Software Delegated Exception Interface (SDEI)

Platform Design Document

Document number:	ARM DEN 0054C
Release Quality:	BETA
Issue Number:	0
Confidentiality:	Non-Confidential

Copyright © 2017-2021 Arm or its affiliates. All rights reserved.
Document number: ARM DEN 0054C

Software Delegated Exception Interface
System Software on Arm specification

Copyright © 2017-2021 Arm Limited or its affiliates. All rights reserved.

Release information

The Change history table lists the changes made to this document.

Table 1 Change history

Date	Issue	Confidentiality	Change
8 May 2017	A	Non-confidential	First release
6 October 2020	B	Non-confidential	Cleanup and clarifications, license update
14 September 2021	C	Non-confidential	Minor version rev to 1.1 Defined new relative mode for SDEI_EVENT_REGISTER. Defined _DSM method in Appendix D for representing SDEI events in ACPI. Clarifications in Appendix C (ACPI table definitions for SDEI) for how CPER data should be handled. Misc errata clean up.

Arm Non-Confidential Document Licence ("Licence")

This Licence is a legal agreement between you and Arm Limited ("Arm") for the use of Arm's intellectual property (including, without limitation, any copyright) embodied in the document accompanying this Licence ("Document"). Arm licenses its intellectual property in the Document to you on condition that you agree to the terms of this Licence. By using or copying the Document you indicate that you agree to be bound by the terms of this Licence.

"Subsidiary" means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Document is **NON-CONFIDENTIAL** and any use by you and your Subsidiaries ("Licensee") is subject to the terms of this Licence between you and Arm.

Subject to the terms and conditions of this Licence, Arm hereby grants to Licensee under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide licence to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the licence granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the licence granted in (i) above.

Licensee hereby agrees that the licences granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

THE 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. Arm may make changes to the Document at any time and without notice. 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, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PETMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE'S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENCE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This Licence shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to Licensee. Licensee may terminate this Licence at any time. Upon termination of this Licence by Licensee or by Arm, Licensee shall stop using the Document and destroy all copies of the Document in its possession. Upon termination of this Licence, all terms shall survive except for the licence grants.

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence 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. No licence, express, implied or otherwise, is granted to Licensee under this Licence, to use the Arm trade marks in connection with the Document or any products based thereon. Visit Arm's website at https://www.arm.com/company/policies/trademarks for more information about Arm's trademarks.

The validity, construction and performance of this Licence shall be governed by English Law.

Copyright © 2017, 2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

Table of Contents

1 Introduction ... 6

1.1 Additional reading ... 6
1.2 Feedback ... 6
1.3 Glossary ... 7
1.4 Document structure ... 8

2 Overview ... 9

2.1 SDEI intended usage ... 9

3 Definitions ... 11

3.1 Software Delegated Exception Model ... 11
3.2 Client and Dispatcher ... 11
3.3 Event ... 11
3.4 Interface and Exception levels ... 12

4 System overview ... 15

4.1 Processing Element (PE) ... 15
4.2 Interrupt controller ... 15
4.3 Prioritizing events ... 15
4.4 Event number allocation ... 17

5 Interface ... 19

5.1 SDEI calls ... 19
5.2 Event context ... 40
5.3 Return Codes ... 43

6 Programmers' Overview ... 44

6.1 Event handler states and properties ... 44
6.2 Event dispatching ... 48
6.3 Bound events ... 49
6.4 Interface Discovery ... 49
6.5 Power management and SDEI events ... 49
6.6 Registering and handling an event ... 53
6.7 Unregistering an event ... 53
6.8 Virtual SDEI events ... 54

7 Appendix A: Implementing use cases ... 56

7.1 Physical interrupt as SDEI event ... 56
7.2 Isolated physical interrupt as SDEI event ... 56

8 Appendix B: Implementation notes with GICv2 and GICv3 architecture 57

8.1 GICv2 ... 57
8.2 GICv3 ... 57

9 Appendix C: Pseudocode for dispatcher ... 58

9.1 Private event dispatcher ... 58
9.2 Shared event dispatcher ... 58

10 Appendix D: ACPI table definitions for SDEI ... 60

11 Appendix E: ACPI definitions for SDEI events ... 62

11.1 Describing SDEI events in ACPI Namespace ... 62
11.2 SDEI _DSM method ... 62
11.3 Example ... 63

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

1 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, which must be serviced immediately by an OS or hypervisor. These events are often orthogonal to normal OS operation and the events 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 document defines a standard interface through which these events can be exposed by the firmware and handled by OS and hypervisors. The SDEI is not suitable for device interrupt handling, which is best handled by the OS itself.

1.1 Additional reading

This section lists relevant publications from Arm and third parties.

See the Arm Infocenter, `http://infocenter.arm.com`, for access to Arm documentation.

1.1.1 Arm publications

The following documents contain information relevant to this document:

	Document name	Document number
1.	Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile	Arm DDI 0487
2.	Critical Interrupt Prioritization	Arm PRDC 013242
3.	SMC Calling Conventions	Arm DEN 0028B
4.	Power State Coordination Interface	Arm DEN 0022C
5.	Arm Generic Interrupt Controller Architecture Specification version 3.0	Arm IHI 0069C
6.	Arm Generic Interrupt Controller Architecture Specification version 2.0	Arm IHI 0048B
7.	Advanced Configuration and Power Interface Specification v6.2	

1.2 Feedback

Arm welcomes feedback on its documentation.

1.2.1 Feedback on this manual

If you have comments on the content of this manual, send an e-mail to `errata@arm.com`. Provide:

- The title.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- The number, Arm DEN 0054C.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1.3 Glossary

This document uses the following terms and abbreviations.

Term	Description
AArch64 state	The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.
ACPI	The Advanced Configuration and Power Interface specification. This defines a standard for device configuration and power management by an OS.
Client	The software entity that uses SDEI. This includes operating system and hypervisor.
Dispatcher	The software entity, which dispatches events to the client. This includes hypervisor or firmware.
EL0	The lowest Exception level. This Exception level is unprivileged. The Exception level used to execute user applications, in Non-secure state.
EL1	Privileged Exception level. The Exception level typically used to execute operating systems.
EL2	Hypervisor Exception level. The Exception level used to execute hypervisor code. EL2 is always in Non-secure state.
EL3	Secure monitor Exception level. This Exception level has the highest privilege and is always in Secure state. If implemented, a PE always reset and commence execution at this Exception level.
FDT	Flattened Device Tree. This is a hardware description methodology. Firmware tables are constructed that describe the hardware. These tables are passed to the OS at boot time. An OS can interrogate the data they contain when it needs to discover the hardware properties of a device.
Firmware	See Secure Platform Firmware.
Function Identifier	A 32-bit integer, which identifies the function being invoked by this SMC/HVC call. Passed in X0 into every SMC/HVC call.
HVC	Hypervisor Call. An instruction that causes a synchronous exception that is taken to EL2.
Hypervisor	The hypervisor executes at EL2. It supports the execution of multiple EL1 operating systems. In this document, the term hypervisor includes any software that is running at EL2. EL2 software could include operating systems if the PE implements the Virtualization Host Extensions.
Non-secure state	The Security state that restricts access to only the Non-secure system resources such as memory, peripherals, and System registers.
Normal world	The execution environment when the core is in the Non-secure state.
OS	Application operating system such as Linux or Windows. This also includes virtualized OS running under a hypervisor.
PE	The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A processing element implementation that is compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

RAS	Reliability, Availability, and Serviceability
Secure Platform Firmware (SPF)	Owned by the silicon vendor and OEM. This firmware is the first software component that executes at boot on an application PE. It provides a number of services, including platform initialization, the installation of the S-EL1 software, and routing of Secure Monitor Calls.
Secure EL1	The Secure EL1 Exception level, the Exception level used to execute the S-EL1 software in Secure state. The software can be a Secure OS or S-EL1 firmware.
Secure state	The Arm Security state that enables access to the Secure and Non-secure systems resources, such as memory, peripherals and System registers.
SMC	Secure Monitor Call. An instruction that causes a synchronous exception that is taken to EL3.
SoC	System on Chip.
System event	An event typically generated by the system firmware which requires immediate attention from the OS or hypervisor.
Virtualization Host Extensions	Virtualized Host Extensions (VHE) is an extension to the Arm Architecture that enables operating systems to run at the EL2 privilege level. See Arm *Architecture Reference Manual, Armv8 for Armv8-A architecture profile*.
VHE	Virtualization Host Extensions

1.4 Document structure

This document is organized as follows,

- Section 2 provides an overview of SDEI and lists typical use cases for SDEI.
- Section 3 provides common definitions that are used in this document.
- Section 4 provides system requirements for implementing SDEI.
- Section 5 provides the interface functions and details about the interface.
- Section 6 provides Programmers' overview on SDEI.
- Appendix A provides details about implementing use cases mentioned in Section 2.
- Appendix B provides implementation details with GICv2 and GICv3.
- Appendix C lists pseudocode for SDEI event dispatcher.
- Appendix D provides ACPI definitions for SDEI.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

2 Overview

The Software Delegated Exception (SDE) is a mechanism to deliver extraordinary System events (also called events in this document) to an OS or hypervisor that preempt all other exceptions and exclusion mechanisms. This document defines the Software Delegated Exception Interface (SDEI) which can be used by OS and hypervisor to subscribe to and manage high priority events.

SDEI should be used instead of a normal interrupt when the exception must be delivered and must not be delayed by interrupt masking or critical sections. Platform error handling and software watchdogs are examples that fall into this category.

The SDEI handler execution environment is limited, as at the time of the event the state and consistency of the underlying OS kernel or hypervisor is unknown. All resources must be pre-allocated, and interactions with the OS or hypervisor must use methods that are guaranteed to be safe.

2.1 SDEI intended usage

SDEI provides a high priority event delivery mechanism, which has higher priority than interrupts that target OSs and hypervisors. SDEI enables a calling hypervisor or OS or both to:

1. Subscribe to and handle a system event.
2. Mask a system event.
3. Migrate handling of a system event to a different PE.
4. Add or remove a PE from participating in event handling.
5. Convert an existing interrupt into a source of SDEI events.
6. Generate software events.

With SDEI and a description of the events from the platform, OS or hypervisor software is able to handle system events. The SDEI is designed to work alongside the *Power State Coordination Interface*, which handles power management operations.

SDEI events can be masked and the interface provides functions for masking events. However, Arm recommends that the events are masked only in rare situations.

SDEI handler execution environment is limited as at the time of the event the underlying state of the OS or hypervisor is unknown. Therefore, SDEI is not suitable for handling normal device interrupts, which are better handled using the standard services provided by the OS or hypervisor.

2.1.1 Typical use cases

The following sections lists use cases where prioritizing the system event is beneficial.

- ***System Error handling (RAS)***

  At any time of execution, the PE, memory, or system buses can generate errors. Some of these errors can be corrected in software and might require software handling from different execution privileges. Firmware first handling is a common approach to error handling, where a higher Exception level provides an initial error handling, after which the error is delegated to a lower Exception level. For critical errors, this delegation will not work if errors occur in a critical section where interrupts are masked. SDEI provides a solution for this problem.

- ***Software Watchdog timer***

  A high priority event can be used to implement a software watchdog timer. When the watchdog timer ticks, the event handler can examine the system for any activity. The system can be reset if there was no activity detected since the previous tick. With prioritizing the timer event, the handler can execute even if the system is busy handling an interrupt.

- ***Kernel Debugging***

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Debugging system software usually involves examining the execution path, registers, and memory. Software debugging is often impaired by the interrupt masking because it might prevent the debugger from interrupting the PE. With a prioritized event, the state of the system can be examined even if the system is within its critical section.

- ***Sample Profiling***

Sample-based profiling can have blind spots for those critical sections that have interrupts masked. A high priority event-based profiler can eliminate such blind spots.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

3 Definitions

This section outlines the various definitions that are followed in this document.

3.1 Software Delegated Exception Model

There are different methods for prioritizing system events and a detailed explanation can be found in the *Critical Interrupt Prioritization* document. SDEI described in this document uses the mechanism in which the system events are trapped to a higher Exception level. Trapping to higher Exception level always preempts the lower Exception level execution, prioritizing the events that are trapped. We call this mechanism a Software Delegated Exception.

3.2 Client and Dispatcher

Software Delegated Exception is a software agreement between higher and lower Exception levels for delegating events from the higher Exception level to the lower Exception level.

The higher Exception level software is called the dispatcher. The dispatcher handles the request from lower Exception level and delegates the event.

The lower Exception level software is called the client. The client uses the interface provided by the dispatcher and handles the events.

3.2.1 Client and Dispatcher Exception level

The client Exception level, EL_C, is the Exception level that the client is executing in. EL_C can be Non-secure EL1 or EL2.

The dispatcher Exception level, EL_D, is the Exception level that the dispatcher executes in. EL_D can be EL2 or EL3.

For a firmware dispatcher (EL_D = EL3), the client must be a hypervisor (EL_C = EL2). If a hypervisor is not present or not enabled, the client must be an OS (EL_C = Non-secure EL1). In this document, hypervisor refers to any software running at EL2. If a PE implements VHE an operating system might run at EL2.

For a hypervisor dispatcher (EL_D = EL2), the client must be a guest OS (EL_C = Non-secure EL1).

See *Interface and Exception levels* on page 12 for various SDEI instances that exist between different Exception levels.

3.3 Event

An event is any notification that the dispatcher wants to inform the client about. The dispatcher passes the event numbers to the client using an IMPLEMENTATION DEFINED mechanism. OS and hypervisor can then, subscribe, and handle these events.

3.3.1 Event source

Events are typically generated as a result of hardware exceptions or hardware interrupts. Hardware events might be notified to the firmware to do the first level of handling. As part of the handling, the firmware might require lower Exception levels to handle the event. Firmware can expose SDEI events for the lower exception handling.

In a related use case, a hardware interrupt can generate an event. The firmware can expose the interrupt as a SDEI event for client handling.

Events can also be generated by software using the system interrupt controller.

3.3.2 Event type

Every SDEI event is either a Private event or a Shared event.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A Private event is local to the PE that generated the event and can only be handled on that PE. Private event is analogous to a private peripheral interrupt (PPI).

A Shared event is a global event, which can be handled by a single PE among a set of target PEs. The target PEs are selected using the routing mode. Shared event is analogous to shared peripheral interrupt (SPI).

The event type can be queried using an SDEI call, see *SDEI_EVENT_GET_INFO* on page 29. The routing mode of a shared event is decided by the client and must be configured during registration of an event handler, see *SDEI_EVENT_REGISTER* on page 20.

3.3.3 Event definition

An event can be defined,

- statically by the platform called *platform events* or
- dynamically by the client, called *bound events*.

Platform events are defined by the platform. The platform events can further be divided to standard or vendor events, see *Event number allocation* on page 17

Bound events are SDEI events created for client interrupts. The bound events can be created and released by the client during its execution. For more information, see *Bound events* on page 49.

3.4 Interface and Exception levels

A system can provide multiple SDEI instantiations depending on the Exception levels that are implemented in the PE and the software executing in those Exception levels. This section describes the various SDEI instances that are permitted in a system and the method of invoking the interface that should be used by the client. The discussion assumes the typical Exception level usage model as described in the *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile.*

3.4.1 SDEI instances in a system

The following SDEI instances are permitted in a system:

1. **Physical SDEI: Firmware dispatcher and Non-secure client**

   Firmware at EL3 provides SDEI to a hypervisor or, if a hypervisor is not present or not enabled, a Non-secure OS.

