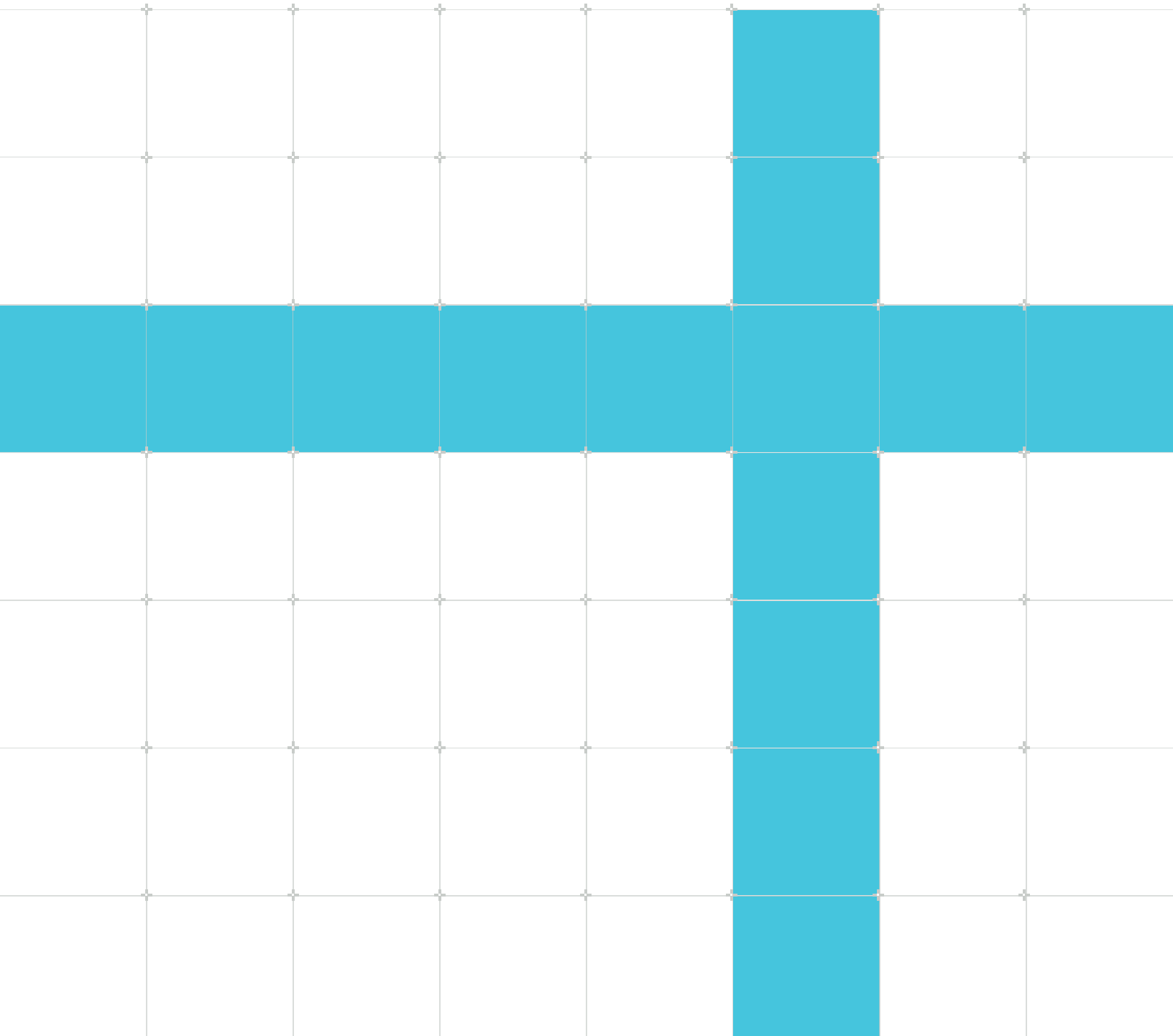
arm

Learn the architecture - AArch64 Exception Model

Version 1.3

Non-Confidential

Issue 01
102412_0103_01_en



Learn the architecture - AArch64 Exception Model

Release information

Document history

Issue	Date	Confidentiality	Change
0102-01	13 April 2022	Non-Confidential	Initial release
0102-02	1 July 2022	Non-Confidential	Minor text fix in Handling exceptions
0103-01	19 December 2022	Non-Confidential	Restructured and content added for Exception types and Exception handling

Proprietary Notice

REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

The AArch64 Exception Model guide introduces the exception and privilege model in Armv8-A and Armv9-A. It covers the different types of exceptions in the Arm architecture, and the behavior of the processor in relation to exceptions.

The contents are intended for developers of low-level code, such as boot code or kernels. It is particularly relevant to anyone writing code to set up or manage exceptions.

At the end of this guide you can check your knowledge. You will be able to list the Exception levels, state how execution can move between them, and name and describe the Execution states. You will also be able to detail the components of a simple AArch64 vector table as well as describe how exception handlers are used.

2. Privilege and Exception levels

Before we explain the details of the AArch64 Exception model, we need to introduce the concept of privilege.

Modern software is developed to be split into different modules, each with a different level of access to system and processor resources. An example of this is the split between the operating system kernel and user applications. The operating system needs to perform actions which we do not want a user application to be able to perform. The kernel needs a high level of access to system resources, whereas user applications need limited ability to configure the system. Privilege dictates which processor resources a software entity can see and control.

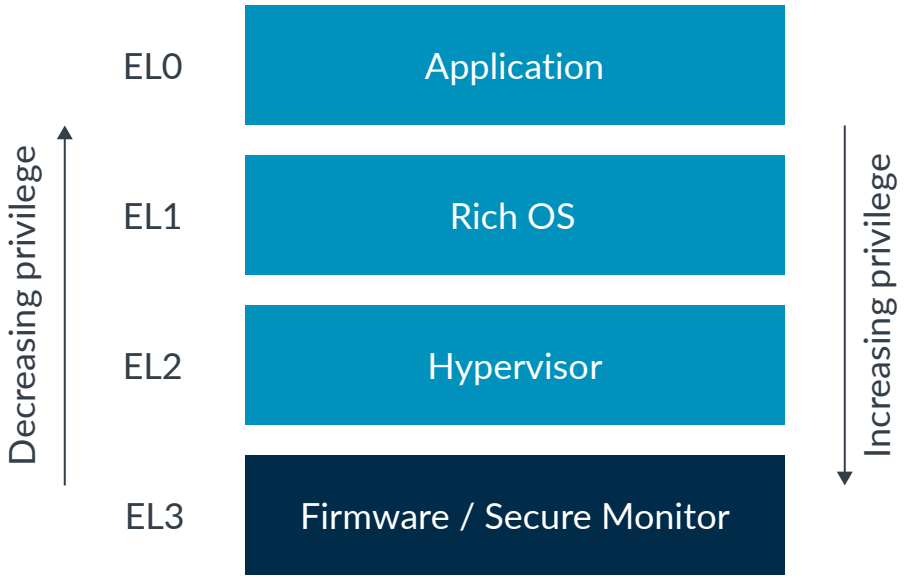
The AArch64 architectures enable this split by implementing different levels of privilege. The current privilege level can only change when the processor takes an exception or returns from an exception. Therefore, these privilege levels are referred to as Exception levels in the Arm architecture.

2.1 Exception levels

The name for privilege in AArch64 is Exception level, often abbreviated to EL. The Exception levels are numbered, normally abbreviated and referred to as EL<x>, where <x> is a number between 0 and 3. The higher the level of privilege the higher the number. For example, the lowest level of privilege is referred to as EL0.

As shown in Figure 2-1: Exception levels on page 9, there are four Exception levels: EL0, EL1, EL2 and EL3.

Figure 2-1: Exception levels



The architecture does not specify what software uses which Exception level. A common usage model is application code running at EL0, with a rich Operating System (OS) such as Linux running at EL1. EL2 may be used by a hypervisor, with EL3 used by firmware and security gateway code. For example, Linux can call firmware functions at EL3, using software interface standards, to abstract the intent from the lower-level details for powering on or off a core. This model means the bulk of PE processing typically occurs at EL0/1.

Note

The architecture does not enforce this software model, but standard software assumes this model. For this reason, the rest of this guide assumes this usage model.

The Exception level can only change when any of the following occur:

- Taking an exception
- Returning from an exception
- Processor reset
- During Debug state
- Exiting from Debug state

When taking an exception the Exception level can increase or stay the same. You can never move to a lower privilege level by taking an exception. When returning from an exception the Exception level can decrease or stay the same. You can never move to a higher privilege level by returning from an exception. We discuss this further in later sections of this guide.

2.2 Types of privilege

There are two types of privilege relevant to the AArch64 Exception model:

- Privilege in the memory system
- Privilege from the point of view of accessing processor resources

Both types of privilege are affected by the current privileged Exception level.

2.2.1 Memory privilege

The A-profile of the Arm architecture implements a virtual memory system, in which a Memory Management Unit (MMU) allows software to assign attributes to regions of memory. These attributes include read/write permissions that can be configured to allow separate access permissions for privileged and unprivileged accesses.

Memory access initiated when the processor is executing in EL0 are checked against the unprivileged access permissions. Memory accesses from EL1, EL2, and EL3 are checked against the privileged access permissions.

Because this memory configuration is programmed by software using the MMU's translation tables, you should consider the privilege necessary to program those tables. The MMU configuration is stored in System registers, and the ability to access those registers is also controlled by the current Exception level.

In the Arm architecture, registers are split into two main categories:

- Registers that provide system control or status reporting
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions

2.2.2 Register access

Configuration settings for AArch64 processors are held in a series of registers known as System registers. The combination of settings in the System registers defines the current processor context. Access to the System registers is controlled by the current Exception level.

For example, VBAR_EL1 is the Vector Base Address Register. We will describe what it is used for later in this guide. For now, what is important is the _EL1 suffix. This tells us that software needs at least EL1 privilege to access the register.

The architecture has many registers with conceptually similar functions that have names that differ only by their Exception level suffix. These are independent, individual registers that have their own encodings in the instruction set and will be implemented separately in hardware.

