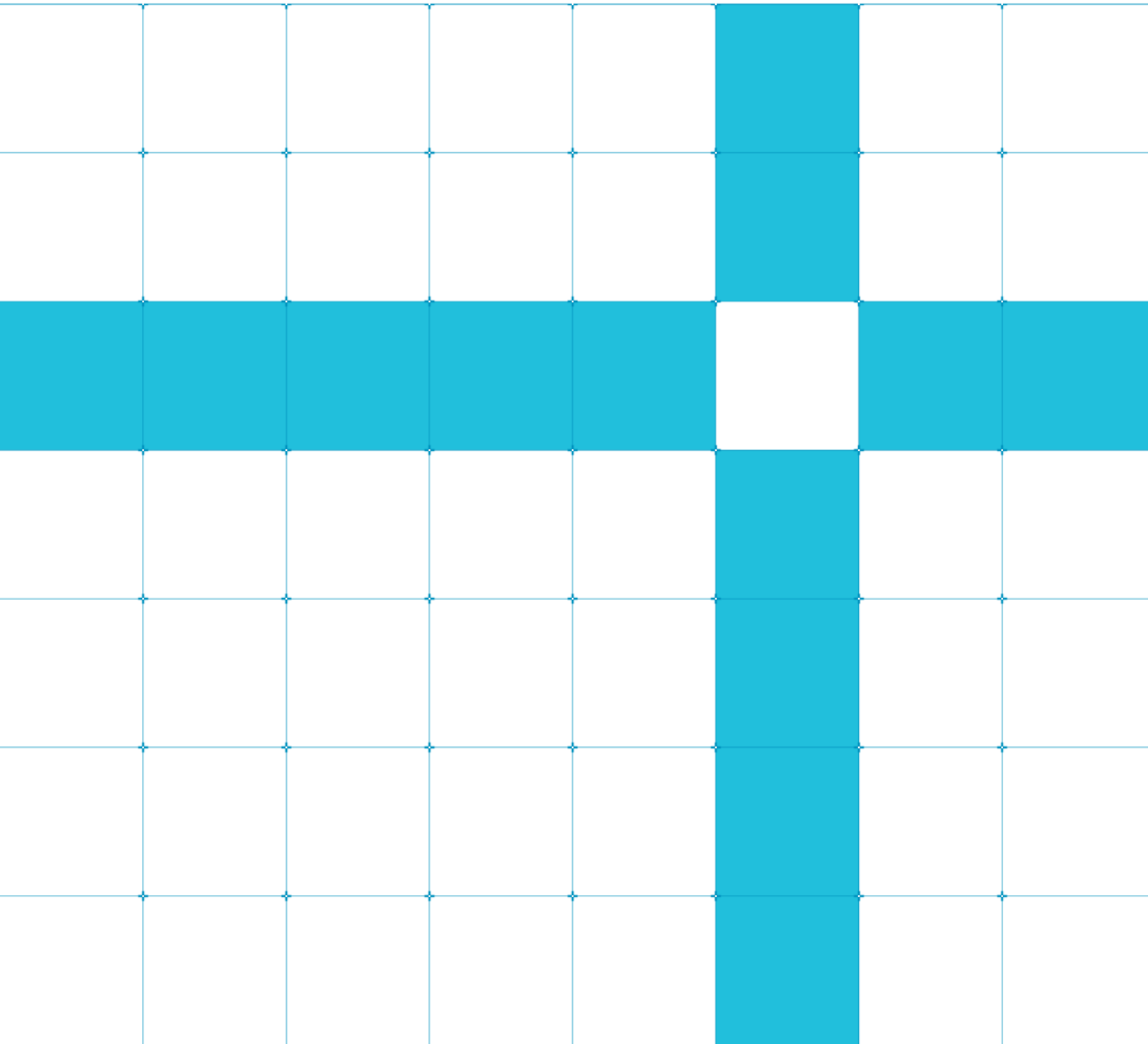




# Arm® SDEI Architecture Compliance Test Scenario

Version 1.1



## Arm® SDEI Architecture Compliance

### Test Scenario

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

### Release Information

#### Document History

Version	Date	Confidentiality	Change
1.0	31 January 2018	Non-Confidential	Betav0.6 release with the following updates: <ul style="list-style-type: none"><li>Updated test case ordering.</li><li>Updated API names.</li><li>Removed infeasible tests.</li></ul>
1.1	16 May 2018	Non-Confidential	No changes for v1.0 release. Note: The document now follows a new template.