   A guest OS that is executing under a hypervisor is not permitted to register an SDEI handler, unless the hypervisor offers support for SDEI. Any attempt by a guest OS to do this would result in an Unknown SMC Function Identifier error, as defined in the SMC calling convention specification. This error maps to the NOT_SUPPORTED error code in the SDEI specification.

2. **Virtual SDEI: Hypervisor dispatcher and guest OS**

   Hypervisor at EL2 provides SDEI to a guest OS. The guest OS registers with the hypervisor for virtual events. Virtual events originate from either the firmware or the hypervisor

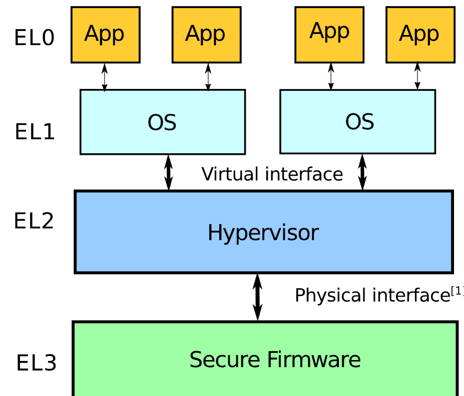
   If the events are originating from firmware, the hypervisor registers with the firmware and the guest OS registers with the hypervisor. Depending on the SDEI implementation present in the hypervisor, it can provide services to single or multiple guest OSs. When an event triggers, the firmware delegates the event to the hypervisor. The dispatcher at hypervisor may further delegate the event to one or more appropriate guest OSs.

This specification does not define an interface between firmware at EL3 and software at Secure EL2 or Secure EL1.It is IMPLEMENTATION DEFINED whether SDEI exists between EL3 and Secure EL2/EL1.

There is no SDEI between EL1 and EL0.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 1 SDEI instances



[1] In a system without hypervisor, there is no virtual interface and physical interface is between firmware and OS.

3.4.2 Conduits

Interface calls from the client to the dispatcher are made using SMC or HVC instructions. The channel that is used for communication between the client and dispatcher is termed Conduit as defined in the *SMC Calling Conventions* specification. This section describes possible PE implementations and the conduits that are used.

When both of EL3 and EL2 are implemented:

The following SDEI instances are permitted:

- Physical SDEI.
- Virtual SDEI.

A client for a physical SDEI is a hypervisor at EL2 or, if the hypervisor is not enabled, an OS at EL1. They must use an SMC Conduit.

For a hypervisor, the SDEI calls must always come through EL2.

Implementation note: For a Type-2 (OS hosted) hypervisor that runs on PEs that do not implement VHE, a hypervisor stub may run at EL2 with the host OS running at EL1. In this case, the firmware dispatcher cannot distinguish between an SDEI request from the host at EL1 or a guest at EL1. Therefore, the hypervisor stub at EL2 must always trap SDEI calls from EL1. One implementation approach is for the hypervisor stub at EL2 to forward the SDEI calls from the host OS to the firmware dispatcher by invoking identical SDEI calls with identical parameters. Because the hypervisor stub made SDEI calls to register the event handlers, the firmware dispatcher will execute the event handlers at EL2. This means that the address of event handlers must be valid in the EL2 translation regime, even though the addresses originated from EL1.

Clients for virtual SDEI are guest OSes at EL1. For virtual SDEI, a guest OS can use an SMC or HVC Conduit for invoking SDEI calls. The guest OS executing under a hypervisor is not permitted to directly register with firmware and the hypervisor must always trap the guest requests.

When EL3 is not implemented and EL2 is implemented:

The following SDEI instance is permitted:

1. Virtual SDEI.

The Client for a virtual SDEI is an OS at Non-secure EL1. They must use the HVC conduit.

When EL3 is implemented and EL2 is not implemented:

The following SDEI instances are permitted:

1. Physical SDEI.

The Client for a physical SDEI is an OS at Non-secure EL1. They must use the SMC conduit.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The following table summarizes the SDEI instances available in a system. The SDEI specification permits any combination of these interfaces in a system unless specified otherwise.

Table 2: SDEI instances and Conduits

EL3 present	EL2 present and enabled	SDEI instance	Client	EL_C	Dispatcher	EL_D	Conduit
Yes	No	Physical	OS	EL1	Firmware	EL3	SMC
Yes	Yes	Physical	Hypervisor	EL2	Firmware	EL3	SMC
-	Yes	Virtual	OS	EL1	Hypervisor	EL2	SMC[1,2] / HVC

[1] SMC must be trapped by hypervisor.
[2] SMC is available only when EL3 is implemented.

 Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

4 System overview

This section describes the system requirements and system configuration that is required to support SDEI.

4.1 Processing Element (PE)

SDEI is for implementation in SoCs that are based on the Armv8-A architecture. To implement SDEI, the PE is required to have EL2, EL3, or both. This means that SDEI requires the PE to include the Virtualization Extension, the Security Extensions, or both. Both the client and dispatcher of SDEI must execute in AArch64 state.

4.2 Interrupt controller

Arm recommends that:

- An interrupt controller is present in the system.
- The SDEI instance utilizes the interrupt controller wherever possible to reduce software overhead and provide timely handling of events.

Arm recommends the Generic Interrupt Controller (GIC). See *Arm Generic Interrupt Controller Architecture Specification version 3.0*.

If the system has a non-GIC interrupt controller, the following features are recommended for SDEI implementation:

- Interrupt routing modes to support event routing.
- Private and shared interrupts to support private and shared events.
- Ability for software to set an interrupt pending for event signaling and delegation.
- Interrupt groups to trap interrupts at the EL_D.
- Interrupt priority to raise the priority of events over client interrupts.
- Ability to generate interrupts in software to support software signaling of events.

4.3 Prioritizing events

SDEI events of interest are trapped to a higher Exception level, preempting any lower Exception level execution and prioritizing the events that are trapped. The selective trapping of events is achieved with the help of both PE architecture controls and the interrupt controller.

4.3.1 Arm PE architecture

The *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile* defines the levels of execution privileges termed Exception levels. To control when interrupts can be taken, the Armv8-A architecture provides:

- Process state (PSTATE) interrupt masks. The PSTATE interrupt masks prevent interrupts from being taken to the current Exception level.
- Interrupt routing controls. Interrupts can be trapped to a higher Exception level than the current Exception level.

Higher priority asynchronous events can be achieved by associating the events to an appropriate interrupt group and targeting the interrupt group to EL_D. For more details, refer *Appendix B: Implementation notes with GICv2 and GICv3 architecture* on page 57. This preempts the execution of EL_C or lower exceptions even when the client has its interrupts masked in its PSTATE. The exception handler at EL_D can triage the interrupt and delegate the interrupt as necessary to the client. The dispatcher can simulate an exception-like entry into the client, with the client providing an additional asynchronous entry point similar to an interrupt entry point. Therefore this architecture is called the Software Delegated Exception Model.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.2 Interrupt controller

The interrupt controller is configured so that the SDEI events are given higher priority to preempt the execution of lower Exception level. This is essential to raise a physical interrupt and there by interrupt the PE. The following section details how the interrupt priority must be managed for different Clients. See *Appendix B: Implementation notes with GICv2 and GICv3 architecture* on page 57 for implementation notes with GIC.

4.3.2.1 Event class

To support the SDEI instances, a system can group the events to the respective SDEI classes as follows:

Physical SDEI

The events in this class must be given priority higher than the client's interrupt sources, so that it can preempt an active Non-secure interrupt.

Virtual SDEI

The virtual events in this class must preempt the virtual interrupts.

For both Physical SDEI and Virtual SDEI, there are two levels of priority:

1. **Normal priority class**, for non-critical system events. A typical example is a software watchdog timer.
2. **Critical priority class**, for critical system events. A typical example is error handling.

Critical priority class events must preempt all normal priority class events, including those that are currently being handled. The event handler for critical priority class events always executes to completion before any other code executes in the client context for the interrupted PE.

Events belonging to a class cannot preempt events from the same class.

For platform events, the priority of the event is configured by the platform and the client cannot change the priority. All bound events are of normal priority.

The client can query the priority of all available events using SDEI call, see *SDEI_EVENT_GET_INFO* on page 29.

The priorities of various event classes are summarized as follows. The convention used here is as follows,

- A > B denotes that A can preempt B.

The preemption rules are:

Physical critical event	>	Physical normal event	>	Non-secure interrupt
Virtual critical event	>	Virtual normal event	>	Virtual interrupt

If the system supports both physical and virtual interface class, it must define a priority between the SDEI instances. The priority must be defined such that the physical events always preempt the virtual events:

Physical critical event	>	Physical normal event	>	Virtual critical event	>	Virtual normal event

The event membership in a class and the priority of the class is IMPLEMENTATION DEFINED. For instance, an error-handling event is a good fit for a critical priority in the respective class for physical and virtual SDEI clients. This will ensure that the error event can still preempt the PE, which is handling a normal priority event in the respective class.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.2.2 Nesting depth of running SDEI event handlers

The nesting depth of running SDEI event handlers depends on the number of priority classes provided by the dispatcher. With two levels of priority, the active running depth of the SDEI handler is two. The nesting depth of various SDEI instances is as follows:

Physical SDEI

The physical SDEI is managed by the EL3 dispatcher. If EL3 dispatcher supports both normal and critical priority events, there can be a maximum of two *physical SDEI,* events one from each priority class that is handled by a PE. If the dispatcher implementation supports only one of the classes, the maximum depth of nested handler is one.

Virtual SDEI

The virtual SDEI is managed by the hypervisor dispatcher. If the hypervisor dispatcher implementation supports both normal and critical priority events, there can be a maximum of two virtual *SDEI* events, one from each priority class that is handled by a PE. If there is only one priority class supported, the maximum depth of nested handler is one.

4.4 Event number allocation

An SDEI event number is a 32-bit signed number. The event space is partitioned into vendor defined events and standard events.

Standard events are defined by Arm. The current version of the specification only defines event number 0, which is a software signaled SDEI event. More events will be added to the standard event space in future.

Vendor defined events are implementation specific events. They are allocated in the vendor event space. All bound events must be allocated in this space.

The format of the event number is as follows:

Table 3: Event number definition

Bit Numbers	Bit Mask	Description
31	0x8000_0000	Must be zero.
30	0x4000_0000	Vendor bit used to specify vendor defined events. Set to 0 for standard events.
29:24	0x3F00_0000	Must be zero. Reserved for future use.
23:0	0x00FF_FFFF	Event numbers

The current version of the specification defines the following range of event numbers,

Table 4: Event number space

Event numbers	Description
0	Software signaled event
0x0000_0001 - 0x00FF_FFFF	Standard events. Reserved for future expansion.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

0x4000_0000 - 0x40FF_FFFF	Vendor defined events.

The physical and virtual SDEI instance must implement the standard event 0, which denotes a software signaled event. The event numbers are local to the SDEI instance but for simplicity, Arm recommends that event numbers are reused across different SDEI instances for similar events. For instance, a watchdog event can have the same SDEI event number for both physical and virtual SDEI instance.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5 Interface

This chapter describes the SDEI calls, the handler execution context and the return error codes of the interface calls.

5.1 SDEI calls

Clients invoke the SDEI calls by using either the SMC or HVC conduit as described in *Conduits on page 13*. The functions adhere to the *SMC Calling Conventions* and in particular, the register usage follows the specification for SMC64 calls.

All functions can be safely called within or outside an event handler, unless otherwise stated.

In a system with multiple clients, an interface call by a client retrieves information or operates on events that are available only to that client.

An SDEI instance must implement all functions described in this chapter.

The *SMC Calling Conventions* requires that all unimplemented functions return an *Unknown SMC Function Identifier* which maps to the `NOT_SUPPORTED` error code. This specification follows this convention. Therefore, a `NOT_SUPPORTED` error code indicates an implementation that does not support SDEI.

5.1.1 SDEI_VERSION

Description	Returns the version of the SDEI implementation.	
Parameter		
uint32 Function ID:	0xC400_0020	
Return		
int64	On success, the format of the version number is as follows:	
	Bit [63] Must be 0. **Bits [62:48]** Major revision: must be 1 for this revision of SDEI. **Bits [47:32]** Minor revision: must be 1 for this revision of SDEI. **Bits [31:0]** Vendor-defined version number.	
	On error:	
	`NOT_SUPPORTED`	SDEI is not supported.

5.1.1.1 Usage

Each implemented SDEI instance must support this call and return its version number.

The version number is a 63-bit unsigned integer, with the upper 31 bits denoting the major and minor revision, and the lower 32 bits denoting a vendor-defined number.

The following rules apply to the version numbering:

- Different major revision values indicate possibly incompatible functions. A newer major revision might:
  - Introduce new functions.
  - Deprecate older functions.
  - Change behavior of existing functions.
- For two versions, A and B, for which the major revisions are identical, if the minor revision of B is greater than the minor revision of A, then all functions in

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A must work in B. However, it is possible for B to have a higher function count than A.

- This specification does not define the format of the vendor-defined version number. However, a higher value must indicate a newer version.

5.1.1.2 Dispatcher responsibilities

If a valid version number is returned by this call, the SDEI instance must implement all SDEI calls described in this chapter.

5.1.2 SDEI_EVENT_REGISTER

Description	Register a handler to the specified event.
Parameters	
uint32 Function ID	0xC400_0021
int32 event	Event number.
uint64 entry_point_address	Entry point address at EL_C for the event handler.
uint64 ep_argument	User-defined argument passed to entry point routine.
uint64 flags	Registration Flags, **Bits [63:2]**: Reserved for future use. Must be zero. **Bit [1]**: *relative_mode:* Specifies whether the entry_point_address passed to this function is an absolute address or relative to the Vector Base Address Register (VBAR) for EL_C. Possible values are: • 0: entry_point_address is an absolute address • 1: entry_point_address is relative to VBAR Note**:** *relative_mode* is an optional feature. Whether an implementation supports *relative_mode* may be determined using the SDEI_FEATURES function. **Bit [0]**: *routing_mode* Routing mode for shared event. Possible values are: • 0: `RM_ANY` (Route to any PE in the system) • 1: `RM_PE` (Route to the PE specified by affinity)
uint64 affinity	Affinity argument. The format of this field depends on the selected routing mode. Currently the format is defined only when the selected routing mode is `RM_PE`. With `RM_PE`, this field follows the MPIDR format as described in *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile*. **Bits [40:63]**: Must be zero. **Bits [32:39]:** Aff3 : Match Aff3 of target PE MPIDR. **Bits [24:31]**: Must be zero. **Bits [16:23]:** Aff2 : Match Aff2 of target PE MPIDR. **Bits [8:15]:** Aff1 : Match Aff1 of target PE MPIDR. **Bits [0:7]:** Aff0 : Match Aff0 of target PE MPIDR.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Return		
int64	SUCCESS	Event registered successfully.
	NOT_SUPPORTED	SDEI is not supported.
	INVALID_PARAMETERS	Invalid parameters in the call.
	DENIED	Inappropriate event state.

5.1.2.1 Usage

After the specified event has been registered, the event is disabled and the handler is not called until the event is enabled. The event handler always executes in EL_C.

