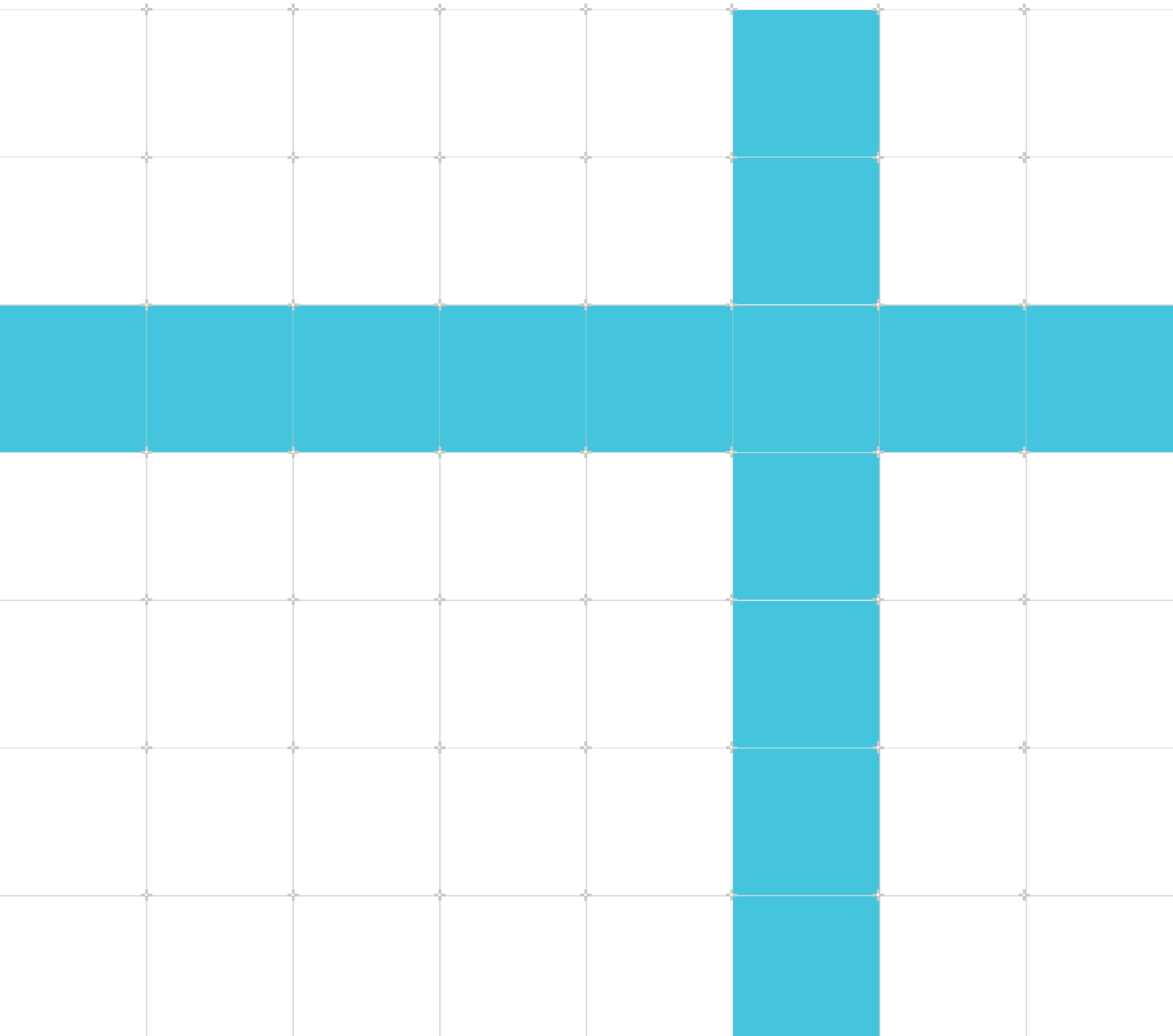
arm

Learn the architecture - AArch64 Exception model

Version 1.2

Issue 02
102412_0102_02_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

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

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

This guide introduces the exception and privilege model in Armv8-A and Armv9-A. This guide covers the different types of exceptions in the Arm architecture, and the behavior of the processor when it receives an exception.

This guide is suitable for developers of low-level code, such as boot code or drivers. It is particularly relevant to anyone writing code to set up or manage the exceptions.