The registers have similar names to reflect that they perform similar tasks, but they are entirely independent registers with their own access semantics. The suffix of the System register name indicates the lowest Exception level from which that register can be accessed.

There is a System Control Register (SCTLR) for each implemented Exception level. Each register controls the architectural features for that EL, such as the MMU, caches and alignment checking:

SCTLR_EL1

Top-level system control for EL0 and EL1

SCTLR_EL2

Top-level system control for EL2

SCTLR_EL3

Top-level system control for EL3

EL1 and EL0 share the same MMU configuration and control is restricted to privileged code running at EL1. Therefore there is no SCTLR_EL0 and all control is from the EL1 accessible register. This model is generally followed for other control registers.

Higher Exception levels have the privilege to access registers that control lower levels. For example, EL2 has the privilege to access SCTLR_EL1 if necessary. This register cannot be accessed from EL0, and any attempt to do so generates an exception.

In the general operation of the system, the privileged Exception levels control their own configuration. However, more privileged levels sometimes access registers associated with lower Exception levels. For example, this could be to implement virtualization features and context switching.

The following System registers are discussed in relation to the AArch64 Exception model in this guide:

Table 2-1: Key System registers when handling exceptions

Register	Name	Description
Exception Link Register	ELR_ELx	Holds the address of the instruction which caused the exception
Exception Syndrome Register	ESR_ELx	Includes information about the reasons for the exception
Fault Address Register	FAR_ELx	Holds the virtual faulting address
Hypervisor Configuration Register	HCR_ELx	Controls virtualization settings and trapping of exceptions to EL2
Secure Configuration Register	SCR_ELx	Controls Secure state and trapping of exceptions to EL3
System Control Register	SCTLR_ELx	Controls standard memory, system facilities, and provides status information for implemented functions
Saved Program Status Register	SPSR_ELx	Holds the saved processor state when an exception is taken to this ELx
Vector Base Address Register	VBAR_ELx	Holds the exception base address for any exception that is taken to ELx

3. Execution and Security states

The AArch64 architecture provides four Exception levels. There are also two Execution states and up to four Security states. The current state of an Armv8-A or Armv9-A processor is determined by the Exception level and the current Execution state.

3.1 Execution states

The current Execution state defines the standard width of the general-purpose register and the available instruction sets. The Execution state also affects aspects of the memory models and how exceptions are managed.

Armv8-A and Armv9-A support two Execution states:

AArch32

AArch32 is a 32-bit Execution state. Operation in this state is backward compatible with previous architectures. It supports the T32 and A32 instruction sets. The standard register width is 32 bits.

AArch64

AArch64 is a 64-bit Execution state. It supports the A64 instruction set. The standard register width is 64 bits.

The later chapters in this guide focus in more depth on how exceptions are handled in AArch64.

3.1.1 Changing Execution state

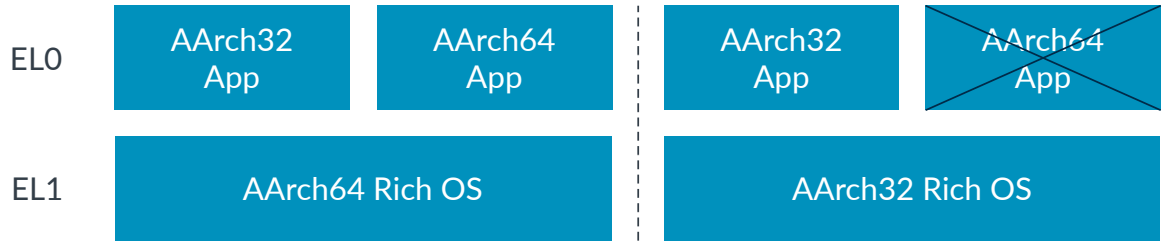
A Processing Element (PE) can only change Execution state on reset or when the Exception level changes. When the PE moves between Exception levels it is possible to change Execution state, however transitioning between AArch32 and AArch64 is only allowed subject to additional rules:

- When moving from a lower Exception level to a higher level, the Execution state can stay the same or change to AArch64.
- When moving from a higher Exception level to a lower level, the Execution state can stay the same or change to AArch32.

Putting these two rules together means that a 64-bit layer can host a 32-bit layer, but not the other way around. For example, a 64-bit OS kernel can host both 64-bit and 32-bit applications, while a 32-bit OS kernel can only host 32-bit applications.

The following diagram illustrates these scenarios:

Figure 3-1: Exception levels on 32- and 64-bit layers



In this example we have used an OS and applications, but the same rules apply to all Exception levels. For example, a 32-bit hypervisor at EL2 could only host a 32-bit OS at EL1.

The Armv8-A architecture supports AArch32 and AArch64 Execution state at all Exception levels. It is an implementation choice to support either or both for each Exception level.

For Armv8-A processors, the Execution state on reset is determined by an **IMPLEMENTATION DEFINED** mechanism. For example, Cortex-A32 always resets into AArch32 state.

For Armv9-A processors, AArch64 is required to be supported at all ELs. It is an implementation choice whether to support AArch32 at EL0. All other Exception levels are AArch64, and on reset the Execution state is always AArch64.

The Execution state of each Exception level is defined by the control register at the next higher implemented Exception level. This utilizes the Hypervisor Configuration Register (HCR_EL2) at EL2 and the Secure Configuration Register (SCR_EL3) at EL3. At reset, if switchable, EL3 execution is set by an external pin. This topic is covered further in the section on Routing and interrupt controllers.

3.2 Security states

AArch64 allows for the implementation of multiple Security states. This allows a further partitioning of software to isolate and compartmentalize trusted software.

Most Cortex-A processors support two Security states:

Secure state

In this state, a Processing Element (PE) can access both the Secure and Non-secure physical address spaces, and the Secure copy of banked registers.

Non-secure state

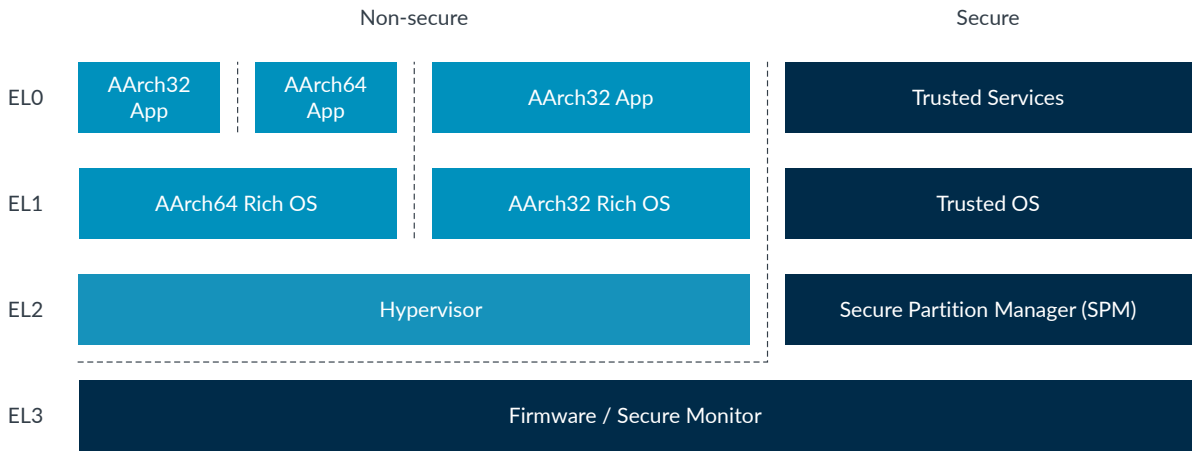
This is often also referred to as Normal world. In this state, a PE can only access the Non-secure physical address space. The PE can only access System registers that allow non-secure accesses.

The Security state defines which of the implemented Exception levels can be accessed, which areas of memory can currently be accessed, and how those accesses are represented on the system memory bus. If in Non-secure state, the PE can only access the Non-secure physical address space. In Secure state, the PE can access both the Secure and Non-secure physical address spaces.

An example of this would be to have your operating system, such as Android, running in the Normal world, with a payment or DRM system running in the Secure world. We need a higher degree of trust in the Secure world systems and need to keep them separate to protect information such as payment details and keys. Having the two security states provides this separation.

This diagram shows the Exception levels and Security states, with different Execution states being used:

Figure 3-2: Exception levels and Security states using different Execution states



The uses of these Security states are described in more detail in our guide TrustZone for AArch64.

3.2.1 Changing Security state

If TrustZone is implemented then the processor can be in either Secure state or Non-secure state. This is selected by the SCR_EL3.NS bit.

You may have noted in the previous diagram that EL3 is in Secure state. EL3 is the most privileged Exception level and the Security state of EL3 is fixed. EL3 is able to access all copies of a banked System register.

In Armv8-A EL3 is always in Secure state. In Armv9-A, EL3 is part of Secure state unless RME is implemented. If RME is implemented, Root state is a separation of EL3 from the rest of Secure state. RME is covered more in the next section.

Whenever you want to switch from one Security state to another you must pass through EL3. Software at EL3 is responsible for managing access to the different available Security states

and acts as a gatekeeper, controlling access to the Security states of EL2, EL1 and EL0. The SCR_EL3.NS bit enables a change of Security state on return from EL3.

At EL0, EL1, and EL2 the PE can be in either Secure state or Non-secure state. You often see this written as:

- NS_EL1: Non-secure state, Exception level 1
- S_EL1: Secure state, Exception level 1

Changing Security state is discussed in more detail in TrustZone for AArch64 and Realm Management Extension.