The SDEI_EVENT_REGISTER call, changes the event handler state, see the state diagram in *Event handler states and properties* on page 44 for more information.

Any event that triggers before it has been registered a handler is either ignored or remains pending. It is IMPLEMENTATION DEFINED whether the event is ignored or remains pending.

A client can register only one handler to one event at a time. If the client needs to register a different handler to the event or change the *ep_argument* for the event, it must unregister the event.

For a shared event, SDEI_EVENT_REGISTER registers the handler globally for the calling client. For a private event, this call only registers the handler for the calling PE.

5.1.2.2 Parameters

- *event* is the event number that the client is registering. The event number is either provided by the dispatcher or created dynamic using the SDEI_INTERRUPT_BIND call.
- *entry_point_address* holds the entry point of the event handler. Event handlers are executed at EL_C and in the client translation regime. Specifying an invalid address will result an exception (e.g. translation fault, permission fault) in the client as defined in the ARMv8-A architecture when the invalid address is executed. Refer to *Event handler context* for more details about the context of event handlers.
- *ep_argument* can be any value as defined by the client software. Typical usage might have this as an alternate stack pointer or a structure defining the event context for the client software. Dispatcher software must pass this argument unchanged to the handler routine.
- *flags* define the flags for registering the event. Currently only the routing mode of an event is specified through the flags. The unused fields are reserved for future expansion of the specification. Routing mode is valid only for a shared event. For a private event, the routing mode is ignored. This specification requires the support of the following routing modes,
    - RM_ANY: The event is routed to any PE in the system.
    - RM_PE: The event is routed to a single PE as specified by the affinity argument.
- *affinity* specifies the PE affinity of a shared event. The format of this field depends on the selected routing mode. This parameter is unused for private events and shared events with routing mode RM_ANY. For routing mode RM_PE, *affinity* specifies the PE to which the event will be routed and the format follows the MPIDR value as described in *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile.* New formats might be introduced in later revisions of the specification.

5.1.2.3 Client responsibilities

Before a SDEI_EVENT_REGISTER call, the client must,

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- Identify the event number, obtained either from the dispatcher or by creating a bound event, see *Bound events* on page 49.
- For a shared event, specify an appropriate routing model and affinity if applicable.

The client must specify a valid entry point address. The client must make sure that the address is valid to all the PEs that can handle this event. The address can be a physical or virtual address and must be valid according to the translation regime that will be used when the event is enabled.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned for any of the following,
  - Invalid event number.
  - Invalid event handler address, if the dispatcher can determine that it is not valid.
  - For shared events, invalid routing mode or invalid affinity specified for routing mode, `RM_PE`.
- DENIED is returned,
  - If the event is already registered by the client.

If the event is in state *handler-unregister-pending*, see Event handler states and properties on page 44 for more information. The state of the event can be examined using SDEI_EVENT_STATUS call on page 28.

5.1.3 SDEI_EVENT_ENABLE

Description	Enable the specified event.	
Parameters		
uint32 Function ID:	0xC400_0022	
int32 event	Event number	
Return		
int64	`SUCCESS`	Event enabled successfully.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid parameters in the call.
	`DENIED`	Inappropriate event handler state.

5.1.3.1 Usage

This call enables the event for the client. The client can receive an event only after this call. Any event that triggers before the enable and after the register will stay pending and only be delivered to the client when the event is enabled.

The `SDEI_EVENT_ENABLE` call, changes the event handler state, see the state diagram in *Event handler states and properties* on page 44 for more information.

Enabling an event, which is already enabled, is permitted and has no effect. For a private event, this call enables the event only for the calling PE. For a shared event, this call enables the event globally for the calling client.

5.1.3.2 Client responsibilities

Before a `SDEI_EVENT_ENABLE` call, the client must:

- Register the event using `SDEI_EVENT_REGISTER`.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- Perform any optional configuration of the event.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned for an unknown event number.
- DENIED is returned if,
  - Event is not registered by the client.
  - Event handler is in *handler-unregister-pending* state. See Event handler states and properties on page 44 to know more about this.

5.1.3.3 Dispatcher responsibilities

The specific responsibilities of the dispatcher at the time that an event is enabled are use-case dependent. See *Appendix A: Implementing use cases*. For example, enabling a bound event may involve enabling a physical interrupt at the interrupt controller. Enabling a system event may require system-specific interactions.

5.1.4 SDEI_EVENT_DISABLE

Description	Disable the specified event.	
Parameters		
uint32 Function ID	0xC400_0023	
int32 event	Event number.	
Return		
int64	`SUCCESS`	Event disabled successfully.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid event number in the call.
	`DENIED`	Inappropriate event handler state.

5.1.4.1 Usage

This call disables the event for the client. When the event is disabled, no further events will be passed to client. Any event that triggers after it is disabled will stay pending and passed to the client when the event is enabled again. This call has no effect on a currently running event handler.

The `SDEI_EVENT_DISABLE` call, changes the event handler state, see the state diagram in *Event handler states and properties* on page 44 for more information.

Disabling an event, which is already disabled, is permitted and has no effect. For a private event, this call disables the event only for the calling PE. For a shared event, this call disables the event globally for the calling client.

5.1.4.2 Client responsibilities

Before a `SDEI_EVENT_DISABLE` call, the client must register the event using `SDEI_EVENT_REGISTER`.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned for an unknown event number.
- DENIED is returned if,
  - Event is not registered by the client.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- Event handler is in state *handler-unregister-pending*. See *Event handler states and properties* on page 44 to know more about this.

5.1.4.3 Dispatcher responsibilities

The specific responsibilities of the dispatcher when an event is disabled are use-case dependent. See *Appendix A: Implementing use cases*.

5.1.5 SDEI_EVENT_CONTEXT

<table>
<tr><td>Description</td><td colspan="2">To retrieve additional context information within an event handler.</td></tr>
<tr><td colspan="3">Parameters</td></tr>
<tr><td>uint32 Function ID</td><td colspan="2">0xC400_0024</td></tr>
<tr><td>uint32 param_id</td><td colspan="2">Parameter identifier. Possible values are,
0-17: Returns register X0-X17 of the client PE at the time of the event.
All other values for param_id are reserved for future use.</td></tr>
<tr><td colspan="3">Return</td></tr>
<tr><td colspan="3">On success</td></tr>
<tr><td>int64</td><td colspan="2">The value of the register as specified by param_id will be returned.</td></tr>
<tr><td colspan="3">On error,</td></tr>
<tr><td rowspan="3">int64</td><td><code>NOT_SUPPORTED</code></td><td>SDEI is not supported.</td></tr>
<tr><td><code>INVALID_PARAMETERS</code></td><td>Invalid param_id value in the call.</td></tr>
<tr><td><code>DENIED</code></td><td>The event handler is not running for the calling PE.</td></tr>
</table>

5.1.5.1 Usage

This call is used to retrieve additional context information from an event handler. When the dispatcher passes event to the client, the event number, ep_argument, interrupted PC and PSTATE is provided to the event handler through registers X0 – X3 respectively. This call can be used to retrieve the value of registers X0 – X17 of the PE which receives the event. See *Event handler context* page 40 for more information about the event handler context.

The type of context information returned through this call is selected using the *param_id* argument. Currently the following values of *param_id* are defined,

- 0-17: General purpose registers X0-X17 of the PE at the time of the event.

The `SDEI_EVENT_CONTEXT` call is typically used within an event handler. The call is valid only when the event handler property *handler-running* is set to TRUE), see *Event handler states and properties* on page 44. For private and shared events, this call can be invoked only from the PE that received the event.

5.1.5.2 Client responsibilities

If the event handler property *handler-running* is set to TRUE, the call always succeeds, and the requested register value is returned.

If the client executes the call when the *handler-running* property is FALSE or from a different PE than the one in which the event handler is running, the call will always fail with one of the following error codes,

- INVALID_PARAMETERS is returned for an unknown param_id value.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- DENIED is returned if the *handler-running* property is FALSE on the calling PE.

5.1.5.3 Dispatcher responsibilities

The context information must be available until the event handler is completed. The call must always succeed when the event is running.

5.1.6 SDEI_EVENT_COMPLETE

Description	Complete the event handling and resume execution from the interrupted context.	
Parameters		
uint32 Function ID	0xC400_0025	
uint32 status_code	0: EV_HANDLED	The event was handled successfully.
	1: EV_FAILED	The event was not handled.
Return		
On success this call never returns and resumes the execution from the point of interruption.		
On error,		
int64	NOT_SUPPORTED	SDEI is not supported.
	DENIED	No event handler running for the calling PE.

5.1.6.1 Usage

This call is used by the handler to complete the handling of the event. The event handler function must never return, it has no valid return address, and always ends with a `SDEI_EVENT_COMPLETE` call or `SDEI_EVENT_COMPLETE_AND_RESUME` call, see page 26. If successful, the call resumes the execution of the client from where the event occurred.

This call is valid only when the *handler-running* property is set to TRUE. This call sets the event handler property *handler-running* to FALSE. See *Event handler states and properties* on page 44.

For private and shared events, this call can be invoked only from the PE that received the event. For a private event, this call completes the event handling for the calling PE. For a shared event, this call completes the event handling globally.

5.1.6.2 Parameters

status_code can be used to indicate the status of event handling. The following status values are defined,

- EV_HANDLED indicates that the event was successfully handled by the client.
- EV_FAILED indicates that the client failed to handle the event. The associated action for this status code is platform and event specific. For example, with a software watchdog event, the platform implementation might choose to reset the system for this status code.

5.1.6.3 Client responsibilities

The client must never return from the event handler and must always call `SDEI_EVENT_COMPLETE` or `SDEI_EVENT_COMPLETE_AND_RESUME` to end the event handling and resume execution in the client. If the client fails to complete the event handling, the client might behave in an unpredictable way.

For a hardware event due to a physical interrupt, the trigger could be edge triggered or level sensitive. For both type of triggers, the client must call `SDEI_EVENT_COMPLETE` to exit

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

the event handler. In addition, for a level sensitive interrupt, it is up to the client to clear the interrupt at source before calling `SDEI_EVENT_COMPLETE` from the event handler.

The call returns only if there is an error and the client must handle the following potential return error code,

- DENIED is returned if,
    - the *handler-running* property is set to FALSE on the calling PE

5.1.6.4 Dispatcher responsibilities

The *status_code* available for a given event and for a given client is IMPLEMENTATION DEFINED by the platform.

5.1.7 SDEI_EVENT_COMPLETE_AND_RESUME

Description	Complete the event handling and resume execution at the specified address in EL_C.	
Parameters		
uint32 Function ID	0xC400_0026	
uint64 resume_addr	Address in client to resume the execution from.	
Return		
On success this call never returns and resumes the client from the provided address.		
On error,		
int64	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid resume address.
	`DENIED`	No event handler running for the calling PE.

5.1.7.1 Usage

This call is used by the handler to complete the handling of an event and resume the execution from the client at a specified address. This call is similar to `SDEI_EVENT_COMPLETE` except that the execution resumes from EL_C at the specified address in the client as opposed to the interrupted context.

If successful, the call never returns to the caller and resumes the execution at *resume_addr* from EL_C.

This call is valid only when the *handler-running* property is set to TRUE. The call sets *handler-running* to FALSE. See *Event handler states and properties* on page 44.

For private and shared events, this call can be invoked only from the PE that received the event. For a private event, this call completes the event handling for the calling PE. For a shared event, this call completes the event handling globally.

This call is particularly useful when the interrupted Exception level is different from EL_C and the client needs to do additional processing of the event before resuming the interrupted context. For example, if a firmware SDEI event interrupted the guest OS and the hypervisor wants to inject a virtual SDEI event into the guest after completing the handling at hypervisor.

The resume handler Execution state mimics a synchronous exception handler for EL_C where the exception return address is set to the PC at which the SDEI event was taken. Refer *Event resume context* on page 42 for more details about the context.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.7.2 Parameters

resume_addr must be a valid resume address in EL_C. The resume address must be valid in translation regime of the client at the point when the event occurred. Specifying an invalid address will result an exception (e.g. translation fault, permission fault) in the client as defined in the ARMv8-A architecture when the invalid address is executed.

5.1.7.3 Client responsibilities

The client must use `SDEI_EVENT_COMPLETE_AND_RESUME` to end the event handling and resume execution from the client. This is typically useful in the form of 'post-processing' an event, for example, cascading events to a lower Exception level such as a hypervisor injecting virtual events to a guest, or choosing to preempt or terminate guest execution. If the client fails to complete the event handling, it might result in UNPREDICTABLE behavior in the client.

The call returns only if there is an error and the client must handle the following potential return error code,

- INVALID_PARAMETERS is returned if the dispatcher can identify that the resume address is invalid, for example address is not 4-byte aligned.
- DENIED is returned if the event handler property *handler-running* is set to FALSE on the calling PE.

5.1.7.4 Dispatcher responsibilities

If the client fails to complete the event handling, the dispatcher implementation must ensure that this does not affect the dispatcher from functioning.

If the client is executing a yielding SMC call and SDEI event interrupts the Secure execution of the SMC call, and the client completes the handler using `SDEI_EVENT_COMPLETE_AND_RESUME`, the resume handler must be executed prior to resuming the secure execution. The SMC call can be restarted later from the client using an IMPLEMENTATION DEFINED mechanism agreed with the Secure software.

5.1.8 SDEI_EVENT_UNREGISTER

Description	Unregister an event handler.	
Parameters		
uint32 Function ID	0xC400_0027	
int32 event	Event number	
Return		
int64	`SUCCESS`	Event successfully unregistered.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid event number.
	`DENIED`	Event not registered by client.
	`PENDING`	Event handler is running and unregister will be pending.

5.1.8.1 Usage

This call is used to unregister the client from receiving any future events specified by *event*. Any event that triggers after a successful unregister, might be queued or discarded with the specific behavior being event and platform specific. This call has no effect on a currently running event handler.

The `SDEI_EVENT_UNREGISTER` call, changes the event handler state, see the state diagram in *Event handler states and properties* on page 44 for more information.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

This function completes asynchronously without waiting for any current handlers executing on other PEs to complete. If unregister is called when the event handler is running, unregister will be pending and return code will be set to `PENDING`. For instance, if the call is invoked from the event handler, unregister would return `PENDING`.

When this call returns `PENDING`, the unregister will be performed when the handler completes the event-handling using `SDEI_EVENT_COMPLETE` or `SDEI_EVENT_COMPLETE_AND_RESUME`. The client will not receive any future events of type *event*, after the client completes the currently running handler. Calling unregister on an event in the unregister-pending state is permitted and will return `PENDING`.

For a shared event, this call will unregister the event globally. For a private event, this call will unregister the event for the calling PE.

5.1.8.2 Client responsibilities

Before a `SDEI_EVENT_UNREGISTER` call, the client must register the event using `SDEI_EVENT_REGISTER`.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned for an unknown event number.
- DENIED is returned if event is not registered by the client.
- PENDING is returned if the event handler property *handler-running* is set to TRUE. This error indicates that any resource that the event handler uses cannot be freed by the client until the event handler completes the event handling.

5.1.8.3 Dispatcher responsibilities

The dispatcher implementation must ensure that after a successful unregister, no events will be delivered to the client. If the event handler is in *handler-unregister-pending* state, no events will be delivered to the client after the client completes the currently running handler.