At the end of this guide you can check your knowledge. You will be able to list the Exception levels in and state how execution can move between them, and name and describe the Execution states. You will also be able to create a simple AArch64 vector table and exception handler.

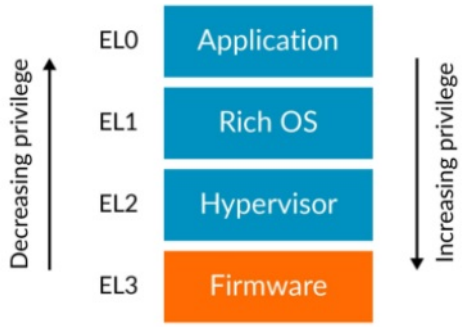
2. Privilege and Exception levels

Before we explain the details of the Armv8-A and Armv9-A exception model, let's start by introducing the concept of privilege. Modern software expects 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, which has a high level of access to system resources, and user applications, which have a more limited ability to configure the system.

Armv8-A and Armv9-A enables this split by implementing different levels of privilege. The current level of privilege can only change when the processor takes or returns from an exception. Therefore, these privilege levels are referred to as Exception levels in the Arm architecture. Each Exception level is numbered, and the higher levels of privilege have higher numbers.

As shown in the following diagram, the Exception levels are referred to as EL<x>, with x as a number between 0 and 3. For example, the lowest level of privilege is referred to as EL0.

Figure 2-1: Exception levels



A common usage model has application code running at EL0, with an operating system running at EL1. EL2 is used by a hypervisor, with EL3 being reserved by low-level firmware and security code.

Note

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

Types of privilege

There are two types of privilege relevant to this topic. The first is privilege in the memory system, and the second is privilege from the point of view of accessing processor resources. Both are affected by the current Exception level.

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, which can be configured with two degrees of freedom. This configuration allows separate access permissions for privileged and unprivileged accesses.

Memory access initiated when the processor is executing in EL0 will be checked against the Unprivileged access permissions. Memory accesses from EL1, EL2 and EL3 will be 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.

Register Access

Configuration settings for Armv8-A processors are held in a series of registers known as System registers. The combination of settings in the System registers define the current processor Context. Access to the System registers is controlled by the current Exception level.

The name of the System register indicates the lowest Exception level from which that register can be accessed. For instance, TTBR0_EL1 is the register that holds the base address of the translation table used by EL0 and EL1. This register cannot be accessed from EL0, and any attempt to do so will cause an exception to be generated.

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. For example, the following registers all perform MMU configuration for different translation regimes.

The registers have similar names to reflect that they perform similar tasks, but they are entirely independent registers with their own access semantics:

- `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


Note

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. In the general operation of the system, the privileged Exception levels will usually control their own configuration. However, more privileged levels will sometimes access registers associated with lower Exception levels to for example, implement virtualization features or to read and write the register set as part of a save-and-restore operation during a context switch or power management operation.

3. Execution and Security states

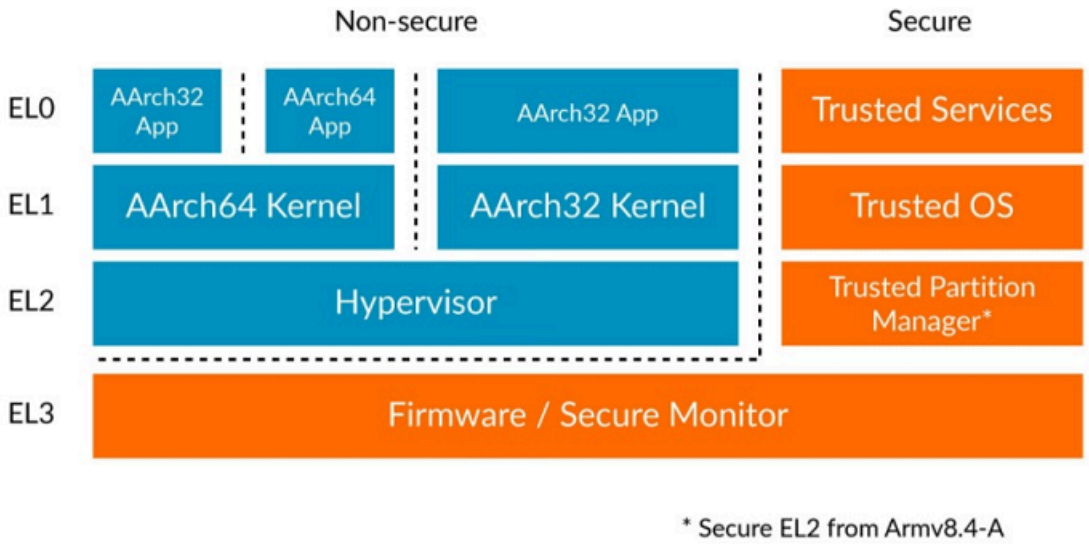
The current state of an Armv8-A or Armv9-A processor is determined by the Exception level and the current Execution state. The current Execution state defines the standard width of the general-purpose register and the available instruction sets.