3.2.2 Realm Management Extension

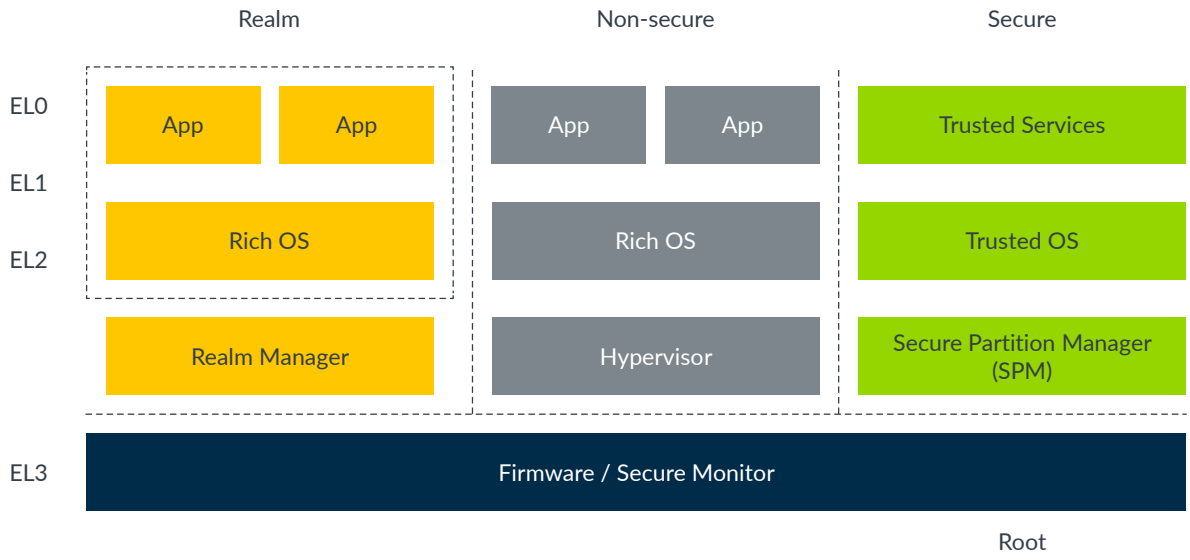
Armv9-A introduced support for the Realm Management Extension (RME). When RME is implemented, two additional Security states are supported:

- Realm state: In this state a PE can access Non-secure and Realm physical address spaces.
- Root state: In this state a PE can access all physical address spaces. Root state is only available in EL3.

RME isolates EL3 from all other Security states. With RME, EL3 moves out of Secure state and into its own Security state called Root. Exception level 3 hosts the platform and initial boot code and therefore must be trusted by the software in Secure, Non-secure, and Realm states.

The following diagram shows the Security states in an RME-enabled PE, and how these Security states map to Exception levels:

Figure 3-3: Security states in an RME-enabled PE



For more information on RME, see the Realm Management Extension.

3.3 Impact of implemented Exception levels

It is an implementation choice whether all Exception levels are implemented, and which Execution states are allowed for each implemented Exception level, for any specific processor.

EL0 and EL1 are the only Exception levels that must be implemented and are mandatory. EL2 and EL3 are optional.

Please note that if EL3 or EL2 have not been implemented the following should be considered:

- EL2 contains much of the virtualization functionality. Implementations that do not have EL2 do not have access to these features. For more on virtualization see the AArch64 virtualization guide.
- EL3 is the only level that can change Security state. If an implementation chooses not to implement EL3, that PE would only have access to a single Security state. The state you are permanently in is therefore **IMPLEMENTATION DEFINED**.

There are several software implementations that have been developed requiring platforms to support these Exception levels. For example, KVM Linux requires EL2, EL1, and EL0.

Note

In Armv8.0-A, EL2 only existed in Non-secure state because there was no virtualization support in Secure state. Armv8.4-A added S.EL2 as an optional feature with an enable bit (SCR_EL3.EEL2) to give backwards compatibility. In Armv9-A, if

you support EL2 you must support it in both Security states, however the disable bit is still present.

An Armv8-A processor implementation can choose which Execution states are valid for each Exception level and this is **IMPLEMENTATION DEFINED**. If AArch32 is allowed at an Exception level, it must be allowed at all lower Exception levels. For example, if EL3 allows AArch32, then it must be allowed at all lower Exception levels.

Many implementations allow all Executions states and all Exception levels, but there are existing implementations with limitations. For example:

- Cortex-A35 supports AArch32 and AArch64 at all Exception levels
- Cortex-A32 only supports AArch32
- Neoverse-N2 supports AArch32 at EL0. All other Exception levels are AArch64 only.

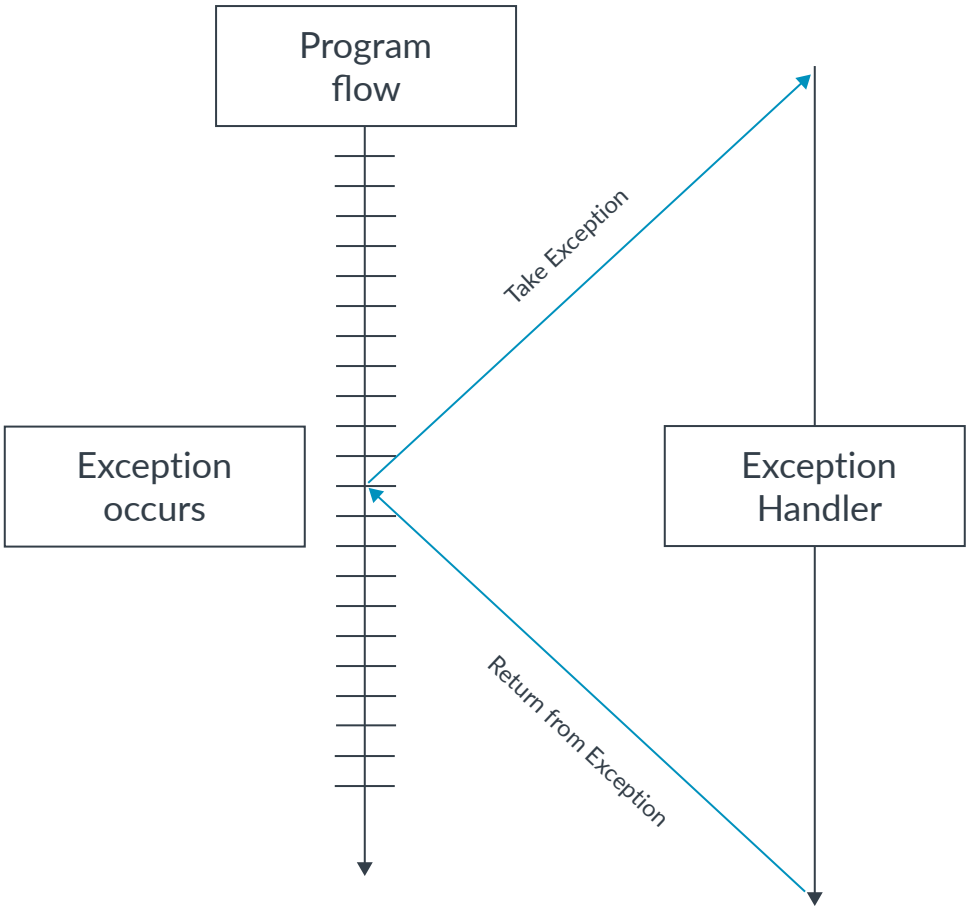
AArch32 provides backwards compatibility with the old 32-bit architecture, and from Armv9-A is restricted to optional implementation only at EL0. Armv9-A supports AArch64 at all Exception levels and only optionally supports AArch32 at EL0. This means that legacy applications can be run, but not kernels, hypervisors, or firmware.

You should refer to the Technical Reference Manual (TRM) of your processor to check what Exception levels are supported.

4. Exception types

Exceptions are conditions or system events that usually require remedial action or an update of system status by privileged software to ensure smooth functioning of the system. An exception is therefore any event that can cause the currently executing program to be suspended. Taking an exception causes a change in state to execute code to handle that exception.

Figure 4-1: Handling an exception



Other processor architectures might describe this as an interrupt. In AArch64, interrupts are a specific type of externally generated exception.

Exceptions are used for many different reasons, including the following:

- Emulating virtual devices
- Virtual memory management
- Handling software errors

- Handling hardware errors
- Debugging
- Performing calls to different privilege or security states
- Handling interrupts (timers, device interactions)
- Handling across different Execution states (known as interprocessing)

When an exception is taken, instead of moving to the next instruction in the current code, the processor stops current execution and branches to a piece of code to deal with the request. This code is known as an exception handler. Once the event is dealt with, the execution can return to the original program. Each exception type has its own exception handler. This is shown in Figure 4-1: Handling an exception on page 18.

The Arm architecture categorizes exceptions into two broad types: synchronous exceptions and asynchronous exceptions.

4.1 Synchronous exceptions

Synchronous exceptions are exceptions that can be caused by, or are related to, the instruction that is currently being executed. Synchronous exceptions are synchronous to the execution stream, as they are directly related to the currently executing instruction. For example a synchronous exception would be triggered by an instruction attempting to write to a read-only location as defined by the MMU.

If the following all apply, then an exception is synchronous:

- The exception has been generated as a result of direct, or attempted, execution of an instruction.
- The address to return to once the exception has been handled (return address) has an architecturally-defined relationship with the instruction that caused the exception.
- The exception is precise, meaning the register state presented on entry to the exception handler is consistent with every instruction before the offending instruction having been executed, and none after it.

There are many different types of synchronous exception, and it is possible that a given instruction might cause multiple synchronous exceptions. The Arm architecture provides a fixed priority order for synchronous exceptions.

The following sections discuss some of the different causes of synchronous exceptions in more depth:

- Invalid instructions and trap exceptions
- Memory accesses
- Exception-generating instructions
- Debug exceptions

4.1.1 Invalid instructions and trap exceptions

Synchronous exceptions can be caused by attempting to execute an invalid instruction. There are many reasons for invalid instructions, including instructions that are **UNDEFINED**, not allowed at the current Exception level, or that have been disabled. Any attempt to execute an instruction that is not recognized by the core generates an **UNDEFINED** exception.

The architecture also allows a controlling entity, such as an OS or Hypervisor, to set up traps to intercept operations at lower Exception levels. A trap triggers an exception when a given action, such as reading a specific register, is performed.

For example, an OS kernel at EL1 might disable the use of floating-point instructions at EL0, to save time when context-switching between applications. This is known as lazy context-switching ; the number of registers pushed to the stack can be reduced if, for example, SIMD or Floating Point (FP) units have not been used prior to the context switch. A trap exception can then be used to handle edge cases.