5.1.9 SDEI_EVENT_STATUS

Description	Retrieve the status of an event.
Parameters	
uint32 Function ID	0xC400_0028
int32 event	Event number.
Return	
On success the format of the return value is as follows,	

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

<table>
<tr><td>int64</td><td colspan="2">Bits[63-3]: Must be zero.
Bit[2]: Running bit. Possible values are,
• 0: Event handler is not-running.
• 1: Event handler is running.
Bit[1]: Enable bit. Possible values are,
• 0: Event handler is disabled.
• 1: Event handler is enabled.
Bit[0]: Register bit. Possible values are,
• 0: Event handler is unregistered.
• 1: Event handler is registered.
See Table 13 on page 46 for more information on how to map the return value to the event handler states.</td></tr>
<tr><td colspan="3">On error,</td></tr>
<tr><td rowspan="2">int64</td><td><code>NOT_SUPPORTED</code></td><td>SDEI is not supported.</td></tr>
<tr><td><code>INVALID_PARAMETERS</code></td><td>Invalid event number in the call.</td></tr>
</table>

5.1.9.1 Usage

This call can be used to retrieve the status of an event. The status of the event returned by this call can be directly mapped to event handler states, see *Event handler states and properties* on page 44.

5.1.9.2 Client responsibilities

The event status is dynamic and can change even when the return status is inspected. Therefore, the status must not be used to perform operations that depend on the current status of the event unless the client synchronizes itself not to change the event handler state.

The client must handle the following potential return error code,

- `INVALID_PARAMETERS` is returned for an unknown event number.

5.1.10 SDEI_EVENT_GET_INFO

<table>
<tr><td>Description</td><td>Retrieves the information of an event.</td></tr>
<tr><td colspan="2">Parameters</td></tr>
<tr><td>uint32 Function ID</td><td>0xC400_0029</td></tr>
<tr><td>int32 event</td><td>Event number</td></tr>
<tr><td>uint32 info</td><td>Information requested, refer Parameter and Return values below</td></tr>
<tr><td colspan="2">Return</td></tr>
<tr><td colspan="2">int64 result
The return value on success depends on the information requested, refer Parameter and Return values below.</td></tr>
<tr><td colspan="2">On error the following error codes can be returned. Specific error codes that can be returned depending on info value are listed in Parameter and Return values below.</td></tr>
</table>

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

int64	NOT_SUPPORTED	SDEI is not supported.
	INVALID_PARAMETERS	Invalid event number or invalid info value
	DENIED	see *Parameter and Return values* below.

5.1.10.1 Usage

This call is used to retrieve the information of an SDEI event. The call is permitted in all states, however some return values depends on the event handler state, see *Parameter and Return values* below. For event handler states, see *Event handler states and properties* on page 44.

5.1.10.2 Parameter and Return values

The different values possible for *info* parameter and the associated return values are as follows:

Info value	Condition	Success return value	Additional error return
0: EV_TYPE		0: Private event. 1: Shared event.	
1: EV_SIGNALED		0: Event can be software signalled. 1: Event cannot be software signalled.	
2:EV_PRIORITY		0: Normal priority event. 1: Critical priority event.	
3: EV_ROUTING_MODE	Valid only for shared events and when event handler is in *handler-registered* state.	0: RM_ANY (Route to any PE in the system) 1: RM_PE (Route to the PE specified by affinity)	INVALID_PARAMETERS if event is not shared. DENIED if event is not registered.
4: EV_ROUTING_AFF	Valid only for shared events and RM_PE *routing_mode* and when event handler is in *handler-registered* state.	Affinity value. The format of this field depends on the selected routing mode. Currently the format is defined only when the selected routing mode is RM_PE. With RM_PE, this field follows the MPIDR format as described in *SDEI_EVENT_REGISTER* on page 20	INVALID_PARAMETERS if event is not shared or routing mode does not have an associated affinity. DENIED if event is not registered.
Other values of info are reserved for future use			

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.11 SDEI_EVENT_ROUTING_SET

Description	Sets the routing information of a shared event.	
Parameters		
uint32 Function ID	0xC400_002A	
int32 event	Event number.	
uint64 routing_mode	Event routing mode Bits [63:1]: Reserved for future use. Must be zero. Bit [0]: Routing mode Possible values are : • 0: `RM_ANY` (Route to any PE in the system) • 1: `RM_PE` (Route to the PE specified by affinity) see `SDEI_EVENT_REGISTER` on page 20.	
uint64 affinity	When *routing_mode* is `RM_PE`, this field follows the MPIDR format, see `SDEI_EVENT_REGISTER` on page 20.	
Return		
int64	`SUCCESS`	Event routing information set successfully.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid parameters.
	`DENIED`	Inappropriate event handler state.

5.1.11.1 Usage

This call is used to change the routing information of a shared event.

`SDEI_EVENT_ROUTING_SET` is allowed only when the event handler is in *handler-registered* state, see *Event handler states and properties* on page 44.

5.1.11.2 Parameters

routing_mode denotes the routing mode information. This field follows the same format as described in `SDEI_EVENT_REGISTER` on page 20.

affinity denotes the affinity of the event. Currently this is used to specify the PE when the *routing_mode* is `RM_PE`, *see* `SDEI_EVENT_REGISTER` on page 20.

5.1.11.3 Client responsibilities

Before this call, the client must register the event using `SDEI_EVENT_REGISTER`.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned if,
  - Invalid event number.
  - Event is not a shared event.
  - Invalid routing mode or invalid affinity specified for routing mode.
- DENIED is returned if,
  - Event handler is in a state other than: *handler-registered.*

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.11.4 Dispatcher responsibilities

The implementation must support the *routing_mode* and *affinity* combination as permitted in `SDEI_EVENT_REGISTER`.

5.1.12 SDEI_PE_MASK

<table>
<tr><td>Description</td><td colspan="2">Mask the calling PE from receiving SDEI events.</td></tr>
<tr><td colspan="3">Parameters</td></tr>
<tr><td>uint32 Function ID</td><td colspan="2">0xC400_002B</td></tr>
<tr><td colspan="3">Return</td></tr>
<tr><td colspan="3">On success the return value can be,</td></tr>
<tr><td rowspan="2">int64</td><td><code>1:</code> Masked</td><td>The calling PE was masked by this call.</td></tr>
<tr><td><code>0:</code> Not masked</td><td>The PE was not masked by this call but is already masked.</td></tr>
<tr><td colspan="3">On error the return value can be,</td></tr>
<tr><td>int64</td><td><code>NOT_SUPPORTED</code></td><td>SDEI is not supported.</td></tr>
</table>

5.1.12.1 Usage

This call removes the calling PE from participating in SDEI event handling for the calling client. The call masks the PE from receiving SDEI events of both normal and critical priority, analogous to masking the IRQ for the PE. This behavior is independent of any event status (see *SDEI_EVENT_STATUS* on page 28) and only affects the PE making the call. The typical use scenario for this call is to temporarily remove the PE from SDEI handling in particular during power management operations. This call can be invoked by the client to mask the PE, whether or not the PE is already masked. The return value of this call indicates whether the PE was already masked.

The initial state for every PE is to have SDEI events masked. This means that the PE is masked from receiving any SDEI events following powerup or resume from power down state as defined in *Power State Coordination Interface* specification, see *Power management and SDEI events* on page 49 for more details.

The mask operation is per client and per PE. Each client has to execute this call from each PE that is to be removed from event handling. If an SDEI implementation is present for the client, then this call must always succeed.

5.1.12.2 Client responsibilities

If the client decides to mask SDEI events, before the call to `SDEI_PE_MASK`, it must ensure that the preemption is disabled in the client.

The client calls `SDEI_PE_MASK` mainly for two reasons:

1. Powerdown state

   Before a suspend to powerdown state through a `CPU_SUSPEND` call, or a `CPU_OFF` call, as defined in *Power State Coordination Interface*, the client must mask the PE. See *Power management and SDEI events* on page 49 for more details on this.

2. Mask PE from SDEI events

   To temporarily mask the PE from receiving SDEI events. Masking SDEI events could affect the handling of the events and it is expected that the masking be done very rarely.

In the scenario where a registered event is triggered and the PE has its SDEI events masked, the event will stay pending. In this case

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- If the event is private to the PE, it will be delivered to the PE after the unmask call.
- If the event is shared, other target PEs could handle it.

If a shared event has all of its target PEs masked, then the event remains pending, until a target PE for this event executes the unmask call.

5.1.12.3 Dispatcher responsibilities

The dispatcher implementation must ensure that the initial state for every PE is to have the SDEI events masked. During powerup or resume from powerdown state, the Dispatcher will mask the PE from receiving any SDEI events. This means that if the PE is powered up through the powerup entry point, then the PE must be masked for SDEI events by the dispatcher. For a *standby* power state, the dispatcher implementation must retain the mask status. See *Power management and SDEI events* on page 49 for more details on this.

5.1.13 SDEI_PE_UNMASK

Description	Unmask the PE to receive SDEI events.	
Parameters		
uint32 Function ID	0xC400_002C	
Return		
int64	`SUCCESS`	Unmasked PE successfully.
	`NOT_SUPPORTED`	SDEI is not supported.

5.1.13.1 Usage

This call enables the calling PE to participate in SDEI event handling for the client. The call unmasks the PE from receiving SDEI events of both normal and critical priority for the client, analogous to unmasking the IRQ for the PE to enable interrupt exceptions. This is independent of any event status (see *SDEI_EVENT_STATUS* on page 28) and only affects the client PE making this call. This call can be invoked by the client to unmask the PE irrespective of whether the PE is already masked. The initial state for every PE is to have events masked.

A PE can receive a triggered event only if the following

- The client has registered and enabled the SDEI event.
- The client has unmasked the PE from receiving an event.

The unmask operation is per client and per PE. Each client executes this call from each PE that is ready to participate in event handling. If SDEI implementation is present for the client, then this call must always succeed.

5.1.13.2 Client responsibilities

If the client disabled the preemption before executing `SDEI_PE_MASK`, then the client will have to enable the preemption after calling `SDEI_PE_UNMASK`.

The client calls `SDEI_PE_UNMASK` mainly for two reasons:

1. Powerup or resume from powerdown state

   During powerup or resume from powerdown state, the PE will be masked from receiving any SDEI events. The powerup and resume from powerdown state follows the definition as per the *Power State Coordination Interface* specification. The client is required to unmask the PE using this call from each PE that needs to participate in the SDEI event handling soon after it has done initial setup. This typically means that the client executes `SDEI_PE_UNMASK` from the client powerup entry point, when it is ready for handling the SDEI events. See *Power-on sequence* on page 49 for more details on this.

2. Unmask from a previous masking

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

If the client masked SDEI events for a PE using `SDEI_PE_MASK` and the call returned a `masked` state, then the client does `SDEI_PE_UNMASK` to unmask the PE.

5.1.13.3 Dispatcher responsibilities

The dispatcher implementation must dispatch any pending events for the calling PE.

5.1.14 SDEI_INTERRUPT_BIND

Description	Binds an interrupt to an event.	
Parameters		
uint32 Function ID:	0xC400_002D	
uint32 interrupt	Interrupt number.	
Return		
On success an event number is returned as follows,		
int64	**Bits [63:32]**: Must be zero. **Bits [31:0]**: Event number, see *Event number allocation* on page 17.	
On error,		
int64	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid interrupt number in the call.
	`DENIED`	Inappropriate interrupt state.
	`OUT_OF_RESOURCE`	The client has exceeded the available event binding slots, see *Bound events* on page 49.

5.1.14.1 Usage

This interface call can be used to bind any client interrupt in to a normal priority SDEI event. The interrupt can be a private peripheral interrupt (PPI) or a shared peripheral interrupt (SPI). Binding a software generated interrupt (SGI) is not allowed. Binding a PPI results in a private event and binding a SPI results in a shared event. The event number returned by this call is valid across all PEs for both private and shared events. Binding any type of interrupt that is already bound will return the same event number. Refer to *Bound events* on page 49 for more information.

The number of such bind slots that can exist in a system is platform specific and can be discovered using the `SDEI_FEATURES` call, see page 37. It is IMPLEMENTATION DEFINED on how the mapped event number is formed, within the vendor event space, refer *Event number allocation* on page 17.

5.1.14.2 Parameters

interrupt is the interrupt number that is to be promoted as an SDEI event. For physical SDEI instance, the interrupt will be a Non-secure interrupt and for virtual SDEI the interrupt number will be a virtual interrupt.

5.1.14.3 Client responsibilities

The client can bind any PPI or SPI as an event. When the interrupt is promoted to an event, the client will lose the ability to change any properties of the interrupt and only SDEI calls are allowed on the event. The interrupt, prior to binding must be owned and managed by the client, for example, in the case of a physical SDEI instance and the system uses

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

GIC, the interrupt must be in Group-1 Non-secure. Before binding the interrupt, the client must ensure that the interrupt is disabled at the interrupt controller and in inactive state, refer *Arm Generic Interrupt Controller Architecture Specification version 3.0* for more information.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned if
    - Interrupt number is invalid
    - Interrupt number is not allowed for binding. In particular, when the interrupt is not owned by the client, for example is a secure interrupt.
- DENIED is returned if the interrupt is not in Inactive state, as defined in *Arm Generic Interrupt Controller Architecture Specification version 3.0*.
- OUT_OF_RESOURCE is returned if the client has used all the available event bind slots, see *Bound events* on page 49.

5.1.14.4 Dispatcher responsibilities

The dispatcher must reserve a set of bind slots to be used by the client. The dispatcher must only allow the binding of interrupts owned by the clients. All bound interrupts must have their priority elevated to ensure that they can always preempt client execution.

When an interrupt that is bound as an event triggers, the dispatcher acknowledges the interrupt and delivers the event to the client. This ensures that only valid (non-spurious) interrupts are delivered to the client. When the client completes the event, the dispatcher does the end of interrupt. Dispatcher does the interrupt management from the interrupt controller side, however it is the client's responsibility to do the device side interrupt management.

5.1.15 SDEI_INTERRUPT_RELEASE

Description	Release the interrupt from event binding.	
Parameters		
uint32 Function ID:	0xC400_002E	
int32 event	Event number	
Return		
int64	`SUCCESS`	Event released successfully.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid event number in the call.
	`DENIED`	Inappropriate event state.

5.1.15.1 Usage

This interface call can be used to release an interrupt-event binding. If the release call is executed on an interrupt that is not currently bound, an error code `INVALID_PARAMETERS` will be returned.

As the event binding is global, to release a private event, the event must be in *handler-unregistered* state for all registered PEs. To release a shared event, the event handler must be in *handler-unregistered* state, see the state diagram in *Event handler states and properties* on page 44 for more information.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.15.2 Parameters

event is the event number that was returned from an earlier call to `SDEI_INTERRUPT_BIND`, see page 34.

5.1.15.3 Client responsibilities

A successful `SDEI_INTERRUPT_RELEASE` call effectively removes the event number and no further SDEI calls will be valid on this event.

The client must also handle the following potential return error codes,

- INVALID_PARAMETERS is returned if
  - Event number is invalid.
  - Event number is not bound.
- DENIED is returned if the event handler is in a state other than *handler-unregistered*.