### Non-Confidential Proprietary Notice

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

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

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

# Contents

<b>1 About this document .....</b>	<b>6</b>
1.1. References .....	6
1.2. Terms and Abbreviations .....	6
1.3. Scope .....	6
<b>2 Introduction .....</b>	<b>7</b>
<b>3 Cross reference to architecture and tests.....</b>	<b>8</b>
<b>4 Verification scenarios .....</b>	<b>11</b>
4.1. Either have EL2 or EL3 or both.....	11
4.2. GIC v3 is present in the system.....	11
4.3. SDEI Event Register.....	11
4.4. SDEI Interrupt Bind .....	11
4.5. SDEI Event Enable .....	12
4.6. SDEI Event Disable .....	12
4.7. SDEI Event Unregister .....	12
4.8. Client and dispatcher execution state in AArch64 state .....	13
4.9. SDEI Event Routing Set .....	13
4.10. SDEI interrupt release test.....	13
4.11. SDEI Event Status .....	13
4.12. SDEI Event Get Info .....	14
4.13. Priority is indicated by firmware description.....	14
4.14. The priority of the event is configured by the platform and the client cannot change the priority .....	14
4.15. SDEI Private Reset.....	14
4.16. SDEI Shared Reset.....	14
4.17. SDEI PE Mask test .....	15
4.18. SDEI PE Unmask test.....	15
4.19. SDEI Features test.....	15
4.20. Dispatcher reserve at least 2 private event bind slots and 2 shared event bind slots.....	15
4.21. SDEI_VERSION test .....	15
4.22. Error Code (NOT_SUPPORTED).....	16
4.23. Error Code (INVALID_PARAMETERS) .....	16
4.24. Error Code (DENIED) .....	16
4.25. Error Code (PENDING) .....	16
4.26. Error Code (OUT_OF_RESOURCE).....	16

4.27. SDEI Event Complete .....	17
4.28. The Dispatcher must save and later restore the client Execution state .....	17
4.29. Event Handler Context.....	17
4.30. GHES firmware discovery, notify structures and GHES events .....	18
4.31. Event Handler Context denied test.....	18
4.32. SDEI Event Complete and Resume.....	18
4.33. For SDEI_EVENT_COMPLETE_AND_RESUME, execution resumes at the address provided to the call in the resume address parameter. ....	19
4.34. On resumption, the PE registers will contain the interrupted Execution state with some exceptions .....	19
4.35. Client interrupts cannot pre-empt event handler .....	19
4.36. SDEI Events are higher priority than interrupts .....	20
4.37. Check always availability for the following calls – VERSION, EVENT_GET_STATUS, PE_MASK, PE_UNMASK, INTERRUPT_BIND, EVENT_SIGNAL, FEATURES, PRIVATE_RESET, SHARED RESET.....	20
4.38. SDEI Event Signal .....	20
4.39. All SDEI instances must implement the standard event 0 which denotes a software signaled event.....	21
4.40. If multiple events of the same priority are triggered on a PE, the handlers must run in sequence .....	21
4.41. The SDEI events belonging to a class cannot preempt events from the same class.....	21
4.42. Only one instance of a shared event can be handled in a system .....	22
4.43. PSCI_VERSION, AFFINITY_INFO, PSIC_FEATURES are allowed from within SDEI handler.....	22
4.44. Suspend with off - Dispatcher must retain the status of all the events that the PE has registered.....	22
4.45. Suspend with off - Dispatcher must mask the SDEI events for the PE when it wakes up until unmask .....	22
4.46. Changes to the SDEI mask status of the PE, or state of the event only take full effect when the handler completes .....	23
4.47. Following every PE reset, the SDEI events will be masked for the client.....	23
4.48. When PE is off, ensure that no SDEI event must be able to bring back the core online.....	23
4.49. The Dispatcher must be able to save the PE state of all nested, running handlers .....	23
4.50. The handler must not be able to access any residual register state from higher Exception levels .....	24
4.51. Physical events always preempt the virtual events .....	24
4.52. SYSTEM_RESET, SYSTEM_OFF, CPU_OFF and CPU_FREEZE will complete SDEI handler and then perform power operation.....	24

# 1 About this document

This document describes the test scenarios for SDEI compliance.

## 1.1. References

Reference	Document	Author	Title
1	ARM DEN0054A	Arm	<i>Software Delegated Exception Interface</i>
2	Arm DDI 0487	Arm	<i>Arm® Architecture Reference Manual Armv8, for Armv8-A architecture</i>
3	ARM EPM136403 v0.1	Arm	<i>Arm® SDEI Validation Methodology</i>

## 1.2. Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ACS	Arm Architecture Compliance Suite – A group of architecture validation tests for the verification of an architecture feature set.
ELx	Exception Level 'x' where x can be 0,1,2, or 3
PE	Processing Element
PPI	Private Peripheral Interrupt
PSCI	Power State Coordination Interface
SBSA	Server Base System Architecture
SDEI	Software Delegated Exception Interface
SGI	Software Generated Input
SMC	Secure Monitor Call
SPI	Shared Peripheral Interrupt
UEFI	Unified Extensible Firmware Interface

## 1.3. Scope

This document describes the verification scenarios and the strategy that is followed for creating ACS tests for configuration APIs, as described in the SDEI architecture.