In this case, the OS kernel can monitor the state of the SIMD/FP operation by disabling the SIMD/FP unit. When an FP or SIMD instruction is executed, a trap exception is taken to the OS kernel at EL1. The kernel can then enable the SIMD/FP unit, execute the failed instruction and set a flag to state that the SIMD/FP unit has been used. This ensures that the large SIMD/FP register file is included in the register context at the next context switch. If there is no flag asserted at the next context switch then the SIMD/FP registers do not need to be included.

The capability to trap exceptions is particularly important for virtualization. For more on how exceptions are used for virtualization, see the AArch64 virtualization guide.

4.1.2 Memory accesses

Synchronous exceptions can also be caused by memory accesses. This could be as a result of checks performed by the MMU or due to errors returned by the memory system.

For example, while the MMU is enabled all accesses to memory caused by load and store instructions are checked. If you try to access a privileged address from unprivileged code, or try to write to a read-only address, then the MMU blocks the access and triggers a Memory Management Unit (MMU) fault. Because MMU-generated errors are synchronous, the exception is taken before the memory access proceeds.

Memory access errors are discussed in more detail in the AArch64 memory management guide. Memory accesses can also generate asynchronous exceptions, which are covered in the SError section later in this guide.


Note

In AArch64, synchronous aborts cause a synchronous exception. Asynchronous aborts cause an SError interrupt exception.

4.1.3 Exception-generating instructions

There are instructions that intentionally cause an exception to be generated and taken. These instructions are used to implement system call interfaces to allow less privileged software to request services from more privileged software. These are sometimes called system calls and are often used in software-based APIs.

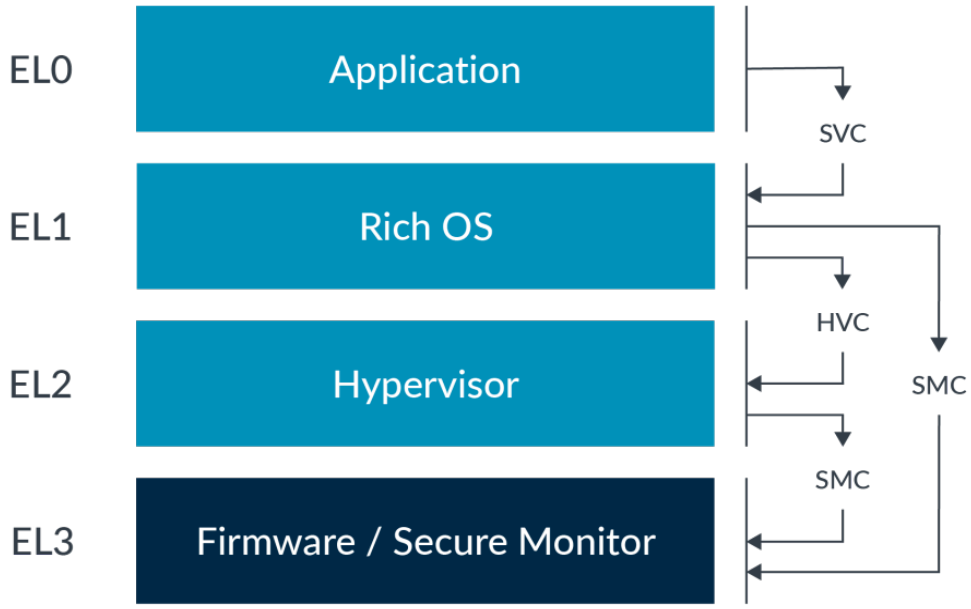
The Arm architecture includes the exception-generating instructions `SVC`, `HVC`, and `SMC`. The purpose of these instructions is solely to generate an exception and enable the PE to move between Exception levels:

- The Supervisor Call (`SVC`) instruction enables a user program at EL0 to request an OS service at EL1
- The Hypervisor Call (`HVC`) instruction, available if the Virtualization Extensions are implemented, enables the OS to request hypervisor services at EL2
- The Secure Monitor Call (`SMC`) instruction, available if the Security Extensions are implemented, enables the Normal world to request Secure world services from firmware at EL3

When the PE is executing at EL0, it cannot call directly to a hypervisor at EL2 or secure monitor at EL3, as this is only possible from EL1 and higher. The application at EL0 must use an SVC call to the kernel, and have the kernel perform the action to call into higher Exception levels.

Assuming the respective Exception levels have been implemented, the OS kernel (EL1) can execute an `HVC` instruction to call the hypervisor at EL2 or call the secure monitor at EL3 with the `SMC` instruction. Similarly, from EL2 the PE can use the `SMC` instruction to call the EL3 secure monitor. This is illustrated in the following diagram:

Figure 4-2: Service call routing



Because exceptions cannot be taken to lower ELs, an svc call made at EL2 cannot cause entry back to EL1.

Note

As discussed previously in relation to Invalid instructions and trap exceptions, a hypervisor might be presenting an emulated view of the system to EL1. In this instance a guest OS would not be able to directly invoke the device firmware using a Secure Monitor Call. Instead such calls would be trapped to EL2, by setting the HCR_EL2.TSC bit in the Hypervisor Configuration Register.

4.1.4 Debug exceptions

Debug exceptions are synchronous exceptions that are routed to the Exception level where a debugger is hosted. The debugger code then executes much like exception handler code.

There are a number of specific debug exceptions that are synchronous, including:

- Breakpoint Instruction exceptions
- Breakpoint exceptions
- Watchpoint exceptions
- Vector Catch exceptions
- Software Step exceptions

For more on debug exceptions and how they are handled, see the AArch64 self-hosted debug guide.

4.2 Asynchronous exceptions

Some types of exceptions are generated externally and therefore are not synchronous with the current instruction stream.

Asynchronous exceptions are not directly associated with the current executing instructions, and are typically system events from outside the processor. This could be system events to which the software needs to respond, such as the activity of a timer or the touch of a screen. We do not know when they will occur.

By definition, an exception is asynchronous if it is not synchronous. Asynchronous exceptions are also known as interrupts.

On taking an asynchronous exception the program flow is interrupted and passes to code to specifically handle this external request. It is not possible to guarantee exactly when an asynchronous exception will be taken, and the AArch64 architecture requires only for it to happen in a finite time.

4.2.1 Physical interrupts

Physical interrupts are those generated in response to a signal from outside the PE, typically by peripherals. Rather than the core constantly polling for external signals, the system informs the core that something has to happen by generating an interrupt.

For example, a system might use a Universal Asynchronous Receiver/Transmitter (UART) interface to communicate with the outside world. When the UART receives data, it needs a mechanism to be able to tell the processor that the new data has arrived and is ready to be processed. One mechanism that the UART could use is to generate an interrupt to signal to the processor.

Complex systems can have many interrupt sources with different levels of priority, including the ability for nested interrupt handling in which a higher priority interrupt can interrupt a lower priority one. The speed that a core can respond to such events might be a critical issue in the system design, and is called interrupt latency.

Next we will look at the different types of physical interrupts.

4.2.2 SError

System Error (SError) is an exception type that is intended to be generated by the memory system in response to unexpected events. We do not expect these events, but need to know if they have

occurred. These are reported asynchronously because the instruction that triggered the event may have already been retired.

A typical example of SError is what was previously referred to as an external asynchronous abort. Examples of SError interrupts include:

- Memory access which has passed all the MMU checks but then encounters an error on the memory bus
- Parity or Error Correction Code (ECC) checking on some RAMs, for example those in built-in caches
- An abort triggered by write-back of dirty data from a cache line to external memory

SErrors are treated as a separate class of asynchronous exception because you would normally have separate handlers for these cases. SError generation is **IMPLEMENTATION DEFINED**.

4.2.3 IRQ and FIQ

The Arm architecture has two asynchronous exception types, IRQ and FIQ, that are intended to be used to support handling of peripheral interrupts. These are used to signal external events, such as a timer going off, and do not represent a system error. They are expected events which are asynchronous to the processor's instruction stream.

IRQ and FIQ have independent routing controls and are often used to implement Secure and Non-secure interrupts, as discussed in the Arm Generic Interrupt Controller v3 and v4 guide. It is **IMPLEMENTATION DEFINED** how the two exception types are used.

In older versions of the Arm architecture, FIQ was used as a higher priority fast interrupt. This is different from AArch64, where FIQ has the same priority as IRQ.

In almost all cases, interrupt controllers are paired with AArch64 processors in a system for collating, prioritizing, and processing all of the interrupts. All Arm implementations use the Arm Generic Interrupt Controller (GIC) architecture for the management of IRQs and FIQs. The GIC performs the tasks of interrupt management, prioritization, and routing, providing a single signal per physical interrupt type into the core. For more information on the GIC architecture see the Arm Generic Interrupt Controller v3 and v4 guide)

4.2.4 Virtual interrupts

A system that uses virtualization has more complex needs for interrupt handling. Some interrupts may be handed by the hypervisor and some may be handled within a VM. The interrupts seen by the VM are virtual interrupts. Virtual interrupts can be externally generated by a device connected

to an interrupt controller or may be generated by software. Additional mechanisms to support this are therefore required, so in AArch64 there is explicit support for virtual interrupts.

- vSError, Virtual System Error
- vIRQ, Virtual IRQ
- vFIQ, Virtual FIQ

Virtual interrupts are controlled per interrupt type. These virtual interrupts function the same as their physical counterparts, however they can only be signaled to EL1.

Virtual interrupts can be generated either from a hypervisor at EL2 or by using an interrupt controller. The hypervisor must set the corresponding routing bit in the Hypervisor Configuration Register (HCR_EL2). For example, to enable vIRQ signaling, a hypervisor must set HCR_EL2.IMO. This setting routes physical IRQ exceptions to EL2, and enables signaling of the virtual exception to EL1.

There are three bits in HCR_EL2 that control virtual interrupt generation:

VSE
: Setting this bit registers a vSError.

VI
: Setting this bit registers a vIRQ.

VF
: Setting this bit registers a vFIQ.

Setting one of these bits is equivalent to an interrupt controller asserting an interrupt signal into a vCPU. An implication of this method is that the hypervisor is then required to emulate the operation of the interrupt controller in the VM. This can cause significant overheads when used for frequent operations, so use of an interrupt controller is advised.