Execution state also affects aspects of the memory models and how exceptions are managed.

The current Security state controls which Exception levels are currently valid, which areas of memory can currently be accessed, and how those accesses are represented on the system memory bus.

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

Figure 3-1: Exception levels and Security states using different Execution states



Execution states

Armv8-A has two available Execution states:

- AArch32: The 32-bit Execution state. Operation in this state is compatible with Armv7-A. There are two available instruction sets: T32 and A32. The standard register width is 32 bits.
- AArch64: The 64-bit Execution state. There is one available instruction set: A64. The standard register width is 64 bits.

It is implementation defined which execution states an Armv8-A processor supports at a given Exception level. For example:

- Cortex-A35 support AArch32 and AArch64 at all Exception levels.
- Cortex-A32 only supports AArch32.

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

Armv9-A supports AArch64 at all Exception levels, AArch32 is optionally supported, but only at EL0.

Security state

The Armv8-A architecture allows for implementation of two Security states. This allows a further partitioning of software to isolate and compartmentalize trusted software.

The two Security states are:

- Secure state: In this state, a Processing Element (PE) can access both the Secure and Non-secure physical address spaces. In this state, the PE can access Secure and Non-secure System registers. Software running in this state can only acknowledge Secure interrupts.
- Non-secure state: In this state, a PE can only access the Non-secure physical address space. The PE can also only access System registers that allow non-secure accesses. Software running in this state can only acknowledge Non-secure interrupts.

The uses of these Security states are described in more detail in our guide TrustZone for Armv8-A.

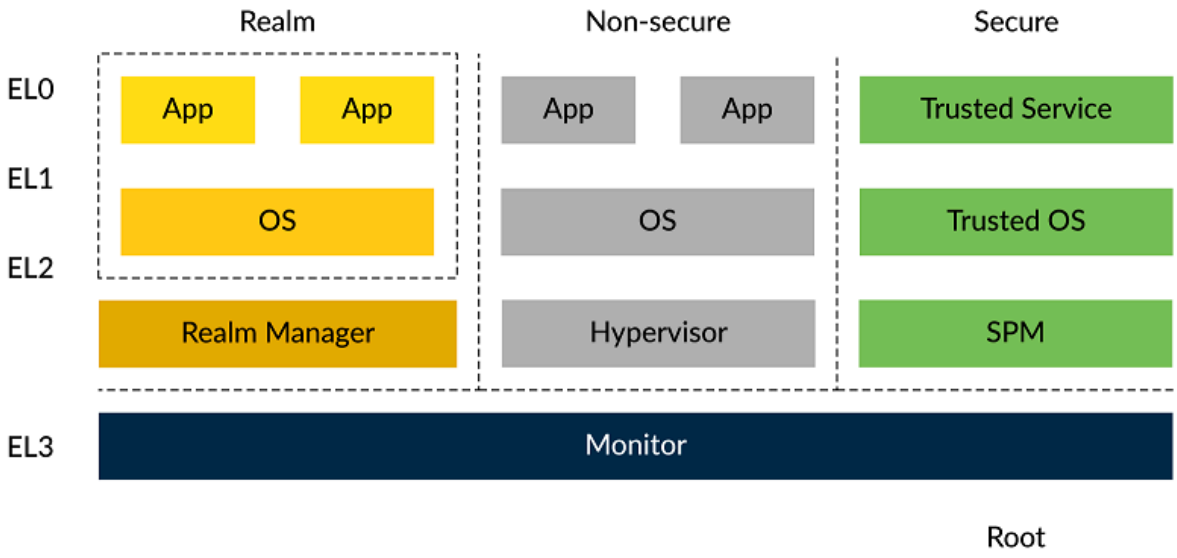
Realm Management Extension

Armv9-A introduces 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.