## 2 Introduction

*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 specification.

### 3 Cross reference to architecture and tests

The following table shows the cross references from the architecture specification to the corresponding scenarios and test numbers in the SDEI ACS.

Test Number	Specification Section	Test scenario
1	4.1	Either have EL2 or EL3 or both
2	4.2	GIC v3 is present in the system
3	5.1.2	SDEI_EVENT_REGISTER(0xC400_0021)
4	5.1.14	SDEI_INTERRUPT_BIND(0xC400_002D)
5	5.1.3	SDEI_EVENT_ENABLE(0xC400_0022)
6	5.1.4	SDEI_EVENT_DISABLE(0xC400_0023)
7	5.1.8	SDEI_EVENT_UNREGISTER(0xC400_0027)
8	4.1	Client execution state is AArch64
9	5.1.11	SDEI_EVENT_ROUTING_SET(0xC400_002A)
10	5.1.15	SDEI_INTERRUPT_RELEASE(0xC400_002E)
11	5.1.9	SDEI_EVENT_STATUS(0xC400_0028)
12	5.1.10	SDEI_EVENT_GET_INFO(0xC400_0029)
	4.3.2.1	Priority is indicated by firmware description.
	4.3.2.1	The priority of the event is configured by the platform and the client cannot change the priority.
13	5.1.18	SDEI_PRIVATE_RESET(0xC400_0031)
14	5.1.19	SDEI_SHARED_RESET(0xC400_0032)
15	5.1.12	SDEI_PE_MASK(0xC400_002B)
16	5.1.13	SDEI_PE_UNMASK(0xC400_002C)
17	5.1.17	SDEI_FEATURES(0xC400_0030)
	6.3	The dispatcher reserves at least 2 private event bind slots and 2 shared event bind slots.
18	5.1.1	SDEI_VERSION (0xC400_0020)
19	5.3	Error Code – NOT SUPPORTED
20	5.3	Error Code – Invalid Parameters
21	5.3	Error Code – DENIED
22	5.3	Error Code – Pending
23	5.3	Error Code – Out_OF_RESOURCE
24	5.1.6	SDEI_EVENT_COMPLETE(0xC400_0025)



Test Number	Specification Section	Test scenario
	5.2.1.2	The dispatcher must save and later restore the client Execution state.
25	5.1.5	SDEI_EVENT_CONTEXT(0xC400_0024)
	5.2.1	The PC is set to the entry_point_address provided in the SDEI_EVENT_REGISTER call.
	5.2.1	X0 is set to the event number.
	5.2.1	X1 is set to the ep_argument provided in the SDEI_EVENT_REGISTER call.
27	5.1.5	SDEI_EVENT_CONTEXT Denied test
28	5.1.7	SDEI_EVENT_COMPLETE_AND_RESUME(0xC400_0026)
	5.2.2.2	For SDEI_EVENT_COMPLETE_AND_RESUME, execution resumes at the address provided to the call in the resume address parameter.
	5.2.2.2	On resumption, the PE registers will contain the interrupted Execution state with some exceptions.
29	5.2	Client interrupts cannot pre-empt event handler.
30	4.3.2	SDEI events are higher priority than interrupts.
31	6.1.2	Check Always availability – VERSION
32	6.1.2	Check Always availability - EVENT_STATUS
33	5.1.16	SDEI_EVENT_SIGNAL(0xC400_002F)
	4.4	All SDEI instances must implement the standard event 0 which denotes a software signaled event
34	6.1.2	Check Always availability – FEATURES
35	6.1.2	Check Always availability – SHARED_RESET
36	6.1.2	Check Always availability – PE_MASK
37	6.1.2	Check Always availability – PE_UNMASK
38	6.1.2	Check Always availability – INTERRUPT_BIND
39	6.1.2	Check Always availability – PRIVATE_RESET
40	5.2.1.2	If multiple events of the same priority are triggered on a PE, the handlers must run in sequence.
41	4.3.2.1	The SDEI events belonging to a class cannot preempt events from the same class.
42	6.1.2	Check Always availability - EVENT_SIGNAL
43	6.2.1	Only one instance of a shared event can be handled in a system.
44	6.5.4	PSCI_VERSION, AFFINITY_INFO, PSIC_FEATURES are allowed from within SDEI handler.