GICv2 and later supports signaling of both physical and virtual interrupts, by providing a physical CPU interface and a virtual CPU interface. For more information on interrupt controllers see Arm Generic Interrupt Controller v3 and v4. Support for virtualization in Secure state was added in Armv8.4-A, and requires Secure EL2 to be enabled and supported.

Virtual interrupts are discussed further in the AArch64 virtualization guide.

4.2.5 Masking

Both physical and virtual asynchronous exceptions can be temporarily masked. This means that asynchronous exceptions can remain in a pending state until they are unmasked and the exception is taken. This is particularly useful for dealing with nested exceptions.

Synchronous exceptions cannot be masked. This is because synchronous exceptions are caused directly by the execution of an instruction so would block execution if they were then left pending or ignored.

The 2021 extensions, Armv8.8-A and Armv9.3-A, added Non-maskable interrupt (NMI) support. When supported and enabled, an interrupt can be presented to the processor as having Superpriority through this capability. Superpriority allows the interrupt to be taken when interrupts without Superpriority would have been masked.

Masking is covered further in Routing and interrupt controllers later in this guide. Masking and the NMI capability is discussed in more depth in the section on Exception masking and non-maskable interrupts (NMI).

5. Handling exceptions

As we covered in the chapter on Exception types, when an exception occurs the current program flow is interrupted. This chapter talks in more depth about how exceptions are handled in practice.

In AArch64 specific terminology is used when talking about taking an exception:

- When the PE responds to an exception, an exception is taken.
- The PE state immediately before taking the exception is the state the exception is *taken from*.
- The PE state immediately after taking the exception is the state the exception is *taken to*.

Therefore the state that the processor is in when the exception is recognized is known as the state the exception is taken from. The state the PE is in immediately after the exception is the state the exception is taken to. For example, it is possible to take an exception from AArch32 EL0 to AArch64 EL1.

After an exception has been handled, the system needs to return from the state it has been taken to. This is known as an Exception return, and the Arm architecture has instructions that trigger an Exception return:

- The state that the PE is in at the point the return instruction is executed is the state that the exception *returns from*.
- The PE state immediately after the exception return instruction has executed is the state that the exception *returns to*.

5.1 Taking an exception

When an exception occurs, the processor saves the current status of the PE alongside the exception return address, and then enters a specific mode to handle the exception.

A snapshot of current state is taken from PSTATE, discussed further in the section on Saving the current processor state. This snapshot is written to the Saved Program Status Register (SPSR) and the return address is written to an Exception Link Register (ELR). For synchronous exceptions and SErrors another register, the Exception Syndrome Register (ESR), is also updated. This records the cause of the exception.

When an exception is taken to an Exception level (ELx) that is using AArch64 state, all of the following occur:

- The contents of PSTATE immediately before the exception was taken is written to SPSR_ELx.
- The preferred exception return address is written to ELR_ELx.

In addition:

- For synchronous exceptions and SError interrupts, exception syndrome information (the cause of the exception) is written to ESR_ELx.

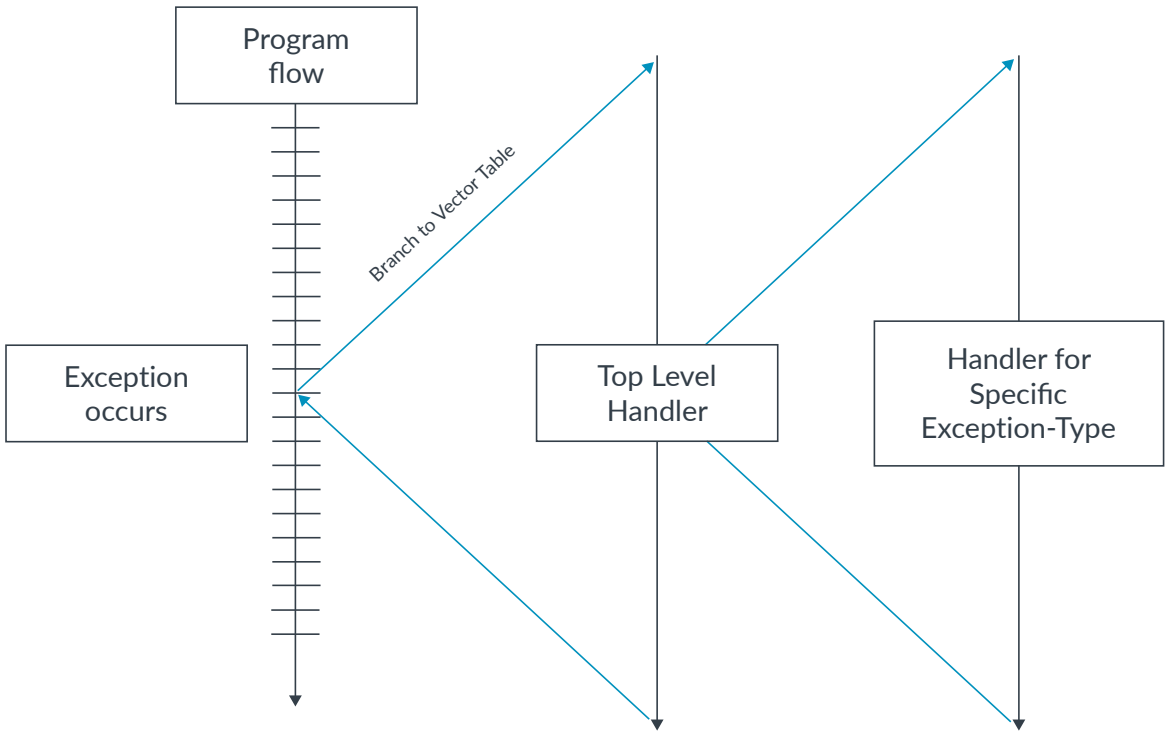
- For address-related synchronous exceptions, such as MMU faults, the virtual address that triggered the exception is written to the Fault Address Register, FAR_ELx.

Exception handling for any given exception starts from a fixed memory address called an exception vector. When an exception occurs the Processing Element (PE) branches to a location in a vector table.

Vector tables in AArch64 are different to many other processor architectures as they contain instructions, not addresses. Each entry contains up to 32 instructions ; just enough to perform basic stacking and call exception-specific handling code.

The vector table location is normally configured to contain the handler code to perform generic actions and to branch to further exception handling code according to the exception type as illustrated in Figure 5-1: Top-level handler on page 28. This vector code is limited to 32 words of code. An exception handler contains the code to address the requested operation and enables the return from the exception state.

Figure 5-1: Top-level handler



Each exception type targets an Exception level (EL) to which an exception is taken. Taking an exception therefore enables routing to a different EL. This is particularly important as the only way

to gain privilege is by taking an exception. The only way to lose or reduce privilege is by performing an exception return.

This means:

- On taking an exception, the EL can stay the same or increase.
- On an exception return, the EL can stay the same or decrease.

It is important to note that taking an exception, or performing an exception return, does not have to entail a change of EL. The target of an exception may be the same as the current EL. The target EL is defined either implicitly according to exception type or by configuration bits within System registers.


Note

In AArch64, you can take exceptions from EL0 to a higher Exception level only. Exceptions are never taken to EL0 and there is no EL0 vector table.

As discussed in the chapter on Execution and Security states, the PE can also only change Execution state at reset or when taking or returning from an exception. The interaction between AArch32 and AArch64 Execution states is called interprocessing. It is important to remember the following in relation to Changing Execution state:

- When moving from a lower Exception level to a higher level, the Execution state can stay the same or change to AArch64.
- When moving from a higher Exception level to a lower level, the Execution state can stay the same or change to AArch32.

The following is defined by the architecture:

- If an Exception level is using AArch32, then all lower Exception levels must use AArch32
- If an Exception level is using AArch64, then all higher Exception levels must use AArch64

In the Armv9-A architecture and some v8-A implementations, AArch32 is only supported at EL0 and exceptions cannot be taken to EL0. This means that to change the Execution state of EL0 requires moving to a more privileged EL, and then returning back again.

As exceptions can be taken from AArch32 to AArch64, AArch64 handler code may need to access AArch32 registers. AArch32 general-purpose registers are directly mapped to the AArch64 registers to allow handler code to access AArch32 registers:

Table 5-1: AArch32 to AArch64 register mappings

AArch32	AArch64
R0-R12	X0-X12
Banked SP and LR	X13-23
Banked FIQ	X24-30

When moving from AArch32 to AArch64, registers not accessible in AArch32 state retain their values from previous AArch64 execution. For registers that are accessible in both execution states, the top half of the 64-bit registers either contain 0 or the old value on taking an exception from an Exception level using AArch32:

- Top 32 bits: **UNKNOWN**
- Bottom 32 bits: The value of the mapped AArch32 register

Registers in AArch64 are discussed in more depth in the AArch64 Instruction Set Architecture guide.

The following sections discuss each step in the process from an exception being taken, through to completing the Exception return.

5.1.1 Saving the current processor state

AArch64 has a concept of processor state known as PSTATE, it is this information that is stored in the SPSR. PSTATE contains things like current Exception level and Arithmetical Logical Unit (ALU) flags. In AArch64, this includes:

- Condition flags
- Execution state controls
- Exception mask bits
- Access control bits
- Timing control bits
- Speculation control bits

For example, the following Exception mask bits (DAIF) in PSTATE allow exception events to be masked. The exception is not taken when the associated bit is set.

- D - Debug exception mask bit
- A - SError asynchronous exception mask bit, for example, asynchronous external abort
- I - IRQ asynchronous exception mask bit
- F - FIQ asynchronous exception mask bit

When an exception is taken, the current state must be preserved so that it can be returned to the correct state later. The PE automatically preserves the exception return address and the current PSTATE. As mentioned in Taking an exception, PSTATE is stored as a snapshot in the SPSR (Saved Program Status Register).

There is an SPSR per Exception level, SPSR_ELx. On taking an exception, the SPSR_ELx used is for the Exception level the exception is *taken to*. For example, if the exception is taken to EL1 then `SPSR_EL1` is updated.