5.1.15.4 Dispatcher responsibilities

After a successful release, the free slot returns to the pool of bind slots. The client can reuse the bind slot for the same or different interrupt number. After releasing a bound event, it is IMPLEMENTATION DEFINED, if the dispatcher returns the same or different event number for a subsequent bind of the same interrupt. The dispatcher must ensure that all registered PEs have completed (not pending) unregistering the handler before releasing an event.

The dispatcher implementation must restore any configuration of an interrupt that is done during SDEI_INTERRUPT_BIND, so that the client owns and manages the interrupt. For example, for a physical SDEI instance with a system that uses GIC, the interrupt must be in Group-1 Non-secure. The interrupt must be disabled at the interrupt controller.

5.1.16 SDEI_EVENT_SIGNAL

Description	Signal a software event to a client PE.	
Parameters		
uint32 Function ID:	0xC400_002F	
int32 event	event number, must be zero.	
uint64 target_pe	Target PE including self.	
Return		
int64	`SUCCESS`	Event was successfully set as pending.
	`NOT_SUPPORTED`	SDEI is not supported.
	`INVALID_PARAMETERS`	Invalid event or target PE.

5.1.16.1 Usage

This interface call can be used to signal a software event to a client PE. The target PE that receives the event is specified in target_pe argument in the MPIDR format similar to `SDEI_EVENT_REGISTER` call on page 20 call. The target PE can be same as the calling PE.

The software signaled event will be delivered as a private event to the target PE. To handle the event, the target PE must register and enable the event. The signaling is an asynchronous process and sets the event pending on the target PE. The event has edge-triggered semantics and the number of event signals may not correspond to the number of times the handler is invoked in the target PE.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.16.2 Parameters

event must be an event that can be signaled. This can be queried using `SDEI_EVENT_GET_INFO`. This version of the specification supports only event number 0 that can be signaled, refer to *Event number allocation* on page 17 for standard events.

target_pe specifies the PE that receives the event. This is the MPIDR of the target_pe as described in `SDEI_EVENT_REGISTER` call on page 20.

5.1.16.3 Client responsibilities

A `SUCCESS` status code from this call indicates that the event was successfully set as pending. However, this does not indicate that the event will be handled by the target PE. To handle the event the target PE must:

- Register for the event.
- Enable the event.
- Unmask SDEI events.

A successful return cannot be relied as an indication for the start or completion of the handler in the target PE, instead the client must use its own mechanisms to determine handler execution status.

5.1.16.4 Dispatcher responsibilities

The dispatcher implementation must ensure that the updates to shared data structures by client is observable by the target PE before generating the event.

5.1.17 SDEI_FEATURES

Description	Enumerate SDEI features
Parameters	
uint32 Function ID:	0xC400_0030
uint32 feature	This argument specifies the feature that is queried. Currently the following value is defined:
	0: BIND_SLOTS - returns the number of private events and shared event slots available for binding. All other values are reserved.
	1: RELATIVE_MODE – returns whether the implementation supports the *relative_mode* feature of SDEI_EVENT_REGISTER.
Return	
On success, the format of the return value is as follows,	
int64	For BIND_SLOTS as the parameter, the return value is as follows:
	Bits [63:32] : must be zero **Bits [31:16]** : The count of shared event slots allocated in the system. **Bits [15:0]** : The count of private event slots allocated in the system.
	For RELATIVE_MODE as the parameter, the return value is as follows:
	0: *relative_mode* is not supported. **1:** *relative_mode* is supported.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

On error,		
int64	NOT_SUPPORTED	SDEI is not supported.
	INVALID_PARAMETERS	Invalid feature requested.

5.1.17.1 Usage

This interface call is used to query SDEI features that are implemented in the system. Currently only the number of bind slots can be queried using this interface. Return value for the bind slots provide the number of private bind slots and shared bind slots that are allocated by the platform for binding.

5.1.17.2 Parameters

parameter used to specify the feature that is queried.

This version of specification defines only the following value:

0 : BIND_SLOTS

All other values are reserved for future.

5.1.18 SDEI_PRIVATE_RESET

Description	Resets private SDEI data of the calling PE.	
Parameters		
uint32 Function ID	0xC400_0031	
Return		
int64	SUCCESS	All SDEI data for the calling PE was reset.
	NOT_SUPPORTED	SDEI is not supported.
	DENIED	At least one event handler was running while this call was invoked.

5.1.18.1 Usage

This call is used to reset all private SDEI event registrations of the calling PE. The call loops through all registered private events of the calling PE and unregisters the event. All private events will be unregistered if this call is executed outside an event handler. If this call is executed from an SDEI event handler, all handlers will be unregistered except for the running handlers. With two levels of event priority implemented, there can be at most two events that are currently running, see *Nesting depth of running SDEI event handlers* on page 17. The running handler will be in state *handler-unregister-pending* until the event is completed, see the state diagram in *Event handler states and properties* on page 44 for more information. This call has no effect on shared events.

5.1.18.2 Client responsibilities

For a PE reset or PE shutdown, in addition to executing this call, the client must also mask the PE and re-target or unregister any shared events targeting this PE.

This call is useful in a reset scenario, especially when the private events registered by the PE are unknown.

To successfully execute this call:

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- It is expected that no private event handlers would have the event handler property *handler-running* set to TRUE. If an event handler is running, unregister will be pending until the event is completed.

5.1.18.3 Dispatcher responsibilities

All private events must be unregistered or unregister-pending (if called from event handler) following this call. Any auxiliary information relating to the event registration must be cleared or must be set similar to the warm boot state of the PE.

5.1.19 SDEI_SHARED_RESET

Description	Resets shared SDEI data.	
Parameters		
uint32 Function ID	0xC400_0032	
Return		
int64	`SUCCESS`	All SDEI system data was reset.
	`NOT_SUPPORTED`	SDEI is not supported.
	`DENIED`	Event was running while this call was invoked.

5.1.19.1 Usage

This call is used to clear all system level SDEI data, which includes shared event registrations and interrupt-event bindings.

This call does the following:

- Loops through all registered shared events and unregisters the event.
- Releases all interrupts bound to events in the system.

The precondition for invoking `SDEI_SHARED_RESET` call is as follows:

- For each managed PE, unregister all private events or invoke `SDEI_PRIVATE_RESET`.
- For each managed PE, mask SDEI events using `SDEI_PE_MASK`. This is to stop the PE from handling any shared events.

To successfully execute this call:

- No shared events should have the event handler property *handler-running* set to TRUE. If an event handler is running, unregister is pending until the event is completed.
- All interrupt bound shared events must be unregistered (in step 1).
- All interrupt bound private events must be unregistered from all registered PEs.

This call has no effect on private events.

5.1.19.2 Client responsibilities

This call is useful in a reset scenario, especially when the various shared events registered and the interrupt-event bindings are unknown.

The call will return a `DENIED` error if there was at least one shared event that was running or at least one interrupt-event binding (private or shared) that was still registered.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

5.1.19.3 Dispatcher responsibilities

Any auxiliary information relating to the shared event registration or interrupt binding must be cleared.

5.2 Event context

When the registered event triggers, the event handler is executed preempting the client or lower level execution. The client interrupts cannot preempt the event handler. A normal priority event can be preempted by a critical priority event. A critical priority event handler will run to completion with respect to the client. The PE running a critical event handler might be preempted to a higher Exception level for other reasons like servicing a firmware interrupt. However, the client execution always resume in the running handler.

The event handler executes in a special execution context as explained in *Event handler context.*

Note that the SDEI handler execution environment is limited, as the state and consistency of the underlying OS kernel or hypervisor will be unknown. All resources must be pre-allocated and interactions with the OS or hypervisor must use methods that are guaranteed to be safe.

5.2.1 Event handler context

Event handlers can run even when the client has interrupts disabled and is executing critical code, such as during exception entry, switching execution threads, or handling interrupts and faults. Therefore, event handlers should not depend on the client state: for example, there may be no usable stack pointer in SP, or there may not be enough space on the stack.

Arm recommends that event handlers should use statically allocated memory or make use of the *ep_argument* parameter that is registered with the handler to provide working memory and stack space to run the handler.

On entry to the handler, the PE registers will contain the interrupted client Execution state, with the following exceptions:

- 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
- X2 is set to the interrupted PC
- X3 is set to the interrupted PSTATE

PSTATE is modified as follows: DAIF = 0b1111, EL = EL_C, nRW = 0, SP = 1. Other PSTATE bits are populated according to the AArch64.TakeException() pseudocode function defined in *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile.* This will run the handler in AArch64 in the client Exception level using “handler” mode, with all interrupts masked, providing the event number, provided argument, interrupt PC, and interrupted PSTATE as parameters.

The additional interrupted PE context that would be available in an asynchronous exception (X0- X3) can be accessed in the event handler with the `SDEI_EVENT_CONTEXT` function.

The handler must preserve all registers except for X0…X17 on completion. Failure to do this will have undefined consequences when client execution resumes.

Register usage and handler requirements are summarized in the table below.

Table 5 Register Usage in SDEI Event Handlers

Register Name	Value on entry	Must be preserved
SP	Interrupted ELx stack pointer	Yes
X30	Interrupted Link Register	Yes

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

X29	Interrupted Frame Pointer	Yes
X19…X28	Interrupted Callee-saved registers	Yes
X18	Interrupted Platform Register	Yes
X16, X17	Interrupted intra-procedure-call scratch registers	No
X9…X15	Interrupted Temporary registers	No
X8	Interrupted Indirect result location register	No
X4…X7	Interrupted Parameter registers	No
X3	Interrupted PSTATE	No
X2	Interrupted PC	No
X1	ep_parameter registered with the handler	No
X0	Event number	No
SP_EL0	Interrupted EL0 stack pointer	Yes
F*n*, D*n*, Q*n*	Interrupted SIMD&FP registers	Yes
PSTATE	As interrupted with DAIF = 0b1111 EL = EL_C nRW = 0 SP = 1 Other PSTATE bits are populated according to the AArch64.TakeException() pseudocode function that is defined in *Arm Architecture Reference Manual, Armv8 for Armv8-A architecture profile.*	No
System registers	Interrupted register values	Yes

5.2.1.1 Client responsibilities

Within the handler, the client software is not restricted in its use of general-purpose, SIMD&FP, or System registers. The handler must ensure that on completion, all registers except X0-X17 are restored to their original value.

The handler may use SIMD&FP registers but the handler cannot assume that access to these registers is enabled at entry to the handler. The register and associated system control state must be restored before the handler completes.

The handler may modify the accessible System registers, but these must be restored before the handler completes.

The handler code should not enable asynchronous exceptions by clearing any of the PSTATE.DAIF bits and should not cause synchronous exceptions to the client Exception level.

Details of the register context on entry to the handler are described in Table 5 above.

The event handler property *handler-running* will be set to TRUE, see *Event handler states and properties* on page 44. The handler may call ‘fast’ calls as defined in *SMC Calling Conventions* document or hypervisor services including SDEI itself. SDEI calls that would affect the current PE or event will take effect when the current handler has completed running.

As part of the event handling, the client might have to clear the event source, in particular for events caused by a level sensitive hardware trigger.

It is not permitted to invoke yielding SMC calls from the SDEI event handler. Yielding calls are defined as calls that can be preempted by a Non-secure interrupt, see *SMC Calling*

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Conventions for more information. However fast (atomic) SMC calls are allowed but it is IMPLEMENTATION DEFINED if they behaves the same as called from outside the handler.

The event handling must always end with a `SDEI_EVENT_COMPLETE` or `SDEI_EVENT_COMPLETE_AND_RESUME` call, and should not return from the handler as it has no valid return address.

5.2.1.2 Dispatcher responsibilities

The dispatcher must ensure that the SDEI event handler takes precedence over any other client execution on a PE if the event is triggered and the PE has SDEI unmasked and there is a registered and enabled SDEI handler for the event.

The dispatcher must ensure that SDEI event handlers cannot interrupt each other, except that a critical event handler must interrupt a running normal event handler. If multiple events of the same priority are triggered on a PE, the handlers must run in sequence. See *Appendix C: Pseudocode for dispatcher* on page 58.

The dispatcher must save and later restore the client Execution state that may be modified/corrupted by the handler (see *Event handler context* on page 40). If multiple event priorities are implemented, the dispatcher must be able to save the PE state of all nested, running handlers

The dispatcher must ensure that all register values observed by the handler are those belonging to the client execution context – in particular the handler must not be able to access any residual register state from higher Exception levels. The dispatcher is permitted to do this via traps with emulation or lazy state switching for registers that support this (For example, SIMD&FP registers).

The dispatcher may allow an SDEI event handler to interrupt a secure firmware or hypervisor operation (for the physical and virtual SDEI respectively), but the dispatcher is permitted to defer the execution of the handler until any such operation has completed.

The dispatcher must run the handler to completion, even if the handler modifies the state of the event or PE through SDEI calls. Changes to the SDEI mask status of the PE, or state of the event only take full effect when the handler completes via `SDEI_EVENT_COMPLETE` or `SDEI_EVENT_COMPLETE_AND_RESUME`.

5.2.2 Event resume context

When an SDEI event handler ends by calling `SDEI_EVENT_COMPLETE_AND_RESUME` (see page 26), execution does not resume from the originally interrupted context. Instead, execution resumes at the address provided to the call in the *resume address* parameter. This context is almost equivalent to a simulated synchronous exception to the client Exception level (EL_C), where ELR_EL_C and $SPSR_EL_C$ are set to the originally interrupted PC and PSTATE respectively.

On resumption, the PE registers will contain the interrupted Execution state, with the following exceptions:

- The PC is set to the *resume_addr as provided in the* `SDEI_EVENT_COMPLETE_AND_RESUME` call.
- PSTATE is modified as follows: DAIF = 0b1111, EL = EL_C, nRW = 0, SP = 1
- ELR_EL_C is set to the PC when the event was taken, where ELR_EL_C is ELR_EL2 for hypervisor client and ELR_EL1 for an OS client.
- $SPSR_EL_C$ is set to the PSTATE when the event was taken, where $SPSR_EL_C$ is SPSR_EL2 for hypervisor client and SPSR_EL1 for an OS client.

All registers other than the ones mentioned above will have the interrupted value.

Apart from the register state changes as mentioned above, the event handler context is different from the resume handler context as follows:

- The resume context is outside of the SDEI handler, and therefore SDEI events are no longer implicitly masked, as they were during the SDEI handler. This means that the resume handler may be interrupted by an SDEI event. If necessary, the client can use

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`SDEI_PE_MASK/SDEI_PE_UNMASK` to prevent such an interruption until the resume handler has been able to save critical exception state: ELR and SPSR.

- The `SDEI_EVENT_CONTEXT` call for accessing event handler register state is not available.
- The resume handler can return to the interrupted context using ERET, as for a normal synchronous exception.

5.3 Return Codes

Table 6 defines the possible values for error codes used with the interface functions. The error return type is 64-bit signed integer. Zero and positive values denotes success and negative values indicates error.

The error values defined here aligns with the values defined in the *Power State Coordination Interface* document.