Test Number	Specification Section	Test scenario
45	6.5.2.2.2	Suspend with off – Dispatcher must retain the status of all the events that the PE has registered.
		Suspend with off – Dispatcher must mask the SDEI events for the PE when it wakes up until unmask.
46	5.2.1.2	Changes to the SDEI mask status of the PE, or state of the event only take full effect when the handler completes.
47	6.5.1	Following every PE reset, the SDEI events will be masked for the client.
48	6.5.2.1	When PE is off, ensure that no SDEI event must be able to bring back the core online.
49	5.2.1.2	The Dispatcher must be able to save the PE state of all nested, running handlers.
50	5.2.1.2	In particular, the handler must not be able to access any residual register state from higher Exception levels.
51	4.3.2.1	Physical events always preempt the virtual events.
52	6.5.4	SYSTEM_RESET, SYSTEM_OFF, CPU_OFF and CPU_FREEZE will complete SDEI handler and then perform power operation.

# 4 Verification scenarios

## 4.1. Either have EL2 or EL3 or both

Test # 1

Steps:

1. Read AA64PFR0\_EL1 register.
2. Test passes if any one of the EL is implemented.
3. Check this for every PE.

## 4.2. GIC v3 is present in the system

Test # 2

Steps:

1. Read GIC version from GIC info table.
2. Test passes if read value is 3.

## 4.3. SDEI Event Register

Test # 3

Steps:

1. Get the event number using `val_event_get` API.
2. Set entry point to event handler.
3. Register event using `val_sdei_event_register`.
4. Test passes if the function returns zero. Otherwise, it should return error number.
5. After test completion unregister the event using `val_sdei_event_unregister` API.
6. Check this for both Private and Shared events.
7. Try to register an invalid event. This must result in an error.

## 4.4. SDEI Interrupt Bind

Test # 4

Steps:

1. Disable the interrupt in GIC Distributor/Redistributor before binding to event using `val_gic_disable_interrupt`.
2. Bind the interrupt to SDEI event using `val_sdei_interrupt_bind` API.
3. Test passes if the function returns zero. Otherwise, it should return error number.
4. Check this for both SPI and PPI.
5. Try to bind the SGI interrupt. This should result in an error.
6. After test completion, release the interrupt using `val_sdei_interrupt_release` API.

## 4.5. SDEI Event Enable

Test # 5

Steps:

1. Get the event number using `val_event_get` API.
2. Register event using `val_sdei_event_register`.
3. Enable the event using `val_sdei_event_enable` API.
4. Test passes if the function returns zero. Otherwise, it should return error number.
5. Check this for both Shared and Private events.
6. After completion of test, unregister the event using `val_sdei_event_unregister` API.
7. Try to enable an invalid event. This should result in an error.

## 4.6. SDEI Event Disable

Test # 6

Steps:

1. Get the event number using `val_event_get` API.
2. Register event using `val_sdei_event_register`.
3. Disable the event using `val_sdei_event_disable` API.
4. Test passes if the function returns zero. Otherwise, it should return error number.
5. Check this for both Shared and Private events.
6. After completion of test unregister the event using `val_sdei_event_unregister` API.
7. Try to disable an invalid event. This should result in an error.

## 4.7. SDEI Event Unregister

Test # 7

Steps:

1. Get the event number using `val_event_get` API.
2. Set entry point to event handler.
3. Register event using `val_sdei_event_register`.
4. Unregister event using `val_sdei_event_unregister`.
5. Test passes if the function returns zero. Otherwise, it should return error number.
6. Check this for both Private and Shared events.
7. Try to unregister an invalid event. This should result in an error.

## 4.8. Client and dispatcher execution state in AArch64 state

Test # 8

Steps:

1. Generate an exception.
2. Verify if the PSTATE register 64-bit is set or not.

## 4.9. SDEI Event Routing Set

Test # 9

Steps:

1. Register the shared event.
2. If successfully registered, call `val_sdei_event_routing_set`.
3. Event routing fails if it returns non-zero value.
4. Unregister the event and return the result.
5. Verify above for RM\_PE, RM\_ANY.
6. Invalid Parameters case -> Invalid event number, Invalid routing mode, Private event.

## 4.10. SDEI interrupt release test

Test # 10

Steps:

1. Disable the interrupt for which we want to verify the call.
2. If successfully disabled, bind the interrupt using `val_sdei_interrupt_bind`.
3. If bind returns success, release the event using `val_sdei_interrupt_release`.
4. Test fails if it returns a non-zero value.
5. Check this for Private & Shared event.
6. Try to release an unbounded event. This should result in an error.