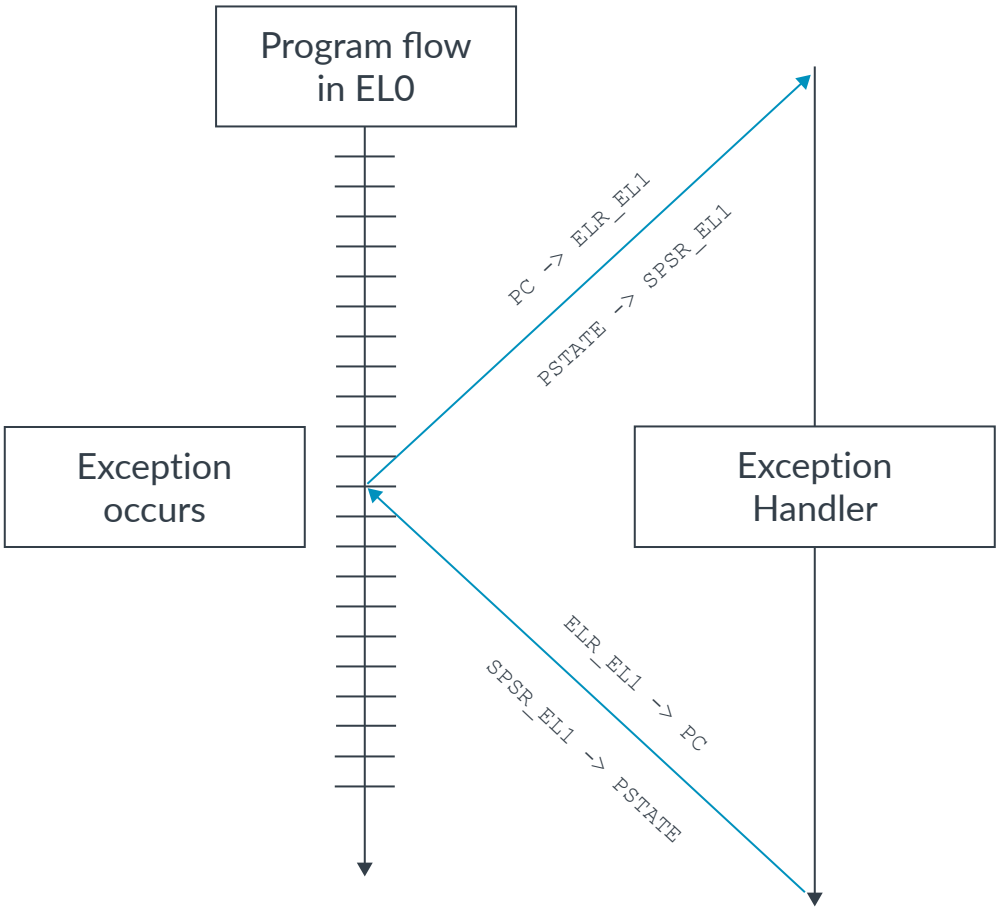
The PE then updates the current PSTATE to the one defined in the architecture for that exception type, reflecting the new state. This includes updating the target Exception level and Security level where impacted.

Note

In Armv7 and earlier, PSTATE was called CPSR and was implemented as a register.

Once PSTATE is updated, the PE can branch to the exception handler in the vector table. Execution starts from the exception vector at the target Exception level as defined by the exception type. To return from an exception the processor can then restore the contents of the SPSR to PSTATE and branch to the return address specified in the ELR. This is shown in Figure 5-2: Saving and restoring processor state on page 31.

Figure 5-2: Saving and restoring processor state



5.1.2 Routing and interrupt controllers

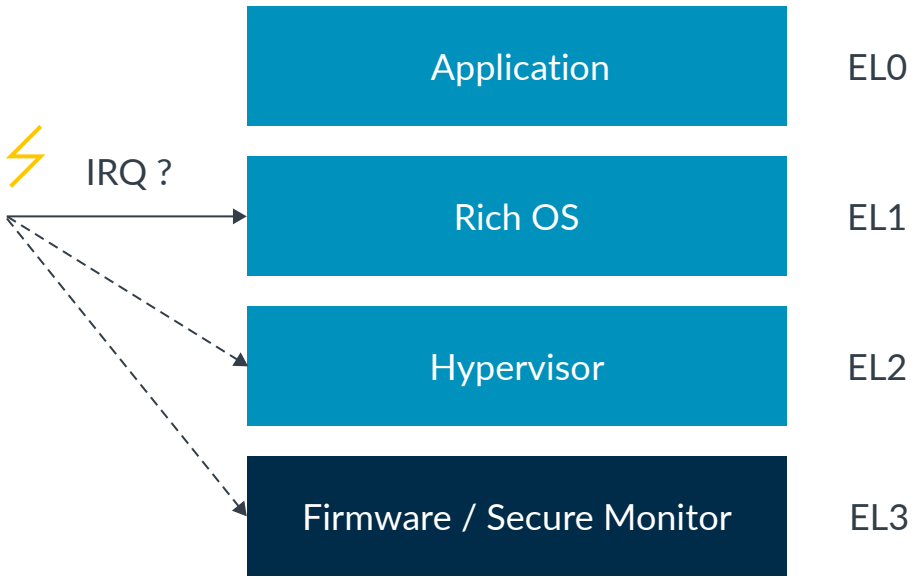
Each exception type has a target Exception level that is either:

- Implicit according to the type of the exception
- Defined by configuration bits in the System registers

The target of an exception is either fixed by the architecture, or configured by software using routing controls. However, exceptions can never be taken to EL0.

Synchronous exceptions are routed according to the rules associated with the exception-generating instructions `SVC`, `HVC`, and `SMC`. When implemented, other classes of exception can be routed to EL2 (Hypervisor) or EL3 (Secure Monitor). Routing is set independently for IRQs, FIQs, and SErrors. For example, an implementation could route all IRQs to EL1, as shown in the following diagram:

Figure 5-3: IRQ routing example



Routing is configured using the Secure Configuration Register SCR_EL3 and the Hypervisor Configuration Register HCR_EL2. The SCR_EL3 register specifies which exceptions are routed to EL3, while the HCR_EL2 register similarly specifies which exceptions are routed to EL2.

These allow different interrupt types to be routed to different Exception levels. For example, IRQs might be handled by the OS at EL1 whereas SErrors would more typically be handled by the firmware running at EL3.

There are separate bits in each controlling register that allow individual control over IRQ, FIQ and SError interrupts. In addition, routing configurations made using SCR_EL3 override routing

configurations made using HCR_EL2 where they conflict. The routing bits in these registers have an **UNKNOWN** value at reset, so they must be initialized by software.

It is common for the Arm Generic Interrupt Controller (GIC) architecture (see the associated guide) to be used to perform the task of interrupt management, prioritization, and routing. This can reduce overheads associated with virtualization.


Note

An exception cannot be taken to an unimplemented Exception level. Calls to an unimplemented Exception level are UNDEFINED. Similarly, trying to return to EL2 when it is disabled, or not implemented, for a given Security state result in a fault being generated on the exception return (ERET).

It was previously mentioned in the section on Masking, that asynchronous exceptions can be temporarily masked and left in a pending state until the exception is unmasked and taken. Routing also impacts masking, as the ability to mask depends on the current and target Exception levels.

Exceptions routed to a higher Exception level cannot be masked by the lower EL. For example, if interrupts are masked in EL1 and an interrupt is routed to EL2 then the EL1 mask will not affect the EL2 operation. Please note, however, that EL2 interrupts may have been masked when the PE execution last exited from EL2 which could still lead to the interrupt being masked on entry to EL2.

Exceptions routed to the current Exception level can be masked by the current level. Exceptions routed to a lower Exception level are always masked. The exception is pended until the PE changes to an Exception level equal to, or lower than, the one routed to. This fits with the rule that you can never lose privilege by taking an exception.

Table 5-2: Masking rules on Exception level change

Exception level change	Impact on Masking
Target EL > Current EL	Not maskable by CurrentEL
Target EL < Current EL	Implicitly masked
Target EL == Current EL	Set by PSTATE mask bits

The Execution state of an Exception level that an exception is taken to is determined by a higher Exception level. Assuming all Exception levels are implemented the following table shows how the Execution state is determined:

Table 5-3: Execution state determination according to target Exception level

Exception level taken to	Execution state determined by
EL1	HCR_EL2.RW
EL2	SCR_EL3.RW
EL3	Reset state of EL3

5.1.3 AArch64 vector tables

When taking an exception to an Exception level using AArch64, vector tables are an area of normal memory containing instructions that are then used to handle the exception.

When an exception occurs, the core needs to be able to execute the handler corresponding to that exception. The handler acts as dispatch code, identifying the cause of the exception and then calling the relevant handler code to deal with the exception. The location in memory where the handler is stored is called the exception vector. In AArch64, exception vectors are stored in exception vector tables.

Each Exception level has its own vector table, with the base address defined by its own Vector Base Address Register, VBAR_EL<x>, where <x> is 1,2, or 3. Note that there is no dedicated vector table for EL0 as exceptions are never taken to EL0.

All vector tables use the same format. There are different exception category entries based on the type of exception and where the exception was taken from. Each exception category has an exception vector at a fixed offset from the vector base address.

The entry used depends on the following factors:

- The type of exception (SError, FIQ, IRQ, Synchronous)
- The Exception levels the exception is being taken from and to
- The Execution states that need to be supported
- The stack pointer being used (see section on Stack pointer selection and stack pointer registers)

The values of the VBAR registers are undefined after reset, so they must be configured before interrupts are enabled.

