test. If yes, execute it.
4. Report status after test execution as required.
5. Put the system to sleep if required.
6. Loop until all the tests are completed.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2 Test build and execution flow

2.2.1 Prerequisites

To build the SDEI compliance suite as a UEFI Shell application, a UEFI EDK2 source tree is required. To build the SDEI compliance suite kernel module, Linux kernel tree version 4.10 or above is required.

For details, see https://github.com/ARM-software/arm-enterprise-acs/sdei-acs/blob/master/README.md

2.2.2 Source code directory

Figure 2-2 shows the source code directory structure for the SDEI ACS.

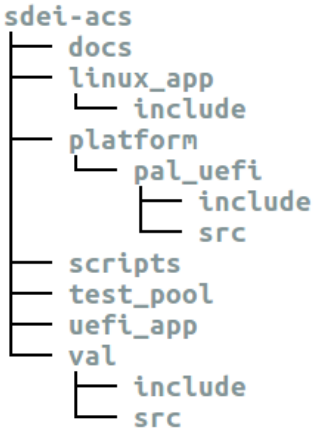

Figure 2-2 SDEI ACS directory structure

Table 2-1 SDEI ACS directory contens

Directory	Contents
pal_uefi	Platform code targeting UEFI implementation.
val	Common code used by the tests. Makes calls to PAL as needed.
uefi_app	UEFI application source to call into the tests entry point.
test_pool	Test case source files for test pool for application development.
linux_app	Linux command-line executable source code.
docs	Documentation.
scripts	Scripts written for this suite.

2.2.3 Test Build – UEFI

UEFI Shell Application

The build steps for the compliance suite to be compiled as a UEFI Shell application are in the README.

2.2.4 Test Build – OS based tests

Linux Application

The build steps for the Linux application driven compliance suite are detailed within the README.

Linux Kernel Module

The build steps for the SDEI compliance suite kernel module which is a dependency for the SDEI compliance suite Linux application are also part of the README.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3
Validation Abstraction Layer

This chapter describes VAL APIs.

It contains the following sections:

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1 About VAL

The VAL provides the APIs that are used by the test suite to implement the tests that can verify SDEI compliance. These APIs are platform-agnostic and use PAL APIs to interact with the hardware.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2 VAL API

This section describes the VAL APIs that the test suite uses to implement the test scenarios.

3.2.1 PE

This API provides the information and functionality required by the test suite that accesses features of a PE.

Table 3-1 PE APIs and their descriptions

API name	Function prototype	Description
`val_pe_create_info_table`	`acs_status_t val_pe_create_info_table(uint64_t *pe_info_table)`	Fills in the PE_INFO table with the information of the PEs in the system. For related definitions, see *Note*.
`val_pe_execute_on_all`	`acs_status_t val_pe_execute_on_all(void *payload, uint64_t arg)`	Executes code addressed by given function pointer on each PE in the system.
`val_pe_execute`	`acs_status_t val_pe_execute(uint32_t index, void *payload, uint64_t arg)`	Executes code addressed by given function pointer on given PE.
`val_pe_reg_read`	`acs_status_t val_pe_reg_read(pe_reg_id_t reg_id, uint64_t *output)`	Reads the PE register for the given `reg_id`. For related definitions, see *Note*.
`val_pe_get_mpid`	`acs_status_t val_pe_get_mpid(void)`	Returns the MPIDR of the current PE.
`val_pe_mpid_to_index`	`acs_status_t val_pe_get_mpid_index(uint64_t mpid, uint32_t *index)`	Converts MIPDR value to PE index.
`val_pe_index_to_mpid`	`acs_status_t val_pe_get_index_to_mpid(uint32_t index, uint64_t *mpid)`	Converts PE index to MPIDR value.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The related definitions for val_pe_create_info_table and val_pe_reg_read are as shown in the following table.

Table 3-2 Related definitions for PE APIs

API Name	Related definitions

val_pe_create_info_table	typedef struct {
	  uint32_t num_of_pe;
	} pe_info_hdr t;
	
	typedef struct {
	  uint32_t pe_num;
	  uint64_t mpidr;
	} pe_info_entry_t;
	
	typedef struct {
	  pe_info_hdr_t header;
	  pe_info_entry_t pe_info[];
	} pe_info_table_t;

val_pe_reg_read	typedef enum { X0, X1, …, …, X17, AA64PFR0_EL1, …, …, } pe_reg_id_t;

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2.2 GIC

These APIs are used to access the GIC to configure and manage interrupts. They provide the capability to the test suite to access interrupt information in the system and use interrupts test implementation.