## 4.11. SDEI Event Status

Test # 11

Steps:

1. Get the event number using `val_event_get` API.
2. Check the event status register, enable and handler running using `val_sdei_event_status` API.
3. Test passes if the function returns zero. Otherwise, it should return error number.
4. Try to check if event status is an invalid event. This should result in an error.

## 4.12. SDEI Event Get Info

Test # 12

Steps:

1. Get the event number using `val_event_get` API.
2. Check the event info such as type, priority, routing mode etc using `val_sdei_event_get_info` API.
3. Test passes if the function returns zero. Otherwise, it should return error number.
4. Try to check if event info is an invalid event. This should result in an error.

## 4.13. Priority is indicated by firmware description

Test # 12

Steps:

1. Bind an interrupt to a event using `val_sdei_interrupt_bind` API.
2. Get the priority of the bound event using `val_sdei_event_get_info` API.
3. The priority value returned should be the expected value of Normal priority. Otherwise, the test fails.
4. Get the priority of a hardware error event using `val_sdei_event_get_info` API.
5. The priority value returned should be the expected value of Normal priority. Otherwise, the test fails.

## 4.14. The priority of the event is configured by the platform and the client cannot change the priority

Test # 12

Steps:

1. Bind a shared interrupt to get the shared event number using `val_interrupt_bind` API.
2. Check the priority of the bound event. It should be normal. If not, the test fails.
3. Check the priority of a hardware error event. It should be critical. If not, the test fails.

## 4.15. SDEI Private Reset

Test # 13

Steps:

1. Call private reset API on each PE to reset all private event registrations.
2. Test passes if the function returns zero. Otherwise, it should return error number.

## 4.16. SDEI Shared Reset

Test # 14

Steps:

1. Call `val_sdei_shared_reset` API to reset all event registrations and interrupt-bindings.
2. Test passes if the function returns zero. Otherwise, it should return error number.

## 4.17. SDEI PE Mask test

Test # 15

Steps:

1. Mask the current PE using val API.
2. Test fails if it returns non-zero value. Otherwise, unmask the PE.

## 4.18. SDEI PE Unmask test

Test # 16

Steps:

1. Unmask the PE.
2. Test fails if it returns non-zero value.

## 4.19. SDEI Features test

Test # 17

Steps:

1. Call val\_sdei\_sdei\_features with valid feature. (that is 0 for bind slots)
2. Validate the return value. Bits [63:32] should be 0.
3. Check SDEI\_FEATURES with invalid feature. This should result in an error INVALID PARAMETERS.

## 4.20. Dispatcher reserve at least 2 private event bind slots and 2 shared event bind slots

Test # 17

Steps:

1. Get the features value using val\_sdei\_features API.
  2. Test bits [31:16] for number shared event slots. The value should be atleast 2.
  3. Test bits [15:0] for number private event slots. The value should be atleast 2.
- If number of both private and shared event slots is at least 2, test passes. Otherwise, the test fails.

## 4.21. SDEI\_VERSION test

Test # 18

Steps:

1. Call val\_sdei\_get\_version.
2. Validate return value.
  - a. Major revision (Bits[62:48]) should be 1.
  - b. Minor revision (Bits[47:32]) should be 0.
  - c. Bit[63] should be 0.

## 4.22. Error Code (NOT\_SUPPORTED)

Test # 19

Steps:

1. Check if SDEI table is present in ACPI table.
2. Get SDEI version for val\_sdei\_version API.
3. If the return value is NOT\_SUPPORTED, the test has passed. Otherwise, the test fails.

## 4.23. Error Code (INVALID\_PARAMETERS)

Test # 20

Steps:

1. Register an invalid event using val\_sdei\_event\_register.
2. Test passes if it returns INVALID\_PARAMETERS error code. Otherwise, the test fails.

## 4.24. Error Code (DENIED)

Test # 21

Steps:

1. Register a valid event.
2. Try to re-register the same event using val\_sdei\_event\_register.
3. Test passes if it returns DENIED error code.

## 4.25. Error Code (PENDING)

Test # 22

Steps:

1. Enable the event using val\_sdei\_event\_enable.
2. Trigger the event.
3. Invoke val\_sdei\_event\_unregister for the event inside the handler.
4. Test passes if it returns PENDING error code.

## 4.26. Error Code (OUT\_OF\_RESOURCE)

Test # 23

Steps:

1. Read SDEI\_FEATURES with feature=0.
2. Get num\_slots available for private or shared event.
3. Bind interrupts for all the available slots.
4. Try to bind an interrupt. The test Passes if it returns OUT\_OF\_RESOURCE error code.
5. Release all the interrupts.