Table 6 Return code and values

Name	Description in SDEI context	Value
SUCCESS	The call completed successfully.	0
NOT_SUPPORTED	SDEI is not supported by the platform.	-1
INVALID_PARAMETERS	Some or all of the parameters passed to the call are invalid.	-2
DENIED	The call is not allowed because of the inappropriate event state.	-3
PENDING	The operation is pending.	-5
OUT_OF_RESOURCE	Out of resource error.	-10

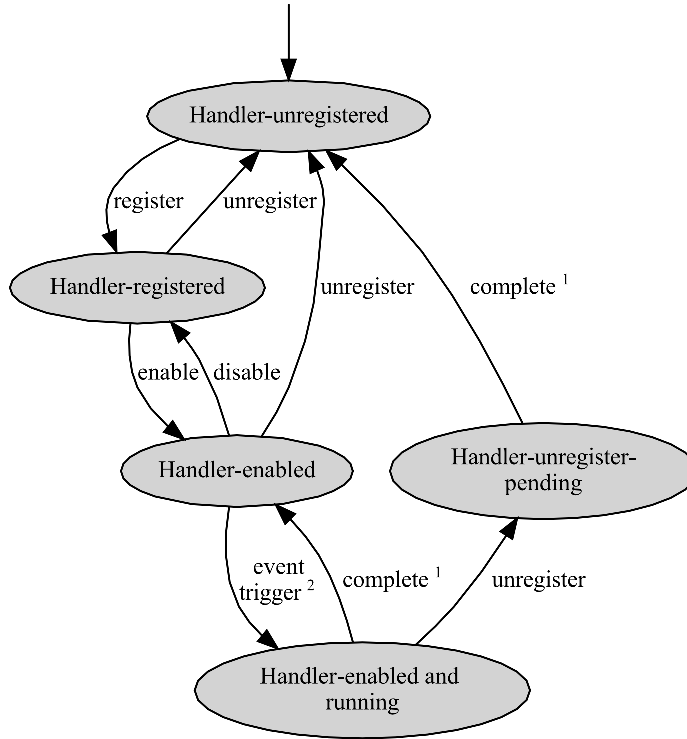
Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

6 Programmers' Overview

This chapter describes the event handler states and SDEI calls available from each of those states. It also describes sequences of common operations performed using SDEI calls.

6.1 Event handler states and properties

Figure 2 shows the SDEI event handler state diagram. All of the transitions are caused by interface calls except for the *event trigger* transition.



1 complete can be either SDEI_EVENT_COMPLETE or SDEI_EVENT_COMPLETE_AND_RESUME.
2 This transition is not caused by an interface call. It occurs when the event triggers.

Figure 2 Event handler state diagram

Note: Invalid transitions and transitions that cause no change in state are not shown.

6.1.1 Description

The state model operates independently on each event. For private events, there is a separate event handler status for each PE. For a shared event, there is a single global event status for the client.

For a bound event, the event must be created first by binding the interrupt and then it follows the exact same state transitions as a platform event.

The following discussion on event states and interface calls assumes that the interface calls operate on the same event for which the handler state is examined.

An SDEI event handler can be in any of the following states,

handler-unregistered state

The *handler-unregistered* state denotes that there is no event handler registered for the event. If the event triggers while in this state, the event is either ignored or remains pending. It is IMPLEMENTATION DEFINED whether the event is ignored or remains pending.

For a shared bound event, the event can be released only from this state. For a private bound event, the event can be released when all the registered handlers for this event are in the *handler-unregistered* state.

 Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The event handler transitions to the *handler-registered* state on a call to SDEI_EVENT_REGISTER, see Table 7. The SDEI calls available from this state are listed in Table 8.

Table 7 State transition from unregistered state

State	Interface call	Next state
handler-unregistered	SDEI_EVENT_REGISTER	handler-registered

Table 8 Interface calls available from unregistered state

State	Interface calls
handler-unregistered	SDEI_EVENT_REGISTER SDEI_EVENT_STATUS SDEI_INTERRUPT_RELEASE

handler-registered state

The *handler-registered* state indicates that an event handler is registered for the event and the event is disabled.

From this state, various configurations can be performed including change of routing, and any optional platform configurations. The client will not receive any events in this state as the event is disabled. Any event that triggers in this state will remain pending until it transitions to *handler-enabled* state.

The handler can transition to the *handler-enabled* state or the *handler-unregistered* state by calling SDEI_EVENT_ENABLE or SDEI_EVENT_UNREGISTER respectively. Table 9 lists the possible transitions. The SDEI calls available from this state are listed in Table 10.

Table 9 State transitions from handler-registered state

State	Command	Next state
handler-registered	SDEI_EVENT_ENABLE	handler-enabled
	SDEI_EVENT_UNREGISTER	handler-unregistered

Table 10 Interface calls available from handler-registered state

State	Available interface calls
Handler-registered	SDEI_EVENT_STATUS SDEI_EVENT_ENABLE SDEI_EVENT_DISABLE SDEI_EVENT_GET_INFO SDEI_EVENT_ROUTING_SET

handler-enabled state

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The client must receive an event only when the event handler is in the *handler-enabled* state. Being in this state implies that the event handler is registered and enabled.

The handler can transition to the *handler-registered* state or *handler-unregistered* state by calling `SDEI_EVENT_DISABLE` or `SDEI_EVENT_UNREGISTER` respectively. Table 11 lists the possible transitions. The various interface calls available from this state are listed in Table 12.

Table 11 State transitions from enabled state

State	Command	Next state
handler-enabled	`SDEI_EVENT_DISABLE`	handler-registered
	`SDEI_EVENT_UNREGISTER`	handler-unregistered

Table 12 Interface calls available from enabled state

State	Available interface calls

Handler-enabled	`SDEI_EVENT_STATUS`
	`SDEI_EVENT_ENABLE`
	`SDEI_EVENT_DISABLE`
	`SDEI_EVENT_GET_INFO`
	`SDEI_EVENT_UNREGISTER`

Handler-running property

Handler-running is a property that can be associated with all the event handler states. The description of the *handler-unregistered*, *handler-registered* and *handler-enabled* states in the preceding sections assume the *handler-running* property is set to FALSE. Setting *handler-running* to TRUE, creates three more states:

- *handler-unregister-pending*
- *handler-registered and handler-running*
- *handler-enabled and handler-running*

The *handler-running* property is true when the event handler is executing on a PE. This property is set to TRUE when the event handler begins executing and is set to FALSE when the event handler executes `SDEI_EVENT_COMPLETE` or `SDEI_EVENT_COMPLETE_AND_RESUME`.

The *handler-running* property is associated with a given event and the PE that handles that event. The interface calls that are available when *handler-running* is TRUE depends on the state with which handler-running is associated. See the descriptions of the respective SDEI calls for more information. However, the interface calls `SDEI_EVENT_CONTEXT`, `SDEI_EVENT_COMPLETE` and `SDEI_EVENT_COMPLETE_AND_RESUME` are always available with *handler-running* set for the PE that handles the event.

A description of each of the states with the *handler-running* property is given in Table 13. The state bit vector column shows the status bits as returned by `SDEI_EVENT_STATUS`. The bit vector is of the format (Reg,Ena,Run) where Reg denotes the handler-register state bit, Ena denotes the *handler-enabled* state bit, and Run denotes the *handler-running* bit.

Table 13 Handler-running property association with states

State	Handler-running	State bit vector (Reg, Ena, Run) [1]	Description

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

handler-unregistered	False	(0,0,0)	Initial state, when handler is not registered and not running, called *handler-unregistered* state.
	True	(0,0,1)	Handler is unregistered, but event handler is running. This happens when the handler is unregistered while event handler is running, called *handler-unregister-pending* state.
handler-registered	False	(1,0,0)	Handler is registered, not enabled and not running, called *handler-registered* state.
	True	(1,0,1)	Event handler is registered and running. This happens when the event handler is disabled when the handler is running, called *handler-registered and handler-running* state.
handler-enabled	False	(1,1,0)	Event handler is registered, enabled but not running, called *handler-enabled* state.
	True	(1,1,1)	Event is registered, enabled and running, called *handler-enabled and handler-running* state.

[1] The state vector (0,1,1) and (0,1,0) are undefined.

6.1.2 Interface calls and states

The interface calls available from each state and property are summarized in Table 14. The list assumes that the state and the interface calls apply to the same event .

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table 14 Interface calls and corresponding event state

Event state	SDEI_EVENT_REGISTER	SDEI_INTERRUPT_RELEASE	SDEI_EVENT_ENABLE	SDEI_EVENT_DISABLE	SDEI_EVENT_UNREGISTER		SDEI_EVENT_CONTEXT[2]	SDEI_EVENT_COMPLETE [2]	SDEI_EVENT_COMPLETE_AND_RESUME [2]
handler-unregistered	✓	✓							
handler-unregister-pending							✓	✓	✓
handler-registered			✓	✓	✓	✓			
handler-registered and handler-running			✓	✓	✓		✓	✓	✓
handler-enabled[1]			✓	✓	✓				
handler-enabled[1] and handler-running			✓	✓	✓		✓	✓	✓

[1] Event handler is registered.

The following interface calls can be invoked independent of the event handler state and must always be available.

- `SDEI_VERSION`
- `SDEI_EVENT_STATUS`
- `SDEI_PE_MASK`
- `SDEI_PE_UNMASK`
- `SDEI_INTERRUPT_BIND`
- `SDEI_EVENT_SIGNAL`

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- SDEI_FEATURES
- SDEI_PRIVATE_RESET
- SDEI_SHARED_RESET

The availability of interface call SDEI_EVENT_GET_INFO depends on the arguments, see page 29.

6.2 Event dispatching

When an event triggers, the dispatcher can dispatch the event only if all of the following are true:

1. The event is enabled.
2. The target PE for the event is enabled.
3. The PE is not already handling the same or a higher priority event.

Appendix C: Pseudocode for dispatcher on page 58 summarizes these conditions. If any of these conditions is not true, the event remains pending until they are all true.

If there are multiple pending events for a target PE within a given priority class, the order in which the handlers are invoked is IMPLEMENTATION DEFINED.

6.2.1 Recurring events

If an event triggers again after the handler for it at EL_C has completed, the handler executes again for the new trigger. However, if the event triggers again while the handler is handling the event, the handler can execute again after completing the event. This depends on the event source and how the event interacts with the system.

At any given time, only one instance of a shared event can be handled in a system. If the shared event retriggers while the handler for it is running, the event might stay pending. Concurrent handling of a shared event is not permitted.

6.3 Bound events

A *bound event* is an SDEI event that corresponds to a client interrupt. SDEI_INTERRUPT_BIND is used to associate an SDEI event with a client interrupt. This is called *binding*.

Binding is removed by using SDEI_INTERRUPT_RELEASE. Binding can only be removed when all of the PEs that registered for the event have unregistered.

Binding of a shared peripheral interrupt (SPI) creates a *shared bound event* and binding of a private peripheral interrupt (PPI) creates a *private bound event*. Software generated interrupts (SGI) cannot be used for binding.

In an SDEI instance, the number of bound events that can be created is IMPLEMENTATION DEFINED and are called *bind slots*. The number of bind slots available is discovered by using SDEI_FEATURES. Arm recommends that a dispatcher reserves at least two private bind slots and two shared bind slots.

The event number returned by SDEI_INTERRUPT_BIND is valid across all PEs for both private and shared events. For a given interrupt number, implementations are not required to provide the same event number for a second bind following a bind and release sequence.

6.4 Interface Discovery

The SDEI implementation can be detected by invoking the call SDEI_VERSION. If the platform is unaware of SDEI, it will return an error code of *unknown function identifier* as specified in *SMC Calling Conventions* document.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

To allow ease of integration, Arm recommends that the details of the SDEI implementation be specified in firmware description, for example using ACPI or FDT. Arm recommends that the entry describe:

- An interface version section, to allow updates to the interface, see `SDEI_VERSION`.
- Conduit information: SMC or HVC as appropriate, see *Interface and Exception levels* on page 12.

Refer *Appendix D: ACPI table definitions* on page 60 for SDEI ACPI definitions. It is IMPLEMENTATION DEFINED on how the platform event numbers are passed to the client.

6.5 Power management and SDEI events

The following sections describe how SDEI events are intended to work alongside power management calls. Refer to *Power State Coordination Interface* to know more about the power state terminology used here.

6.5.1 Power-on sequence

The dispatcher must ensure that following every PE reset, SDEI events are masked for the client. This includes cold-boot of primary/secondary cores or when the PE is powered-on using the `CPU_ON` PSCI call, or when the PE resumes from a powerdown state through the power up entry point. This is to prevent the dispatcher from dispatching events to the PE before it is ready. For instance the client must setup the PE memory translation to make the event handler address valid before enabling the event. When the client has done the initial setup for the PE, it must unmask the PE to receive SDEI events using SDEI_PE_UNMASK. Figure 3 shows an example sequence where client the unmasks the PE from `client_reset_entrypoint()`.

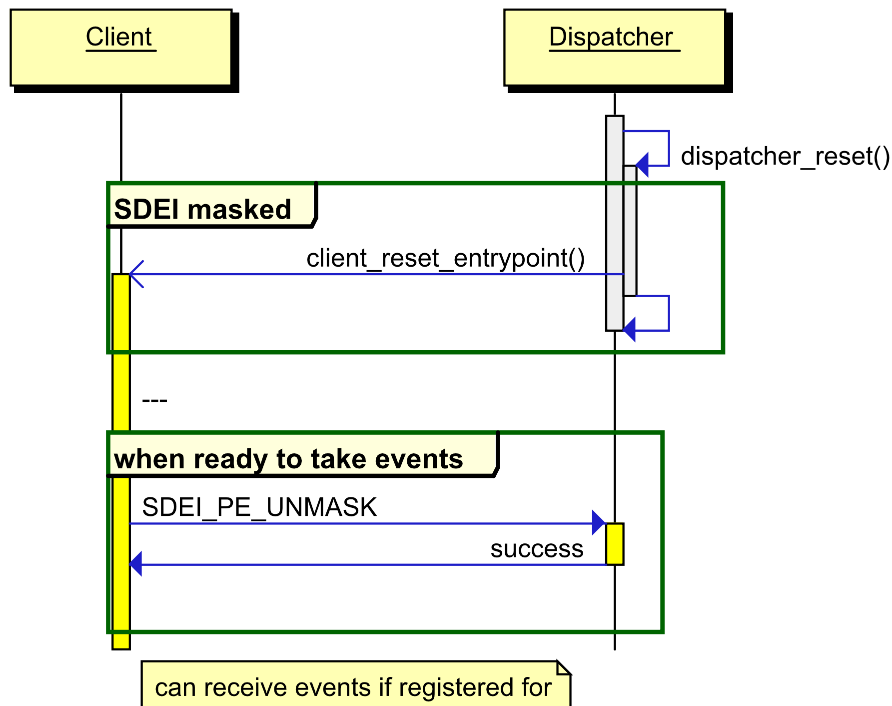


Figure 3 PE unmask sequence by client

6.5.2 Powerdown sequence

The powerdown sequence varies on how the PE is brought back online. The different possibilities are discussed below.

6.5.2.1 CPU_OFF

The `CPU_OFF` PSCI call is used to power down a core. The core can be brought back only using a `CPU_ON` call.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

6.5.2.1.1 Client Responsibilities

To avoid a race between switching off the PE and the event handling, prior to calling `CPU_OFF`, the client must,

- Unregister or disable any private events.
- Route any shared event targeting this PE (routing_mode=`RM_PE`) to another PE.
- Mask SDEI events for this PE.

6.5.2.1.2 Dispatcher Responsibilities

The dispatcher implementation must ensure that no SDEI event must be able to bring back the core online. If a client fails to disable/unregister the event or mask the PE, the receipt of an event will result in PLATFORM DEFINED behavior.