Table 3-3 GIC APIs and their descriptions

API name	Function prototype	Description
`val_gic_create_info_table`	`acs_status_t val_gic_create_info_table(uint64_t *interrupt_info_table)`	Creates the GIC info table. For related definitions, see *Note*.
`val_gic_free_info_table`	`acs_status_t val_gic_free_info_table(void)`	Frees the GIC info table.
`val_get_gicd_base`	`acs_status_t val_get_gicd_base(uint64_t *gicd_base)`	Gets the GIC distributor base.
`val_gic_install_isr`	`acs_status_t val_gic_install_isr(uint32_t intr_id, void *isr)`	Installs interrupt service routine for given interrupt id.
`val_gic_end_of_interrupt`	`acs_status_t val_gic_end_of_interrupt(uint32_t int_id)`	Signals end of interrupt in GIC.
`val_gic_route_interrupt_to_pe`	`acs_status_t val_gic_route_interrupt_to_pe(uint32_t int_id, uint64_t mpidr)`	Routes interrupts to specific PE with given MPIDR value.
`val_gic_get_interrupt_state`	`acs_status_t val_gic_get_interrupt_state(uint32_t int_id, uint32_t *state)`	Gets the interrupt status for a given interrupt id.
`val_gic_clear_interrupt`	`acs_status_t val_gic_clear_interrupt(uint32_t int_id)`	Clears interrupt status registers in GIC for a given interrupt id.
`val_gic_disable_interrupt`	`acs_status_t val_gic_disable_interrupt(uint32_t int_id)`	Disables the interrupt in GIC for given interrupt id.
`val_gic_cpuif_init`	`acs_status_t val_gic_cpuif_init(void)`	Initializes GIC CPU interface.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The related definition for `val_gic_create_info_table` is as shown in the following table.

Table 3-4 Related definitions for GIC APIs

API Name	Related definitions

`val_gic_create_info_table`	`typedef struct {`
	`  uint32_t gic_version;`
	`  uint32_t num_gicd;`
	`  uint32_t num_gicr ;`
	`  uint32_t num_its ;`
	`} gic_info_hdr_t;`
	
	`typedef struct {`
	` uint32_t  type;`
	` uint64_t  base;`
	`  } gic_info_entry_t;`
	
	`typedef struct {`
	`  gic_info_hdr_t  header;`
	`  gic_info_entry_t  gic_info[];`
	`} gic_info_table_t;`

3.2.3 Print

This API provides capability to the test suite to log messages based on verbosity level.

Table 3-5 Print APIs and their descriptions

API Name	Function prototype	Description	Related Definitions

`val_print`	`void val_print(log_level_t level, char *fmt, …)`	Prints messages based on logging level defined by application.	`typedef enum {`
			`  ACS_LOG_ERR,`
			`  ACS_LOG_WARN,`
			`  ACS_LOG_TEST,`
			`  ACS_LOG_DEBUG,`
			`  ACS_LOG_INFO`
			`} log_level_t;`

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2.4 SDEI

These APIs provide capability to the test suite to make SDEI calls to the dispatcher, and use it to implement test cases that verify SDEI compliance.