## 4.27. SDEI Event Complete

Test # 24

Steps:

1. Get the event number using `val_event_get` API.
2. Register event using `val_sdei_event_register`.
3. Enable the event using `val_sdei_event_enable` API.
4. Generate the event and the dispatcher calls registered client event handler.
5. Call `val_sdei_event_complete` in event handler to set event handler property, handler-running to false.
6. Query on event handler status using `val_sdei_event_status` API.
7. If the test passes, check if the result Bit[2] is zero.
8. Check this for both private and shared events.
9. After completion of the test, unregister the event using `val_sdei_event_unregister` API.

## 4.28. The Dispatcher must save and later restore the client Execution state

Test # 24

Steps:

1. Register event using `val_sdei_event_register` API.
2. Save the execution state at memory address A.
3. Trigger event.
4. Save execution state at memory address B.
5. Compare contents of memory at addresses A and B.
6. If content is same test passed. Otherwise, the test failed.

## 4.29. Event Handler Context

The PC is set to the `entry_point_address` provided in the `SDEI_EVENT_REGISTER` call.

X0 is set to the event number.

X1 is set to the `ep_argument` provided in the `SDEI_EVENT_REGISTER` call.

Test # 25

Steps:

1. Register and enable the event.
2. Trigger the event.
3. In Handler, read PC, X0, X1.
4. The PC is set to the `entry_point_address` provided in the `SDEI_EVENT_REGISTER` call.
5. X0 is set to the event number.
6. X1 is set to the `ep_argument` provided in the `SDEI_EVENT_REGISTER` call.

## 4.30. GHES firmware discovery, notify structures and GHES events

Test # 26

### ACPI Test SDEI table presence

Steps:

1. Parse the ACPI table to check for the presence of SDEI table.
2. If SDEI table is found, the test passes. Otherwise, the test fails.

### ACPI Test HEST table presence

Steps:

1. Parse the ACPI table to check for the presence of HEST table.
2. If HEST table is found, the test passes. Otherwise, the test fails.

### ACPI Test SDEI Notification type GHES structure in HEST table

Steps:

1. Parse the GHES structures in HEST table.
2. Check the NotificationType fields of these structures to find the SDEI type structures.
3. If no SDEI notification type structures are found, the test fails. Otherwise, it passes.

## 4.31. Event Handler Context denied test

Test # 27

Steps:

7. Register and enable the event.
8. Trigger the event.
9. In Handler, read PC, X0, X1.
10. The PC is set to the entry\_point\_address provided in the SDEI\_EVENT\_REGISTER call.
11. X0 is set to the event number.
12. X1 is set to the ep\_argument provided in the SDEI\_EVENT\_REGISTER call.

## 4.32. SDEI Event Complete and Resume

Test # 28

Steps:

1. Get the event number using val\_event\_get API.
2. Register event using val\_sdei\_event\_register.
3. Enable the event using val\_sdei\_event\_enable API.
4. Generate the event and dispatcher calls registered client event handler.
5. Call val\_sdei\_event\_complete\_and\_resume in event handler to set event handler property, handler-running to false. After that, it resumes the execution at resume\_addr from ELc.
3. Query on event handler status using val\_sdei\_event\_status API.
4. If test passes, check if the result Bit[2] is zero.
5. Check this for both private and shared events.

6. After completion of test, unregister the event using `val_sdei_event_unregister` API.

### 4.33. For `SDEI_EVENT_COMPLETE_AND_RESUME`, execution resumes at the address provided to the call in the resume address parameter.

Test # 28

Steps:

1. Register event handler using `val_sdei_event_register` API.
2. Enable event handler using `val_sdei_event_enable` API.
3. Trigger event.
4. In the event handler, complete and resume the handler using `val_sdei_event_complete_and_register` API at known address to a test function.
5. In the test function, set the test result as passed. If test function is not called, test fails.

### 4.34. On resumption, the PE registers will contain the interrupted Execution state with some exceptions

Test # 28

Steps:

1. Register event handler using `val_sdei_event_register` API.
2. Enable event handler using `val_sdei_event_enable` API.
3. Save execution state except at memory address X.
4. Trigger event.
5. In event handler, complete the event and resume on address of a test function using `val_sdei_event_complete_and_resume` API.
6. In the test function, save execution state at memory address Y.
7. In test function compare the contents memory address X and Y. If contents match, the test has passed. Otherwise, the test has failed.

### 4.35. Client interrupts cannot pre-empt event handler

Test # 29

Steps:

1. Register interrupt and interrupt handler using `val_gic_install_isr` API to Guest OS.
2. Register event number using `val_sdei_event_register` API.
3. Enable the event number using `val_sdei_event_enable`.
4. Generate the event and dispatcher calls registered client event handler.
5. Generate the interrupt in event handler and poll for shared memory status in event handler.
6. If test passes, timeout occurs. Otherwise, shared memory status is set by interrupt handler.
7. After test completion, unregister the interrupt and event.