Figure 4 shows an example sequence of the `CPU_OFF` case.

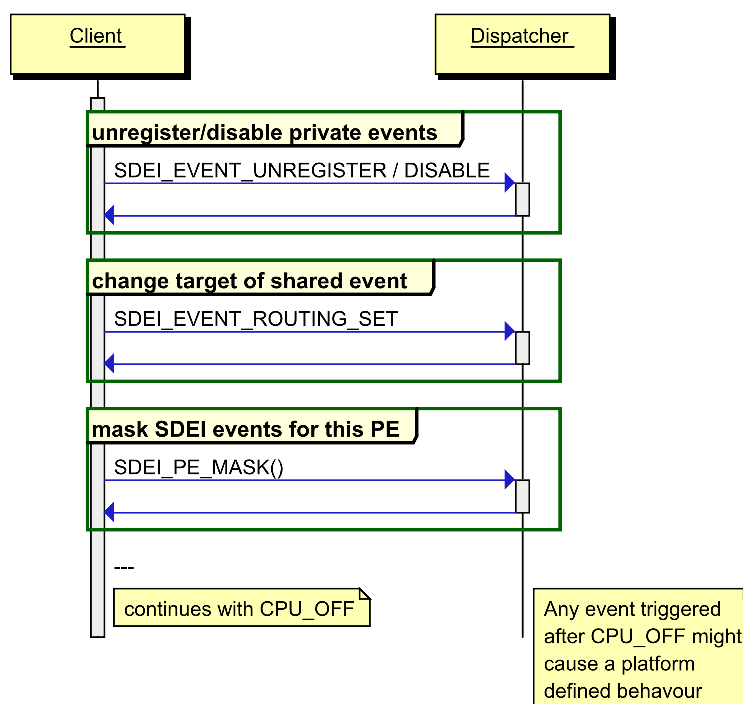


Figure 4 SDEI power down sequence

6.5.2.2 CPU_SUSPEND with powerdown state

The `CPU_SUSPEND` with powerdown PSCI call is used to suspend the PE to a power down state. From this state, the PE can potentially be woken up to handle an SDEI event.

Normally any event that is in enabled state can wake up the PE. Following wake up, private SDEI events which are enabled will stay pending for the PE. Shared events with routing mode as `RM_PE` and affinity set to this PE will stay pending. For all other shared events, the event might stay pending or might be handled by other event target PEs.

6.5.2.2.1 Client responsibilities

To avoid the race with the suspend operation, before requesting for the `CPU_SUSPEND`, the client must,

- Disable any private or shared event that is not a wakeup source for the PE.
- Shared event targeted to this PE (routing mode is `RM_PE`) and which is not a wakeup source must be routed to another PE.
- Mask SDEI events for this PE.

6.5.2.2.2 Dispatcher responsibilities

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The dispatcher must retain the status of all the events that the PE has registered. Any private or shared events that the PE has registered before going to suspend will remain valid when the PE wakes up. The dispatcher must mask the SDEI events for the PE when it wakes up and the event will stay pending until the Client calls SDEI_PE_UNMASK.

Figure 5 shows a possible sequence for a wakeup capable event.

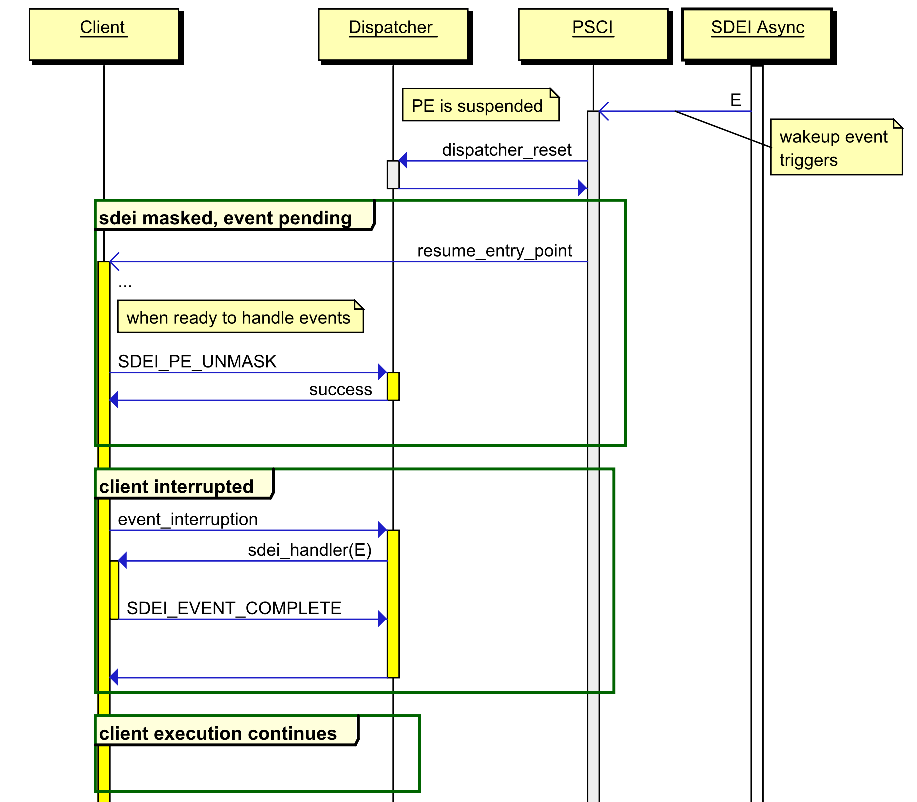


Figure 5 SDEI event wakeup sequence

6.5.3 CPU_SUSPEND with standby state

The CPU_SUSPEND with powerdown PSCI is used to suspend the PE to a low power retention state where all core context is maintained, and can be directly accessed on wakeup. The dispatcher retains all the event status for the events that the PE has registered. In addition to this, the SDEI mask status is retained by the dispatcher. After wakeup, the SDEI mask status will remain as it was left before the suspend. So if the client left the PE unmasked for SDEI events, the PE can receive SDEI events as soon as it comes out of the stand by state. If the client masked the SDEI events before the suspend, then the event will stay pending until the client calls SDEI_PE_UNMASK.

6.5.4 PSCI calls from SDEI handler

PSCI calls are permitted from an SDEI handler. The minimum set of PSCI calls that must work from the handler is listed below:

- PSCI_VERSION
- AFFINITIY_INFO
- PSCI_FEATURES
- SYSTEM_RESET
- SYSTEM_OFF
- CPU_OFF
- CPU_FREEZE
- CPU_ON

 Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The behavior of all the other PSCI calls is IMPLEMENTATION DEFINED. Refer to the *Power State Coordination Interface* document for a description of these functions.

The PSCI calls `SYSTEM_RESET`, `SYSTEM_OFF`, `CPU_OFF` and `CPU_FREEZE` if invoked within an SDEI handler will implicitly complete the SDEI handler(s) and proceed with the power operation performing the same behavior as if they were called outside the handler.

6.6 Registering and handling an event

To receive an SDEI event from the dispatcher, the client software has to

- Register for the event using `SDEI_EVENT_REGISTER`.
- Perform any optional platform specific event configuration.
- Enable the event using `SDEI_EVENT_ENABLE`.

When the event triggers and the PE is unmasked to receive SDEI events, the dispatcher passes the event to the client Exception level by invoking the event handler. To complete the event handling and to resume the execution, the event handler will call `SDEI_EVENT_COMPLETE`. Figure 6 shows the interface call sequence.

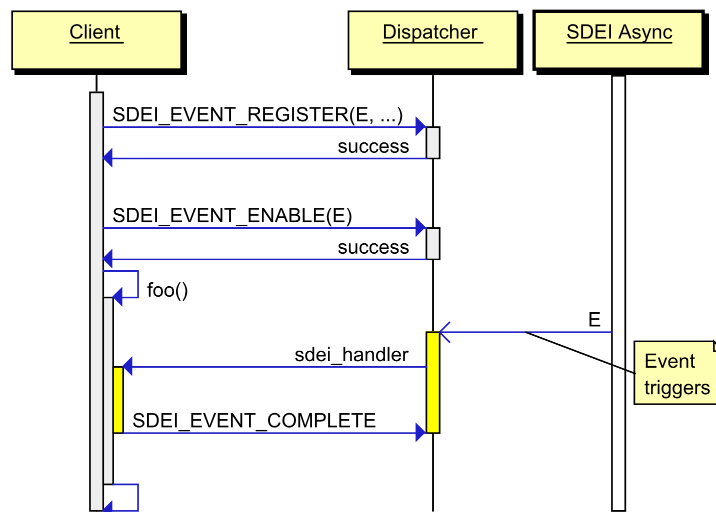


Figure 6 Event register and handling

6.7 Unregistering an event

To unregister an event, the client software has to issue a `SDEI_EVENT_UNREGISTER`. If the event handler is currently running in any PE, the unregister will stay pending. This will be indicated by a return code value, `PENDING` from the `SDEI_EVENT_UNREGISTER` call.

With the `PENDING` status, the unregister request will be queued until the event is completed using `SDEI_EVENT_COMPLETE`. If the client needs to wait for the unregister to complete, for instance to free the resources used by the handler, the client needs to wait until the event handler changes to *handler-unregistered* state, see *Event handler states and properties* on page 44. The status of the event can be examined using `SDEI_EVENT_STATUS` call.

 Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

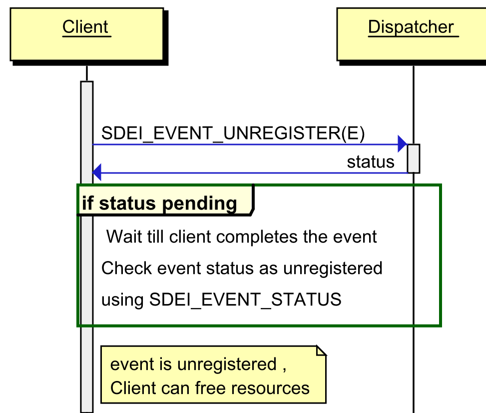


Figure 7 SDEI unregister sequence

6.8 Virtual SDEI events

The guest OS executing under a hypervisor can register for virtual SDEI events. With a hypervisor present, there will be two levels of delegation,

1. Firmware to hypervisor
2. Hypervisor to guest OS

The hypervisor must always trap and process OS SDEI calls from a guest OS as the guest OS cannot directly register with the firmware. For a Type-1 hypervisor the SDEI calls for physical SDEI originate from EL2. For a Type-2 hypervisor, the SDEI calls can originate from EL1 assuming that the calls are from the host OS.

The hypervisor can provide events owned by the hypervisor or firmware, see *Interface and Exception levels* on page 12.

Figure 8 shows a possible sequence where the guest OS registers for a virtual event (VE) which is generated from a physical event (E).

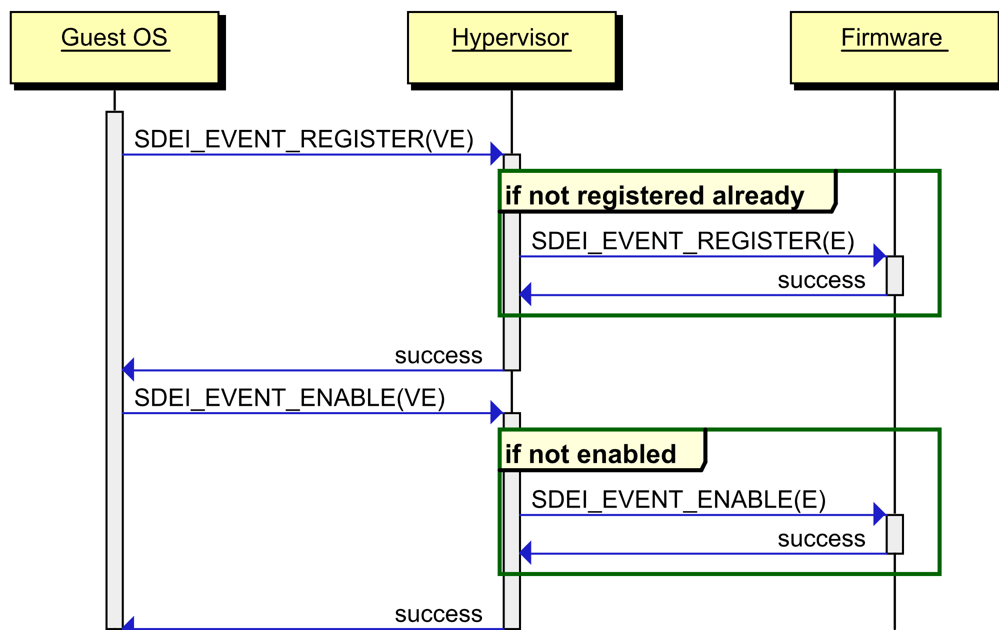


Figure 8 Guest OS SDEI event registration

For the physical events originating from firmware, the hypervisor implementation has the following options to process it:

1. Handle the event within the hypervisor.
2. Delegate a virtual event to the currently executing guest OS.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

3. Delegate virtual event to all registered guest OSs.

For the virtual events originating from the hypervisor, the hypervisor implementation can choose to do option (2) or (3).

It is event and hypervisor dispatcher specific if a virtual event can be registered by multiple guest OSs. In particular, the hypervisor implementation may allow shared physical events to be distributed to multiple guests. If sharing is allowed and the physical SDEI event triggers, then hypervisor dispatcher may have to generate the corresponding virtual event for each of the guests that has registered for this event. When a private or shared physical event targeting a physical PE triggers, the hypervisor dispatcher may have to generate virtual events for each registered guest with virtual PE mapped to this physical PE.

Figure 9 shows a possible sequence where a non-fatal event interrupted guest OS execution with the hypervisor injecting a virtual event to the guest OS.

When the physical event triggers, the firmware dispatcher calls the hypervisor SDEI handler. As the SDEI handler-execution in hypervisor is limited, to further process the event, the handler completes the event with resume as exception. While within the resume handler, the hypervisor dispatcher further delegates the event to the appropriate guest OS(s). The diagram shows option (2) where the virtual event is delivered to the currently executing guest OS.