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

Figure 3-2: Security states in an RME-enabled PE



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

Changing Execution state

A PE can only change Execution state on reset or when the Exception level changes.

For Armv8-A processors, the Execution state on reset is determined by an **IMPLEMENTATION DEFINED** mechanism. Some implementations fix the Execution state at reset. For example, Cortex-A32 will always reset into AArch32 state. In most implementations of Armv8-A, the Executions state after reset is controlled by a signal that is sampled at reset. This allows the reset Execution state to be controlled at the system-on-chip level.

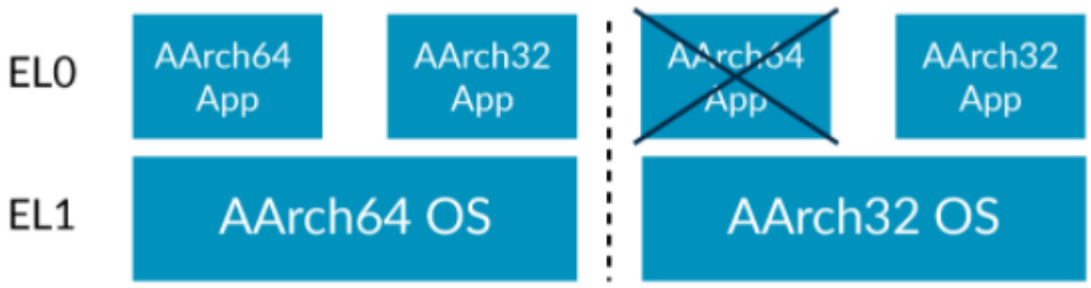
For Armv9-A processors, AArch32 is only supported at EL0. Therefore, on reset the Execution state is always AArch64.

When the PE changes between Exception levels, it is also possible to change Execution state. Transitioning between AArch32 and AArch64 is only allowed subject to certain 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 could only host 32-bit applications. This is illustrated here:

Figure 3-3: 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 32-bit virtual machines at EL1.

Changing Security state

EL3 is the most privileged Exception level. Software at EL3 is responsible for managing access to the different Security states available.

The Security state of EL3 is fixed. In Armv8-A EL3 is always in Secure state. In Armv9-A, EL3 is part of Root state if RME is implemented, otherwise it is part of Secure state.

Software in EL3 controls the Security state of EL2, EL1 and EL0. Software configures the SCR_EL3.NS field, and if RME is implemented SCR_EL3.NSE field, to select the desired Security state. When software performs an exception return leaving EL3, the Security state selected by the NS and NSE fields is entered.

Changing Security state is discussed in more detail in our TrustZone, and RME guides.

Implemented Exception levels and Execution states

The Arm architecture allows an implementation to choose whether all Exception levels are implemented, and to choose which Execution states are allowed for each implemented Exception level.

EL0 and EL1 are the only Exception levels that must be implemented. EL2 and EL3 are optional. Choosing not to implement EL3 or EL2 has important implications.

EL3 is the only level that can change Security state. If an implementation chooses not to implement EL3, that PE would not have access to a single Security state.

Similarly, EL2 contains much of the virtualization functionality. Implementations that do not have EL2 have access to these features. All current Arm implementations of the architecture implement all Exception levels, and it would be impossible to use most standard software without all Exception levels.

An implementation can also choose which Execution states are valid for each Exception level. If AArch32 is allowed at an Exception level, it must be allowed 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-A32 only allows AArch32 at any Exception level.

Implementations, such as the Neoverse-N2 implement all Exception levels but only allow AArch32 at EL0. The other exception levels, EL1, EL2, and EL3, must be AArch64.

4. Exception types

An exception is any event that can cause the currently executing program to be suspended and cause a change in state to execute code to handle that exception. Other processor architectures might describe this as an interrupt. In the Armv8-A and Armv9-A architecture, interrupts are a type of externally generated exception. The Arm architecture categorizes exceptions into two broad types: synchronous exceptions and asynchronous exceptions.

Synchronous exceptions

Synchronous exceptions are exceptions that can be caused by, or related to, the instruction that has just been executed. This means that synchronous exceptions are synchronous to the execution stream.

Synchronous exceptions can be caused by attempting to execute an invalid instruction, either one that is not allowed at the current Exception level or one that has been disabled.