Table 3-6 SDEI APIs and their descriptions

API Name	Function Prototype	Description
val_sdei_event_create_info_table	acs_status_t val_sdei_event_create_info_table(uint64_t * event_info_table)	Creates a table of information on various SDEI events in the system. For related definitions, see *Note*.
val_sdei_free_event_info_table	acs_status_t val_sdei_free_event_info_table (uint64_t *event_info_table)	Frees table of information on various SDEI events in the system.
val_sdei_version	acs_status_t val_sdei_version(uint64_t *version)	Gets version of SDEI dispatcher.
val_sdei_event_register	acs_status_t val_event_register(sdei_event_t *event, sdei_event_callback *cb, void *arg)	Registers a given event with the dispatcher. For related definitions, see *Note*.
val_sdei_event_enable	acs_status_t val_sdei_event_enable(uint32_t event_num)	Enables a given event so that registered handler is called when event occurs.
val_sdei_event_disable	acs_status_t val_sdei_event_disable(uint32_t event_num)	Disables a given event.
val_sdei_event_context	acs_status_t val_sdei_event_context(pe_reg_id_t param_id, uint64_t *param)	Reads value stored in given register (x0 - x17), based on param_id.
val_sdei_event_complete	acs_status_t val_sdei_event_complete(uint32_t status_code)	Marks the event complete, with completion status. It is called from event handler. Execution resumes in the context interrupted by the event.
val_sdei_event_complete_and_resume	acs_status_t val_sdei_event_complete_and_resume(uint64_t resume_addr)	Marks the event complete. Execution resumes from resume_addr.
val_sdei_event_unregister	acs_status_t val_sdei_event_unregister(uint32_t event_num)	Unregisters the given event with the dispatcher.
val_sdei_event_status	acs_status_t val_sdei_event_status(uint32_t event_num, uint64_t *status)	Retrieves status of the event, as one or more of running, enabled or registered.
val_sdei_event_get_info	acs_status_t val_sdei_event_get_info(uint32_t event_num, event_info_type_t info, int64_t *result)	Retrieves information about given event based on info parameter.
val_sdei_event_routing_set	acs_status_t val_sdei_event_routing_set(uint32_t event_num, uint64_t routing_mode, uint64_t affinity)	Changes the routing information of given event.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

API Name	Function Prototype	Description
val_sdei_mask	acs_status_t val_sdei_mask(int64_t *masked)	Masks the PE from receiving any event.
val_sdei_unmask	acs_status_t val_sdei_unmask(void)	Unmasks the PE to receive events.
val_sdei_interrupt_bind	acs_status_t val_sdei_interrupt_bind(uint32_t intr_num, event_t *event)	Binds the given interrupt to given event.
val_sdei_interrupt_release	acs_status_t val_sdei_interrupt_release(uint32_t event_num)	Releases the interrupt bound to given input.
val_sdei_event_signal	acs_status_t val_sdei_event_signal(uint32_t event_num, uint64_t target_pe)	Signals software event to given target PE.
val_sdei_features	acs_status_t val_sdei_features(uint32_t feature, uint64_t *val)	Queries SDEI features implemented by the dispatcher.
val_sdei_private_reset	acs_status_t val_sdei_private_reset(void)	Resets SDEI private data and of the calling API.
val_sdei_shared_reset	acs_status_t val_sdei_shared_reset(void)	Resets shared SDEI data.

Note

The related definitions for `val_sdei_event_create_info_table` and `val_sdei_event_register` are as shown in the following table.

Table 3-7 Related definitions for SDEI APIs

API Name	Related definitions

val_sdei_event_create_info_table	typedef struct {
	  uint32_t num_events;
	} event_info_hdr_t;
	
	typedef struct {
	 uint32_t number;
	 uint32_t type;
	 uint32_t priority;
	 uint32_t bound_interrupt;
	  } event_info_t;
	
	typedef struct {
	  event_info_hdr_t header;
	  event_info_t info[];
	} event_info_table_t;

val_sdei_event_register	typedef struct { uint32_t number; uint64_t type; uint64_t priority; bool is_bound_irq; } sdei_event_t; typedef int (*sdei_event_callback)(uint32_t event, void *arg);

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2.5 Shared Memory

These APIs provide shared memory access to all PEs in the system.

Table 3-8 Shared memory APIs and their descriptions

API name	Function prototype	Description
`val_shared_mem_alloc`	`acs_status_t val_shared_mem_alloc(void)`	Allocates memory that is shared by all PEs.
`val_shared_mem_read`	`acs_status_t val_shared_mem_read(uint32_t index, pe_shared_mem_t *mem)`	Reads data from shared memory at the index that is provided as argument.
`val_shared_mem_write`	`acs_status_t val_shared_mem_write(uint32_t index, pe_shared_mem_t *mem)`	Writes data to shared memory at the index provided as argument. For related definitions, see *Note*.
`val_shared_mem_free`	`void val_shared_mem_free(void)`	Frees shared memory.

Note

The related definitions for `val_shared_mem_write` are as shown in the following table.

Table 3-9 Related definitions for shared memory APIs

API Name	Related definitions

`val_shared_mem_write`	`typedef struct {`
	`uint64_t data0;`
	`uint64_t data1;`
	`uint32_t status;`
	`}`

3.2.6 Memory Mapped I/O

These APIs provide I/O access to memory mapped registers in the system.