## 4.36. SDEI Events are higher priority than interrupts

Test # 30

Steps:

1. Register interrupt and interrupt handler using `val_gic_install_isr` API to Guest OS.
2. Register event number using `val_sdei_event_register` API.
3. Enable the event number using `val_sdei_event_enable`.
4. Generate the interrupt and Guest OS calls the interrupt handler.
5. Generate event in interrupt handler and poll for shared memory status in interrupt handler.
6. Dispatcher calls event handler and set shared memory status in event handler.
7. If test passes, check the shared memory status set by event handler. Otherwise, timeout occurs.
8. After test completion, unregister the interrupt and event.

## 4.37. Check always availability for the following calls – VERSION, EVENT\_GET\_STATUS, PE\_MASK, PE\_UNMASK, INTERRUPT\_BIND, EVENT\_SIGNAL, FEATURES, PRIVATE\_RESET, SHARED RESET.

Test # 31, 32, 34, 35, 36, 37, 38, 39, 42

Steps:

1. Register an event using `val_sdei_event_register` API.
2. Enable event using `val_sdei_event_enable` API.
3. Trigger event.
4. In the handler, make the following SDEI calls using respective APIs.
  - a. `VERSION` (`val_sdei_version`)
  - b. `EVENT_GET_STATUS` (`val_sdei_event_get_status`)
  - c. `PE_MASK` (`val_sdei_pe_mask`)
  - d. `PE_UNMASK`, (`val_sdei_pe_unmask`)
  - e. `INTERRUPT_BIND` (`val_sdei_interrupt_bind`)
  - f. `EVENT_SIGNAL` (`val_sdei_event_signal`)
  - g. `FEATURES` (`val_sdei_features`)
  - h. `PRIVATE_RESET` (`val_sdei_private_reset`)
  - i. `SHARED_RESET` (`val_sdei_shared_reset`)
5. If any of the above calls fail, the test failed. Otherwise, the test passed.

## 4.38. SDEI Event Signal

Test # 27

Steps:

1. Register the event zero on specified target PE using `val_sdei_event_register` API.
2. Enable the event on specified target PE using `val_sdei_event_enable` API.
3. Call `val_sdei_event_signal` api to signal a software event to client PE.
4. Test passes if the function returns zero. Otherwise, it should return error number.

5. After completion of test, unregister the event zero using `val_sdei_event_unregister` API.
6. Try to signal an invalid event number or target PE. This should result in an error.

#### 4.39. All SDEI instances must implement the standard event 0 which denotes a software signaled event

Test # 27

Steps:

1. Register the event zero on specified target PE using `val_sdei_event_register` API.
2. Enable the event on specified target PE using `val_sdei_event_enable` API.
3. Query on event zero using `val_sdei_event_get_info` API.
4. If test passes check if the result bit[1] is zero.
5. After completion of test, unregister the event zero using `val_sdei_event_unregister` API.

#### 4.40. If multiple events of the same priority are triggered on a PE, the handlers must run in sequence

Test # 40

Steps:

1. Register bound event A with normal priority using `val_sdei_register_event` API.
2. Register bound event B with normal priority using `val_sdei_register_event` API.
3. Initialize a reference counter to 0.
4. Trigger event A.
5. In event A handler, increment a reference counter.
6. In event A handler, trigger event B.
7. In event A handler, check reference counter value after a delay. If the value is  $> 1$ , the test failed since it means event B handler was called while event A handler was running.
8. In event B handler, increment reference counter and complete the event.

#### 4.41. The SDEI events belonging to a class cannot preempt events from the same class

Test # 41

Steps:

1. Register a normal priority event (bound event) using `val_sdei_register_event` API.
2. Generate the event.
3. In the handler, increment a reference counter and clear the event.
4. Generate the event again, in the handler.
5. If another instance of handler is called, the reference counter will get incremented.
6. Check the value of reference counter. If its more than 1, test fails. If it is 1, the test passed.

## 4.42. Only one instance of a shared event can be handled in a system

Test # 43

Steps:

1. Client registers SPI bound event with affinity routing mode RM\_ANY using val\_sdei\_event\_register API.
2. Client triggers event by asserting the SPI, expecting handler to be called using val\_get\_gicd\_base API.
3. Handler clears the SPI status registers and signals end of interrupt using val\_gic\_end\_of\_interrupt API.
4. Handler increments reference counter to 1.
5. Handler triggers event by asserting the SPI val\_get\_gicd\_base.
6. Handler checks if reference counter was incremented further. If so, it is inferred that another instance of the handler was called, test status is set as passed. Otherwise, test status is set as failed.
7. Handler completes event handling via val\_sdei\_event\_complete API.