The following describes the fixed offsets from the vector base address within a vector table:

Table 5-4: Vector table offsets

Address	Exception type	Description
VBAR_ELx + 0x780	SError/VSError	Exception from a lower EL and all lower ELs are AArch32
VBAR_ELx + 0x700	FIQ/vFIQ	
VBAR_ELx + 0x680	IRQ/vIRQ	
VBAR_ELx + 0x600	Synchronous	
VBAR_ELx + 0x580	SError/VSError	Exception from a lower EL and at least one lower EL is AArch64
VBAR_ELx + 0x500	FIQ/vFIQ	
VBAR_ELx + 0x480	IRQ/vIRQ	
VBAR_ELx + 0x400	Synchronous	
VBAR_ELx + 0x380	SError/VSError	Exception from the current EL while using SP_ELx
VBAR_ELx + 0x300	FIQ/vFIQ	

Address	Exception type	Description
VBAR_ELx + 0x280	IRQ/vIRQ	
VBAR_ELx + 0x200	Synchronous	
VBAR_ELx + 0x180	SError/VSError	Exception from the current EL while using SP_EL0
VBAR_ELx + 0x100	FIQ/vFIQ	
VBAR_ELx + 0x080	IRQ/vIRQ	
VBAR_ELx + 0x000	Synchronous	

This is recorded into the system register by privileged software so that the core can locate the respective handler when an exception occurs. The vectors are divided into two groups, with each group split into two sub-groups:

- Exception from Lower EL
  - Lower EL using AArch32
  - Lower EL using AArch64
- Exception from the current EL
  - Exception when SP_ELx selected
  - Exception when SP_EL0 selected

Therefore which vector the exception arrives at already provides information about the state of the processor at the time of the exception. We will cover the significance of the selected stack point in the section on Stack pointer selection and stack pointer registers.

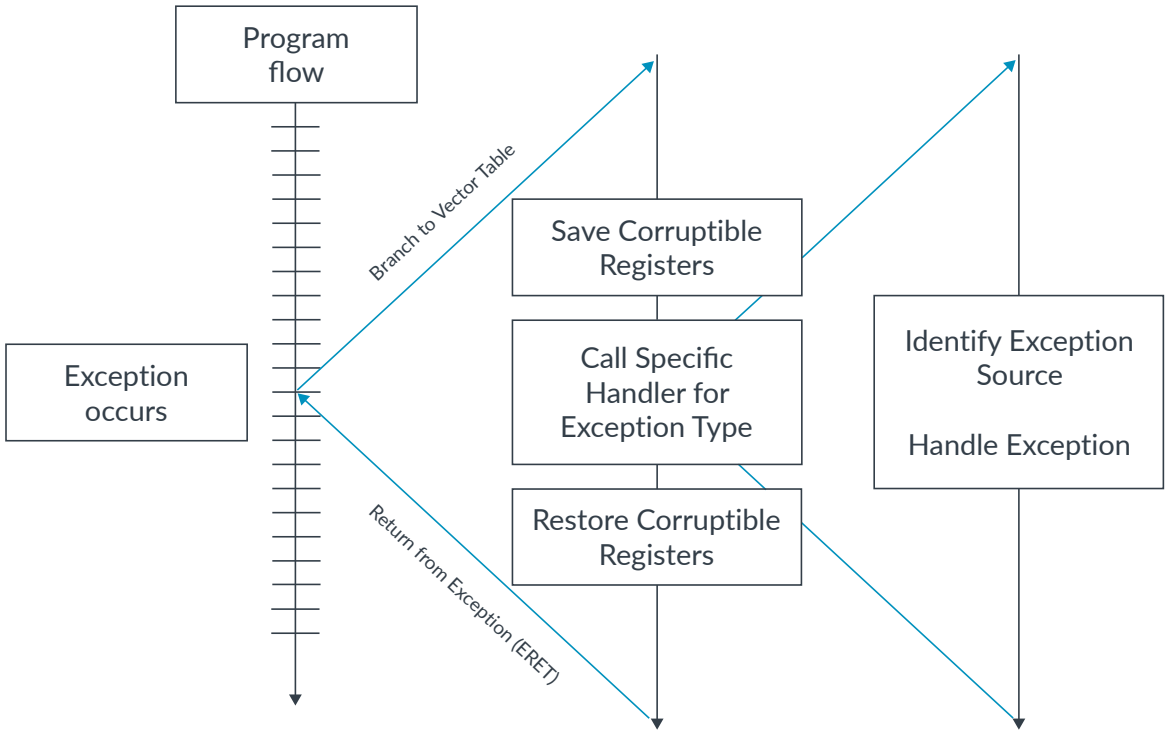
Note that the Execution state reported from the lower EL is for the EL immediately below where the exception was taken to and not necessarily the EL where the exception was taken from.

For example, if an exception is taken from EL0 to EL1, the vector table entry is selected based on the Execution state of EL1. However if an exception is taken from EL0 to EL2, the vector used depends on EL1, not EL0. This is because typically we would expect a hypervisor, or other virtualization management software, to be running at EL2 managing virtual machines (VMs) running at EL0 and EL1. The hypervisor needs to know about the Execution state of OS within the VM, not typically the applications hosted by the OS. There are some special cases, such as when HCR_EL2.TGE is set to 1. This topic is covered in more detail in the AArch64 virtualization guide.

A simple exception handler would instruct the system to stack all corruptible registers, this may be all registers depending on the type of exception, and then call exception-specific handler code. On return, these registers would be restored and an exception return instruction (ERET) called.

The 32 words of space within the vector table is enough to contain the instructions for stacking, calling exception-specific handler code, restoring registers, and initiating the ERET return. This flow is summarized in Figure 5-4: Simple exception handler example on page 36.

Figure 5-4: Simple exception handler example



5.1.4 Stack pointer selection and stack pointer registers

When executing in AArch64, the architecture allows a choice of two stack pointer registers: SP_EL0 or SP_EL<x>, where <x> is the current Exception level. For example, at EL1 it is possible to select SP_EL0 or SP_EL1. This is visible in the vector table example provided in the section on AArch64 vector tables.

When an exception is taken, the stack pointer for the target Exception level SP_ELx is automatically selected. The stack pointer selected by the PE is configured using PSTATE according to the following rules:

- If executing at EL0, then the PE uses the EL0 stack pointer, SP_EL0.
- If executing at EL1, EL2, or EL3, then the PE uses the stack pointer determined by the PSTATE.SP bit:
  - If PSTATE.SP is 0, then the PE uses the EL0 stack pointer, SP_EL0.
  - If PSTATE.SP is 1, then the PE uses the stack pointer ELx of the current Exception level.

As we enter the first-level handler at the vector table, we are using SP_ELx. This should by default be used to save the register context. However, we typically then switch to use SP_EL0 instead for any further processing.

The ELx stack can, for example, be used to store registers that could otherwise be impacted or corrupted by the exception handler, so they can be restored on returning from an exception. This is particularly useful for maintaining a valid stack when handling exceptions caused by stack overflows. The return operation would need to perform this action in reverse.

For example, if the PE is executing at EL1 and an IRQ interrupt is signaled, an IRQ exception is triggered. If IRQs have been configured to route to EL1, it would use SP_EL1 unless configured differently by PSTATE.SP. In the example vector table provided above, execution would take place from address VBAR_EL1 + `0x280`.

This separation means that we can isolate the stack used for exception entry and high-level exception handling from that used for other threads, for example, C code with high stack usage. SP_EL0 typically has a greater stack space allocated, so is always the better stack to use for this case. This also means that exception entry is not dependent on the state of the main stack. There are separate vector table entries that support this change as seen in Table 5-4: Vector table offsets on page 34.

As mentioned in the section on AArch64 vector tables, the vector the exception arrives at already provides information about the state of the processor at the time of the exception. For example, as exceptions should be masked during exception entry and exit, the code in the area of the vector table “Exception from the current EL while using SP_ELx” would only be triggered under exceptional circumstances. An asynchronous exception within this vector grouping suggests a significant system error has occurred and needs to be handled, for example a kernel panic.

5.2 Returning from an exception

Once the exception handler has finished dealing with the exception, the handler returns to the code that was running before the exception happened.

It does this by doing the following:

1. Restoring all previously stacked corruptible registers
2. Initiating an exception return instruction (`ERET`)

The `ERET` instruction restores the previous processor state from the associated SPSR and branches to the exception return address recorded in the ELR. The *return to* Exception level is configured based on the value in SPSR_EL<x>, where <x> is the level being returned from. SPSR_ELx also contains the target Execution state.


Note

The Execution state specified in SPSR_ELx must match the configuration in either SCR_EL3.RW or HCR_EL2.RW, or this generates an illegal exception return.

On execution of the ERET instruction, PSTATE is restored from SPSR_ELx, and the program counter is updated to the value in ELR_ELx. These two updates are performed atomically and indivisibly so that the PE is not left in an undefined state.

As introduced in the section on Saving the current processor state, the Exception Link Registers (ELR) at each EL hold the preferred exception return addresses. This is determined by the type of exception.

For synchronous service calls, such as svc, this is the instruction immediately following the exception call. For other synchronous exceptions this is the address of the instruction that generated the exception. For asynchronous exceptions, the preferred exception return address is the first instruction that had not been executed fully when the exception was taken. The handler is permitted to modify the contents of ELR_ELx, where needed, according to the exception type.

5.3 Exception handling examples

This section provides brief examples of synchronous and asynchronous exception handling from an end-to-end perspective.

5.3.1 Synchronous exception handling

As discussed in the section on Synchronous exceptions, an exception can be considered synchronous if it is generated by direct execution of instructions and the return address indicates the instruction that caused it. The following registers supply key information for handling of asynchronous exceptions, as well as SErrors:

- The Exception Syndrome Register (ESR_ELx) provides information about the cause and type of synchronous exception.
- The Fault Address Register (FAR_ELx) holds the faulting virtual address for address-related synchronous exceptions, such as MMU faults.
- The Exception Link Register (ELR_ELx) holds the address of the triggering instruction, providing the return address for the exception.

Here is a simple example of the AArch64 Exception model using system calls.

If code running at a lower Exception level has to perform a privileged operation it needs to make a call to a higher Exception level. For example, AArch32 application code running at EL0 may need to request heap memory allocation from the AArch64 OS/kernel at EL1. To do this, it generates a supervisor system call by executing an svc instruction, triggering the following:

- The current PSTATE is preserved as a snapshot in SPSR_EL1
- The preferred exception return address (the following instruction) is written to ELR_EL1
- Exception syndrome information (cause of the exception) is written to ESR_EL1
- The target Execution state is determined by reading the RW bit of the Hypervisor Configuration Register HCR_EL2.RW

- The current PSTATE is updated, with Exception level changed to EL1 and Execution state to AArch64
- The core branches to the vector table pointed to by VBAR_EL1 at offset to `0x600` (VBAR_EL1 + `600`) because this is a synchronous exception where the exception is from a lower EL and all lower ELs are AArch32
- Defined registers are stacked to maintain register context
- The type of asynchronous exception is identified from ESR_EL1, in this case SVC
- Specific SVC handler code is then executed
- Once the SVC-specific handler code completes, control returns to the high-level handler
- The handler restores all previously stacked registers and executes an `ERET` instruction
- PSTATE is restored from SPSR_EL1 (including the return to Exception level EL0 and target Execution state AArch32) and the program counter is updated to the value contained in ELR_EL1

Note that in this example AArch64 handler code may require access to AArch32 registers.