Synchronous exceptions can also be caused by memory accesses, as a result of either a misaligned address or because one of the MMU permissions checks has failed. Because these errors are synchronous, the exception can be taken before the memory access is attempted. Memory accesses can also generate asynchronous exceptions, which are discussed in this section. Memory access errors are discussed in more detail in the Memory Management guide.

The Arm architecture has a family of exception-generating instructions: SVC, HVC, and SMC. These instructions are different from a simple invalid instruction, because they target different exception levels and are treated differently when prioritizing exceptions. These instructions are used to implement system call interfaces to allow less privileged code to request services from more privileged code.

Debug exceptions are also synchronous. Debug exceptions are discussed in the Debug overview guide.

Asynchronous exceptions

Some types of exceptions are generated externally, and therefore are not synchronous with the current instruction stream. This means that it is not possible to guarantee exactly when an asynchronous exception will be taken. The Armv8-A architecture requires only for it to happen in a finite time. Asynchronous exceptions can also be temporarily masked. This means that asynchronous exceptions can be left in a pending state before the exception is taken.

The asynchronous exception types are:

Physical interrupts

- SError (System Error)
- IRQ
- FIQ

Virtual Interrupts

- vSError (Virtual System Error)
- vIRQ (Virtual IRQ)
- vFIQ (Virtual FIQ)

The physical interrupts are generated in response to signal generated outside the PE. The virtual interrupts may be externally generated or may be generated by software executing at EL2. Virtual interrupts will be discussed in the Virtualization guide.

Let's look at the different types of physical interrupts.

IRQ and FIQ

The Arm architecture has two exception types, IRQ and FIQ, that are intended to be used to generate peripheral interrupts. In older versions of the Arm architecture, FIQ is used as a higher priority fast interrupt. This is different from Armv8-A and Armv9-A, in which FIQ has the same priority as IRQ.

IRQ and FIQ have independent routing controls and are often used to implement Secure and Non-secure interrupts, as discussed in the Generic Interrupt Controller guide.

Non-maskable interrupts

The 2021 extensions, Armv8.8-A and Armv9.3-A, add Non-maskable interrupt (NMI) support. When supported and enabled , an interrupt can be presented to the processor as having superpriority. 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 over written by subsequent interrupts are taken. There are some points in time when software not be able to handle any interrupts, including NMIs. To deal with this a new PSTATE mask, ALLINT, is added. This allows software to choose between masking no interrupts, most interrupts (but not NMIs) and all interrupts (including NMIs).

Three models of NMI support are supported:

- 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.

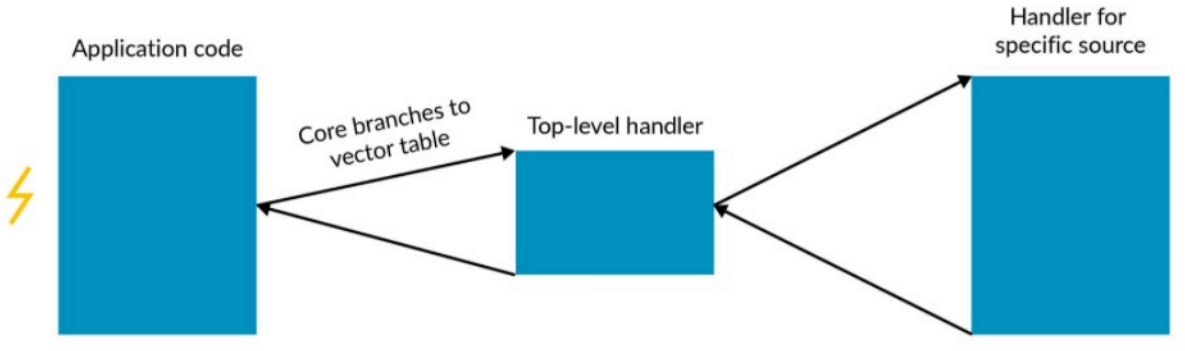
SError

SError is an exception type that is intended to be generated by the memory system in response to erroneous memory accesses. A typical use of SError is what was previously referred to as External, asynchronous abort, for example a memory access which has passed all the MMU checks but encounters an error on the memory bus. This may be reported asynchronously because the instruction may have already been retired. SError interrupts may also be caused by parity or Error Correction Code (ECC) checking on some RAMs, for example those in the built-in caches.