## 4.43. PSCI\_VERSION, AFFINITY\_INFO, PSIC\_FEATURES are allowed from within SDEI handler

Test # 44

Steps:

1. Register a shared event.
2. Trigger the event.
3. In the handler make PSCI calls for interfaces PSCI\_VERSION, AFFINITY\_INFO and PSCI\_FEATURES. If the calls pass, test passes Otherwise, it fails.

## 4.44. Suspend with off - Dispatcher must retain the status of all the events that the PE has registered

Test # 45

Steps:

1. Register a shared event targeting PE n and private event on PE n.
2. Suspend PE n via PSCI\_CPU\_SUSPEND call.
3. Trigger an event to wake up PE n.
4. Check event info for the events registered in step 1. The info should be the same as it was just after the events were registered. If so, the test passed. Otherwise, it failed.

## 4.45. Suspend with off - Dispatcher must mask the SDEI events for the PE when it wakes up until unmask

Test # 45

Steps:

1. Suspend PE n via PSCI\_CPU\_SUSPEND call.
2. Trigger an event to wake up PE n.
3. Check sdei mask status for the PE. If it is masked, then the test passed. Otherwise, the test failed.

## 4.46. Changes to the SDEI mask status of the PE, or state of the event only take full effect when the handler completes

Test # 46

Steps:

1. Register event handler using `val_sdei_event_register` API.
2. Enable event handler using `val_sdei_event_enable` API.
3. Trigger event.
4. In event handler, disable event using `val_sdei_event_disable` API.
5. In event handler, unregister event using `val_sdei_event_unregister` API. This should return with code `PENDING`. If not so, the test failed.
6. In event handler, check event status using `val_sdei_event_get_status` API. The event status should still be registered and running. If not, the test failed.
7. Complete event handler.
8. Check event status using `val_sdei_event_get_status` API. The event status should be unregistered. If not test failed. Otherwise, the test passed.

## 4.47. Following every PE reset, the SDEI events will be masked for the client

Test # 47

Steps:

1. Turn PE off via PSCI interface `PSCI_CPU_OFF`.
2. Turn PE on via PSCI interface `PSCI_CPU_ON`.
3. Check if SDEI is masked on the PE using `val_sdei_event_mask` call. If masked, test passes. Otherwise, it fails.

## 4.48. When PE is off, ensure that no SDEI event must be able to bring back the core online

Test # 48

Steps:

1. Turn the target PE off via `PSCI_CPU_OFF`.
2. Register an event targeting the PE turned off in step 1, using `val_register_event`.
3. If the handler is called PE, the test failed. Otherwise, the test passed.

## 4.49. The Dispatcher must be able to save the PE state of all nested, running handlers

Test # 49

Steps:

1. Register normal priority event A using `val_sdei_event_register` API.
2. Register critical priority event B using `val_sdei_event_register` API.
3. Save execution state at memory address X.
4. Trigger event A.
5. In event A handler, save execution state at memory address Y.
6. In event A handler, trigger event B.

7. In event A handler, after event B has been handled, save execution state at memory address Z.
8. In event A handler, compare contents of memory at addresses Y and Z. If content is same test passed, else failed.
9. After event A has been handled, save execution state at memory address Y.
10. Compare contents of memory at addresses Y and Z. If content is same, the test passed. Otherwise, the test failed.

## 4.50. The handler must not be able to access any residual register state from higher Exception levels

Test # 50

Steps:

1. Register event using `val_sdei_event_register` API.
2. Save execution state at memory address X.
3. Trigger event.
4. In event handler, save execution state at memory address Y.
5. Compare contents of memory at addresses X and Y. If content is same test passed. Otherwise, it failed.
6. Complete event handler.

## 4.51. Physical events always preempt the virtual events

Test # 51

Steps:

1. Register and enable physical event in hypervisor.
2. Register and enable virtual event in guest OS.
3. Trigger virtual event.
4. Inside virtual event handle, trigger physical event.
5. If physical event handler is called, the test passed. Otherwise, the test failed.

## 4.52. `SYSTEM_RESET`, `SYSTEM_OFF`, `CPU_OFF` and `CPU_FREEZE` will complete SDEI handler and then perform power operation

Test # 52

Steps:

1. Register shared event targeted to PE n via `val_sdei_event_register` API.
2. Trigger the shared event, so that handler is called at PE n.
3. Inside the handler make the `PSCI_CPU_OFF` call to power off PE n.
4. From the main PE, check the status of the event using `val_sdei_get_event_info` call. If the event status is still running, then the test failed. Otherwise, the test passed.