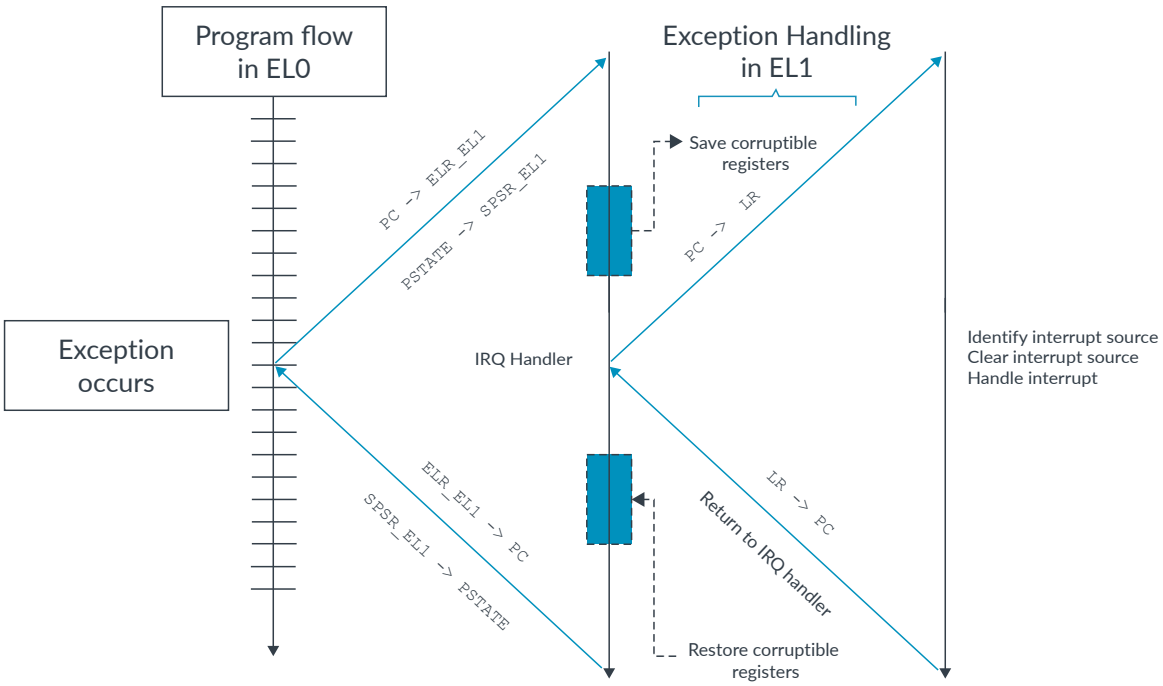
A more complex example involving system calls, would be where AArch64 application code running in a Non-secure (NS) state at EL0 needs to call to a secure, trusted OS running at EL1. To switch Security state you must pass through EL3. However, as covered in the section on Exception-generating instructions, EL0 is not able to directly initiate an SMC call to EL3. In this example, an `svc` instruction would be initiated from NS.EL0 to NS.EL1. NS.EL1 would then need to handle the change of security state by making an SMC call to EL3 to transition to S.EL1. This is covered in the TrustZone for AArch64 guide.

5.3.2 Asynchronous exception handling

Asynchronous exceptions, known as interrupts, are external to the PE and the currently executing instructions. The Arm architecture does not define when asynchronous exceptions are taken. Therefore the prioritization of asynchronous exceptions relative to other exceptions, both synchronous and asynchronous, is **IMPLEMENTATION DEFINED**.

A simple matched example would be if the PE is executing application code at EL0 in AArch32 and an IRQ interrupt occurs. In this example, HCR_EL2 and SCR_EL3 have been configured such that IRQ exceptions are routed to EL1 in AArch64. This is summarized in the following diagram:

Figure 5-5: Simple exception handler example



- The current PSTATE is preserved as a snapshot in SPSR_EL1
- The preferred exception return address (the first instruction that has not yet completed) is written to ELR_EL1
- The target Execution state is determined by reading the RW bit of the Hypervisor Configuration Register HCR_EL2.RW
- The current PSTATE is updated, with Exception level changed to EL1 and Execution state to AArch64
- The core branches to the vector table VBAR_EL1, offset to VBAR_EL1 + `0x680` because this is an IRQ exception where the exception is from a lower EL and all lower ELs are AArch32
- Defined registers are stacked to maintain register context
- Specific IRQ handler code is executed
- Once the IRQ-specific handler code completes, control returns to the high-level handler
- The handler restores all registers and executes an `ERET` instruction
- PSTATE is restored from SPSR_EL1 (including the return to Exception level EL0 and target Execution state AArch32) and the program counter is updated to the value contained in ELR_EL1

Note the IRQ exception itself does not distinguish between different causes of interrupt (for example, timer or UART). Assuming a GIC is being used, this can be identified by reading the GIC's Interrupt Acknowledge Registers (IAR). The read returns the ID of the interrupt and marks that

interrupt as active in the GIC. Once the interrupt is handled, the GIC state needs to be cleared back to inactive by writing to the GIC's End of Interrupt Register (EOIR).

5.3.3 Exception masking and non-maskable interrupts (NMI)

The previous example represented a very simple case of an interrupt. It is sometimes necessary to be able to disable or mask other interrupts from overriding the currently completing exception. Both physical and virtual asynchronous exceptions can be temporarily masked and left in a pending state until unmasked and the exception is taken. This is done through masking interrupts of the same type until explicitly enabled later.

When the processor takes an exception to an AArch64 execution state, the PSTATE interrupt masks (PSTATE.DAIF) are set automatically. DAIF stands for debug, abort (SError), IRQ, and FIQ. The DAIF field is 4 bits, with each bit corresponding to one of the mentioned exception types. By writing a 1 to a bit in the field, we mask or ignore the exception type. It can go to pending, but not get handled. In other words, the PE does not branch to the exception handler until the bit is unmasked, effectively disabling further exceptions of that type from being taken.

Interrupts are always masked at the Exception level where the interrupt is taken. Synchronous exceptions cannot be masked. This is because synchronous exceptions are caused directly by the execution of an instruction so would block execution if it were then left pending or ignored.

You cannot mask asynchronous exceptions routed to a higher Exception level. Software can still set the PSTATE mask bits at the lower EL, however this would not prevent the exception from being taken. For example, if you have FIQs routed to EL3 (SCR_EL3.FIQ=1) then setting PSTATE.F to 1 in EL1 or EL2 does not prevent the interrupt from being taken.

An exception routed to a lower Exception level is always masked regardless of PSTATE. For example, if IRQs are routed to EL2 or EL1 (SCR_EL1.IRQ=0) then IRQs are always implicitly masked while the PE is in EL3.

If software is to support nested exceptions, for example, to allow a higher priority exception to interrupt the handling of a lower priority exception, then software needs to explicitly re-enable interrupts. After corruptible registers have been stacked, exceptions can be nested.

Non-maskable interrupt (NMI) support was added in the 2021 extensions Armv8.8-A and Armv9.3-A. An interrupt with superpriority is classed as an NMI and can be taken even when the PSTATE exception masks would normally prevent it being taken.

There are some restrictions on NMIs. All interrupts are masked, including NMIs, on first taking an interrupt exception. This is to allow software to save required state before it can be overwritten by subsequent interrupts.

There are some points in time when software is not able to handle any interrupts, including NMIs. To deal with this a new PSTATE mask, ALLINT, has been added. This allows software to choose between masking no interrupts, most interrupts (excluding NMIs) and all interrupts (including NMIs).

Three models of NMI support can be selected using the System Control Register:

- SCTLR_ELx.NMI = 0: NMI support disabled. Interrupts can be masked using PSTATE.I and PSTATE.F.
- SCTLR_ELx.NMI = 1: NMI support is enabled.
    - SCTLR_ELx.SPINTMASK = 0 : NMIs are masked by the PSTATE.ALLINT mask.
    - SCTLR_ELx.SPINTMASK = 1 : NMIs are masked by the PSTATE.ALLINT mask or when the target Exception level is using SP_ELx.

NMI support requires the system's interrupt controller to be able to present interrupts with superpriority. If using Arm's Generic Interrupt Controller, support for NMIs is added in GICv3.3 and GICv4.2.

You can find out more about Arm A-profile support for NMIs in the following blog post.

6. Check your knowledge

The following questions help you test your knowledge.

What Exception levels are implemented in Armv8-A?

EL0 and EL1 are mandatory. EL2 and EL3 are optional but implemented by most designs.

What are the Execution states?

AArch32 and AArch64

Which stack is used on exception entry?

SP_ELx is automatically selected to provide a safe exception stack.

How are the vector tables implemented in AArch64

The PE holds the base address of the table in VBAR_ELx. The table itself is instruction memory.

7. Related information

Here are some resources related to material in this guide:

- Arm architecture and reference manuals
- Arm Community: Ask development questions, and find articles and blogs on specific topics from Arm experts

The following Learn the Architecture guides are related to material in this guide:

- TrustZone for AArch64
- Realm Management Extension
- Arm Generic Interrupt Controller v3 and v4
- AArch64 self-hosted debug
- AArch64 memory management
- AArch64 virtualization
- AArch64 Instruction Set Architecture

The following training course is related to material in this guide:

- Introduction to Armv8-A

8. Next steps

This guide has introduced the AArch64 Exception model and exception handling using AArch64. We have looked at Execution and Security states, exception types, and exception handling including AArch64 vector tables.

This knowledge will be useful as you begin to learn more about the architecture, how interrupts work, and the flow of processor behavior. You can put your knowledge into action in developing embedded code, creating vector tables and exception handlers.