5. Handling exceptions

When an exception occurs, the current program flow is interrupted. The Processing Element (PE) will update the current state and branch to a location in the vector table. Usually this location will contain generic code to push the state of the current program onto the stack and then branch to further code. This is illustrated here:

Figure 5-1: Top-level handler



Exception terminology

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.

The Arm architecture has instructions that trigger an exception return. In that case, the state that the PE is in when that instruction is executed is the state that the exception returns from. The state after the exception return instruction has executed is the state that the exception returns to.

Each exception type targets an Exception level. Asynchronous exceptions can be routed to different exception levels.

Taking an exception

When an exception is taken, the current state must be preserved so that it can be returned to. The PE will automatically preserve the exception return address and the current `PSTATE`.

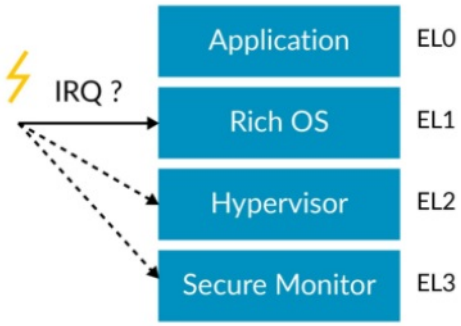
The state stored in the general-purpose registers must be preserved by software. The PE will then update the current `PSTATE` to the one defined in the architecture for that exception type, and branch to the exception handler in the vector table.

The `PSTATE` the exception was taken from is stored in the System register `SPSR_ELx`, where <x> is the number of the Exception level that the exception was taken to. The exception return address is stored in `ELR_ELx`, where <x> is the Exception level that the exception was taken to.

Routing asynchronous exceptions

The three physical interrupt types can be independently routed to one of the privileged Exception levels, EL1, EL2 or EL3. The diagram below uses IRQs as an example:

Figure 5-2: IRQs routed to the privileged Exception levels



This routing is configured using `SCR_EL3` and `HCR_EL2`. Routing configurations made using `SCR_EL3` will override routing configurations made using `HCR_EL2`. These controls allow different interrupt types to be routed to different software.

Exceptions that are routed to a lower Exception level than the level being executed are implicitly masked. The exception will be pended until the PE changes to an Exception level equal to, or lower than, the one routed to.

Determining which Execution state an exception is taken to

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:

Exception level taken to:	Execution state determined by:
EL0	`Exceptions are never taken to EL0`
EL1	`HCR_EL2.RW`
EL2	`SCR_EL3.RW`
EL3	Reset state of EL3

Returning from an exception

Software can initiate a return from an exception by executing an ERET instruction from AArch64. This will cause the Exception level returned to be configured based on the value of `SPSR_ELx`, where <x> is the level being returned from. `SPSR_ELx` contains the target level to be returned to and the target Execution state.

Note that the Execution state specified in `SPSR_ELx` must match the configuration in either `SCR_EL3.RW` or `HCR_EL2.RW`, or this will generate an illegal exception return.

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

Exception stacks

When executing in AArch64, the architecture allows a choice of two stack pointer registers; `SP_EL0` or `SP_ELx`, where <x> is the current Exception level. For example, at EL1 it is possible to select `SP_EL0` or `SP_EL1`.

During general execution, it is expected that all code uses SP_EL0. When taking an exception, SP_ELx is initially selected. This allows a separate stack to be maintained for initial exception handling. This is useful for maintaining a valid stack when handling exceptions caused by stack overflows.

6. The vector tables

When taking an exception to Exception level using AArch64, vector tables are an area of normal memory containing instructions. The Processor Element (PE) holds the base address of the table in a System register, and each exception type has a defined offset from that base.

The privileged Exception levels each have their own vector table defined by a Vector Base Address Register, `VBAR_ELx`, where <x> is 1,2, or 3.

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

The format of the vector table is shown below:

Figure 6-1: Vector table

Each exception type can cause a branch to one of four locations based on the state of the Exception level the exception was taken from.

7. Check your knowledge

The following questions will help you test you 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.

8. 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
- TrustZone for Armv8-A

Useful links to training

- Introduction to Armv8-A

Exception types

- Before Debugging
- Generic Interrupt Controller
- Armv8-A memory management guide
- AArch64 Virtualization

9. Next steps

This guide has introduced the concept of the Armv8-A Exception model and exception handling using AArch64. We have looked at Execution and Security states, exception types, exception handling, and the vector table.

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 the vector table and exception handlers.