Table 3-10 Memory mapped I/O APIs and their descriptions

API name	Function prototype	Description
`val_mmio_read`	`uint32_t val_mmio_read(uint64_t addr)`	Reads memory mapped register at given address.
`val_mmio_write`	`uint32_t val_mmio_write(uint64_t addr, uint32_t data)`	Writes given data at memory mapped register at given address.

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2.7 Watchdog

These APIs provide access to the watchdog timer configuration registers for managing watchdog timer interrupts.

Table 3-11 Watchdog APIs and their descriptions

API name	Function prototype	Description
`val_wd_create_info_table`	`acs_status_t val_wd_create_info_table(uint64_t *wd_info_table)`	Creates the watchdog info table. For related definitions, see *Note*.
`val_wd_get_info`	`uint64_t val_wd_get_info(uint32_t index, WD_INFO_TYPE info_type)`	Creates the watchdog info table. For related definitions, see *Note*.
`val_wd_enable`	`void val_wd_enable(uint32_t index)`	Enables the watchdog timer.
`val_wd_disable`	`void val_wd_disable(uint32_t index)`	Disables the watchdog timer.
`val_wd_set_ws0`	`void val_wd_set_ws0(uint64_t *vaddr, uint32_t index, uint32_t timeout)`	Sets the watchdog timeout value.

Note

The related definitions for `val_wd_create_info_table` and `val_wd_get_info` are as shown in the following table.

Table 3-12 Related definitions for watchdog APIs

API Name	Related definitions

`val_wd_create_info_table`	`typedef struct {`
	`  uint32_t num_wd;`
	`} gic_info_hdr_t;`
	
	`typedef struct {`
	`  uint64_t wd_ctrl_base;`
	`  uint64_t wd_refresh_base;`
	`  uint32_t wd_gsiv;`
	`  uint32_t wd_flags;`
	`} wd_info_entry_t;`
	
	`typedef struct {`
	`  wd_info_hdr_t header;`
	`  wd_info_entry_t wd_info[];`
	`} wd_info_table_t;`

`val_wd_get_info`	`typedef enum {`
	`  WD_INFO_COUNT = 1,`
	`  WD_INFO_CTRL_BASE,`
	`  WD_INFO_REFRESH_BASE,`
	`  WD_INFO_GSIV,`
	`  WD_INFO_ISSECURE`
	`} WD_INFO_TYPE;`

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2.8 Test

These APIs provide the interface to configure and manage tests.

Table 3-13 Test APIs and their descriptions

API name	Function prototype	Description
`val_test_init`	`void val_test_init(sdei_test_control *control`	Initializes the test infrastructure. For related definitions, see *Note*.
`val_test_enable`	`void val_test_enable(sdei_test_control *control, int test_id)`	Enables a test for a given test id.
`val_test_disable`	`void val_test_disable(sdei_test_control *control, int test_id)`	Disables a test for a given test id.
`val_test_pe_set_status`	`void val_test_pe_set_status(uint32_t index, uint32_t status)`	Sets test status for a given PE.
`val_test_pe_get_status`	`uint32_t val_test_pe_get_status(uint32_t index)`	Gets test status for a given PE.
`val_test_get_status`	`uint32_t val_test_get_status(uint32_t num_pe, uint64_t timeout)`	Gets test status for a given number of PEs. It keeps polling for a change of status from pending to a known termination status.

Note

The related definitions for `val_test_init` are as shown in the following table.

Table 3-14 Related definitions for test APIs

API Name	Related definitions

`val_test_init`	`typedef struct sdei_test_control {`
	`  test_flags flags[2];`
	`  unsigned int tests_skipped;`
	`  unsigned int tests_passed;`
	`  unsigned int tests_failed;`
	`  unsigned int tests_aborted;`
	`}sdei_test_control;`
	
	`typedef struct sdei_test_desc {`
	`  uint32_t id;`
	`  char description[124];`
	`  sdei_test_deps *deps;`
	`  uint32_t status;`
	`  sdei_test_fn test_fn;`
	`  int32_t all_pe;`
	`} sdei_test_desc;`

Copyright © 2017, 2018, Arm Limited or its affiliates. All rights reserved.
Non-Confidential