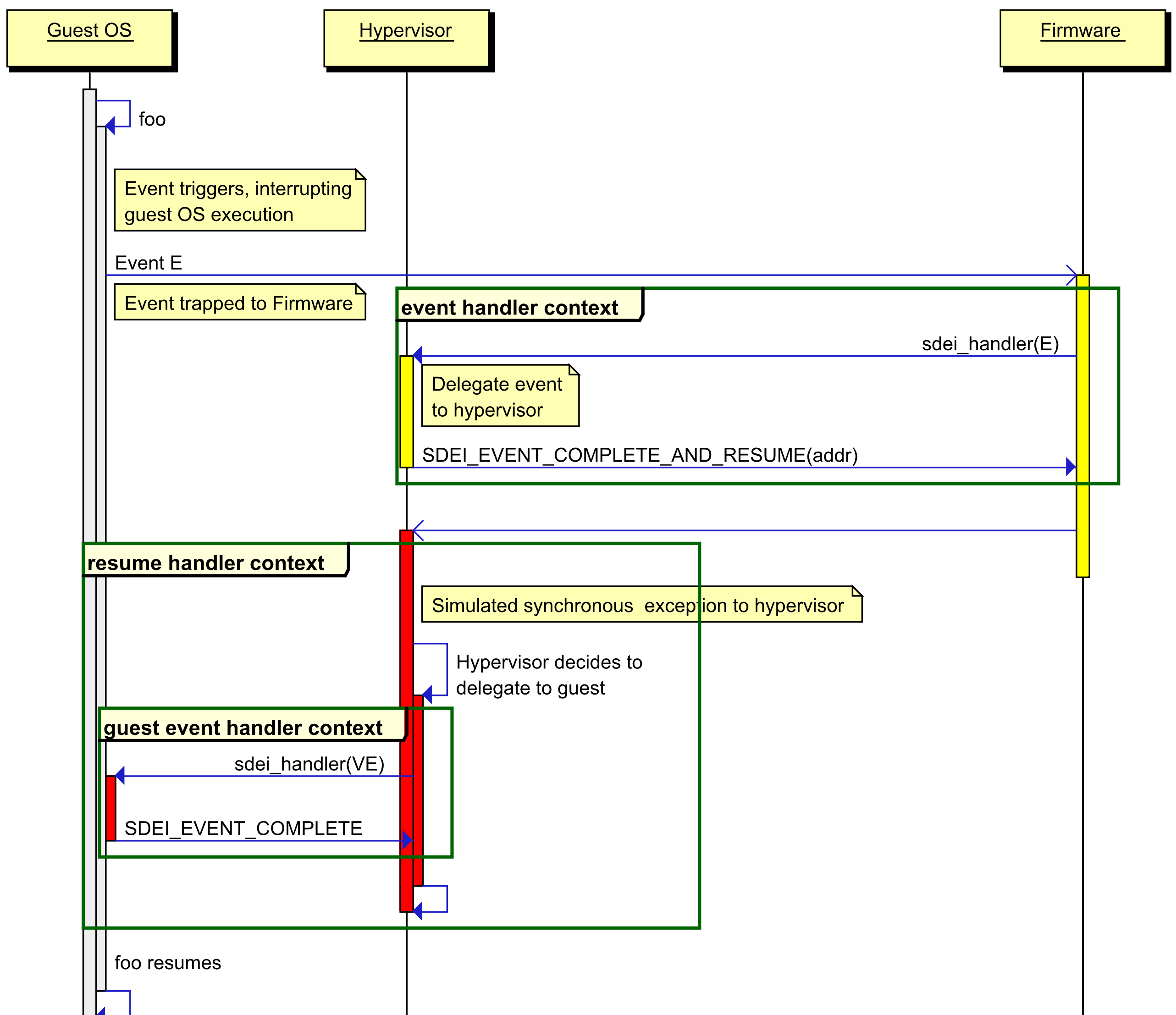


Figure 9 SDEI event delegation to guest OS

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

7 Appendix A: Implementing use cases

This section provides details about possible implementation models for various use cases mentioned in *Typical use cases* on page 9.

In general, asynchronous events may be implemented via physical interrupts. Synchronous events may be implemented using unused SPIs or using SGIs. This allows the flexibility of interrupt state management, routing and priority through the GIC.

7.1 Physical interrupt as SDEI event

This model is used when a physical interrupt has to be delivered as a SDEI event to the client. This is suitable for watchdog timers, profilers and for debugging. The following steps enumerates the sequence of handling this event,

1. Platform defines the event number of the interrupt and pass the event number to the client through IMPLEMENTATION DEFINED mechanism. Alternatively the client can bind the required interrupt and create the event.
2. Client software registers and enables the event.
3. When the event triggers, the dispatcher passes the event to the client through the registered entry point.
4. The handler routine similar to an interrupt handling routine, handles the event, clears the device interrupt and completes the event.

The event enable and disable operations directly enables and disables the physical interrupt. This model is more suitable when the dispatcher wants to provide a service.

7.2 Isolated physical interrupt as SDEI event

In this model, the physical interrupt is isolated from the SDEI event and any operation on the event does not directly apply to the physical interrupt. This model is suited for more complex use cases like error-handling. In this model, there can be distinct interrupts for each client and a different physical interrupt which is the source of the event.

For instance, let the platform define event E as an error-handling event. This event is generated whenever there is an error recovery procedure to be done by the client. The firmware might have its own handling for the error and in addition can provide an opportunity for the client to handle the error. The following steps enumerates the sequence of handling this event,

1. Client obtains event number from the dispatcher or creates a bound event.
2. The OS registers and enables the event.
3. When the hardware event triggers, a software component in dispatcher Exception level does the first level of processing.
4. The software in dispatcher triggers the SDEI event for client and passes control to the SDEI handler in the OS.
5. OS handles and completes the event. The action taken on the *status_code* of `SDEI_EVENT_COMPLETE` is platform specific.

The event enable and disable operations here applies only to the event generation for the client. Even if the client decide to disable the event, the event can still be handled by the software in dispatcher as indicated in step 3.

 Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

8 Appendix B: Implementation notes with GICv2 and GICv3 architecture

The following section explains the implementation details for GICv2 and GICv3 architectures to support SDEI events generated via an interrupt source.

8.1 GICv2

In GICv2 architecture there are two interrupt groups:

- Group 0 interrupts, which are always Secure.
- Group 1 interrupts, which are always Non-secure.

To implement SDEI in an Armv8-A system using GICv2, the Secure Group 0 interrupt has to be shared between Secure EL1 software and EL3 firmware.

One mechanism to implement SDEI in a GICv2 system is to configure the Secure Group 0 interrupts to trap in EL3. The dispatcher in EL3 can triage and delegate the event to Non-secure client. This is particularly useful if there is no Secure EL1 software or the Secure EL1 software does not use any secure interrupts.

If both Secure EL1 and EL3 software need to use the Group 0 interrupts, the interrupts can be trapped in EL3 in which case there will be an IMPLEMENTATION DEFINED mechanism to pass a Secure EL1 interrupt to the Secure EL1 software. Alternatively, the secure interrupt can be trapped in Secure EL1 and any unknown interrupts like SDEI can be passed to EL3 for further handling. The EL3 dispatcher can then triage and delegate the event to Non-secure client.

8.2 GICv3

In GICv3 architecture there are three interrupt groups:

- Group 0 interrupts, which are always secure.
- Secure Group 1 interrupts.
- Non-secure Group 1 interrupts.

Each group is mapped to either FIQ or IRQ interrupt lines by the GIC.

In an Armv8-A system using GICv3:

- Group 1 interrupts for the current Security state are mapped to the IRQ interrupt.
- Secure Group 0 interrupts and Group 1 interrupts for the other Security state are mapped to FIQ interrupt.

When the Secure EL1 software is handling the Secure Group 1 interrupt, it might have to disable the Non-secure Group 1 interrupt but allow Secure Group 0 interrupts to trigger. This can be achieved either by:

1. Secure EL1 software disables Non-secure Group 1 interrupt-group.
2. The system can allocate interrupt priorities in such a way that, handling a Secure Group 1 interrupt will effectively disable the Non-secure Group 1 interrupts but allow the Group 0 interrupts. The interrupt priorities can be assigned as follows:
    - All Group 0 interrupts are given priority over Secure Group1 interrupts
    - All Secure Group 1 interrupts are given priority over Non Secure Group 1 interrupts

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

9 Appendix C: Pseudocode for dispatcher

9.1 Private event dispatcher

Each event dispatcher can be logically thought as constantly checking for the condition to run an event handler while the client execution is progressing. This check is performed for each private event available for the client and for each PE present in the system. The pseudocode for the condition is summarized as follows:

```
Dispatcher(Client C)
 For each P in PE
  For each E in PrivateEvents
      IsSignalled(E, P) and
        IsEnabled(E, P) and
        IsUnmasked(P) and
        ((IsCriticalEvent(E) and !CriticalEventRunning(P, C)) ||
        (!IsCriticalEvent(E) and !EventRunning(P, C)))
```

Where,

`IsSignalled(E, P)`	Returns true if the event E is triggered for PE P and false otherwise.
`IsEnabled(E, P)`	Returns true if the event E is enabled for PE P and false otherwise.
`IsUnmasked(P)`	Returns true if the PE P is unmasked for SDEI events and false otherwise.
`IsCriticalEvent(E)`	Returns true if the event E is from critical priority class and false otherwise.
`CriticalEventRunning(P, C)`	Returns true if the PE P and Client C is running a critical priority event and false otherwise.
`EventRunning(P, C)`	Returns true if the PE P is running any – normal or critical SDEI event.

9.2 Shared event dispatcher

Each event dispatcher can be logically thought as constantly checking for the condition to run an event handler while the client execution is progressing. This check is performed for each shared event available for the client and for each PE present in the system. The pseudocode for the condition is summarized as follows:

```
Dispatcher(Client C)
 For each P in PE
  For each E in SharedEvents
      IsSignalled(E) and
        IsEnabled(E) and
        IsEventTarget(E, P)
        IsUnmasked(P) and
        ((IsCriticalEvent(E) and !CriticalEventRunning(P, C)) ||
        (!IsCriticalEvent(E) and !EventRunning(P, C)))
```

Where,

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`IsSignalled(E)`	Returns true if the event E is triggered and false otherwise.
`IsEnabled(E)`	Returns true if the event E is enabled and false otherwise.
`IsEventTarget(E, P)`	Returns true if either the event E is targeted for PE P using `RM_PE` and affinity as routing parameters or the event is currently targeted only for PE P, to ensure that a single trigger of a shared event is not handled by multiple PEs.
`IsUnmasked(P)`	Returns true if the PE P is unmasked for SDEI events and false otherwise.
`IsCriticalEvent(E)`	Returns true if the event E is from critical priority class and false otherwise.
`CriticalEventRunning(P, C)`	Returns true if the PE P and Client C is running a critical priority event and false otherwise.
`EventRunning(P, C)`	Returns true if the PE P is running any – normal or critical SDEI event.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

10 Appendix D: ACPI table definitions for SDEI

The SDEI ACPI table advertises the presence of the Software Delegated Exception Interface implemented by platform firmware or a hypervisor for use by an OS. The table consists only of a basic header with revision 1. Later revisions of the SDEI table may define additional fields.

Table 15 The SDEI table

Field	Byte Length	Byte Offset	Description
Signature	4	0	'SDEI`. Software Delegated Exception Interface Table.
Length	4	4	Length in bytes of this table.
Revision	1	8	This document describes revision 1 of the SDEI Table.
Checksum	1	9	The entire table must sum to zero.
OEMID	6	10	OEM ID.
OEM Table ID	8	16	For the SDEI table, the table ID is the manufacture model ID.
OEM Revision	4	24	OEM revision of the SDEI table for the supplied OEM Table ID.
Creator ID	4	28	The vendor ID of the utility that created the table.
Creator Revision	4	32	The revision of the utility that created the table.

When an OS discovers the SDEI table, it can call `SDEI_VERSION` to discover the version number of the interface and determine the features supported.

The conduit is discovered from the Fixed ACPI Description Table's (FADT) 'Arm Architecture boot flags' PSCI_USE_HVC flag. See *Advanced Configuration and Power Interface Specification v6.2*.

A typical use-case for SDEI is as a notification mechanism for firmware-first RAS errors using the ACPI Platform Error Interface (APEI). When the OS parses the Hardware Error Source Table's (HEST) Generic Hardware Error Source (GHES) entries, it should use the SDEI calls to register a handler for events that use SDEI as a notification method. SDEI uses notification type 11 in the 'Hardware Error Notification Structure', and stores the 32bit event number in the vector. If the platform uses multiple events for RAS, it should describe each one with a GHES entry. Private events should be represented with a single GHES entry.

The SDEI event handler in the OS should copy or consume the Common Platform Error Record (CPER) data associated with the GHES entry before calling SDEI_COMPLETE or SDEI_COMPLETE_AND_RESUME. Doing this ensures that the SDEI implementation in firmware will not overwrite CPER data while it is being used by the OS.

Table 16 shows an example Hardware Error Notification Structure for a single GHES notified by SDEI event number 804.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table 16 Example event definition for SDEI event 804

Field	Byte Length	Byte Offset	Value	Description
Type	1	0	11	SDEI
Length	4	1	28	Total length of the structure in bytes
Configuration Write Enabled	2	2	0	Ignored
Poll Interval	4	4	0	Ignored
Vector	4	8	804	The SDEI event number
Switch To Polling Threshold Value	4	12	0	Ignored
Switch To Polling Threshold Window	4	16	0	Ignored
Error Threshold Value	4	20	0	Ignored
Error Threshold Window	4	24	0	Ignored

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

11 Appendix E: ACPI definitions for SDEI events

This Appendix provides guidelines for describing SDEI events in ACPI namespace. Devices that signal events through SDEI can use the methods outlined in this Appendix to advertise these events to an OS. Likewise, an OS can use these methods to discover if SDEI signaling is supported by a device and identify the associated event numbers.

11.1 Describing SDEI events in ACPI Namespace

This specification defines a dedicated _DSM method that returns the SDEI event number for the SDEI event.

It is also possible for a device to support multiple events that are reported using this approach. Events are identified using an event index. The OS must pass the event index of the event as an argument. The number and meaning of indexes used by a device are specific to that device and must be specified in a standard or specification document that describes the device’s ACPI properties.

11.2 SDEI _DSM method

Table 17 provides a summary of the SDEI _DSM method.

Table 17 _DSM Method for SDEI Event Signaling

Argument	**Description**	**Value**
Arg0	UUID	The _DSM for SDEI signaling uses a reserved UUID of: e83a4698-e3a0-11eb-ba80-0242ac130004
Arg1	Revision ID	0x00
Arg2	Function Index	0x01: Get SDEI Event number.
Arg3	Event Index	For devices that support multiple SDEI events, specifies the index of the SDEI event. If a device supports a single event, this value must be 0x0. The available SDEI events and the meaning of each event index is defined in the binding for the HID for the device. If the caller passes an invalid index, an error code must be return as defined in the Return value definition.

The value returned by the _DSM method is the SDEI event number.

Return value: QWORD	
Bits [63:32]	Bits [31:0]
Reserved, must be zero.	The SDEI event number of the event. If an invalid Event Index was passed by the caller, the returned value must be the invalid event number 0x80000000.

The _DSM method must be placed within the scope of the device object that requires the SDEI event signaling. Presence of this method within the device’s scope is an indication to the OS that the device supports extended SDEI-based event signaling. The events that require the SDEI event signaling are specific to the device itself, and within the purview of the device driver. OSPM must invoke the _DSM method to obtain the event number used.

Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

11.3 Example

The following example reference code describes an ACPI device that supports SDEI-based event signaling with a single event:

```
Device (DEV0) {
   Name (_HID, "DEV0001")
   // Device is capable of signaling two events using SDEI
   Method (_DSM, 0x4, NotSerialized) {
      Switch (Arg0) {
         case (ToUUID (e83a4698-e3a0-11eb-ba80-0242ac130004))
         {
            switch (Arg2) {
            {
               // <ev1> = SDEI event number reserved for the first event
               // <ev2> = SDEI event number reserved for the second event
               case (1) {
                   if (Arg3 == 0)
                      return <ev1>;

                   if (Arg3 == 1)
                      return <ev2>;

                   return 0x8000_0000;
               }
            }
         }
      }
   }
}
```


Copyright © 2017-2021 ARM Limited or its affiliates. All rights reserved.
Non-Confidential