AArch64 Exception and Interrupt Handling

Version 1.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
28 February 2017	0100	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

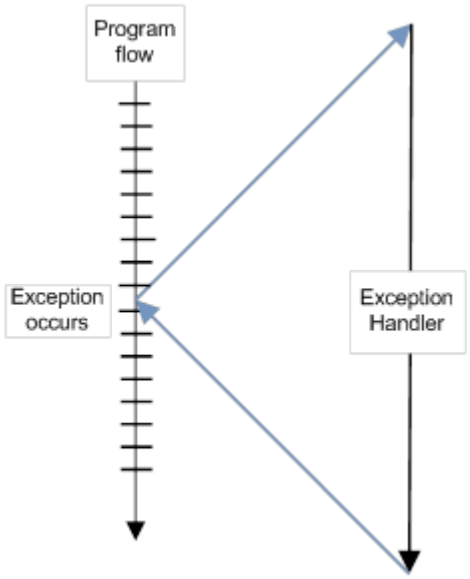
Web Address

http://www.arm.com

Contents

1 AArch64 Exception and interrupt handling

Exceptions are conditions or system events that require some action by privileged software (an exception handler) to ensure smooth functioning of the system. They cause an interruption in the flow of execution.



The term interrupt is sometimes used as a synonym for exception. In ARM terminology, certain types of asynchronous exceptions are referred to as interrupts.

One way to distinguish between the two is that an exception is an event (other than branch or jump instructions) that causes the normal sequential execution of instructions to be modified. An interrupt is an exception that is not caused directly by program execution. Usually, hardware external to the processor core signals an interrupt, such as a button being pressed.

There is an exception handler that is associated with each exception type. When the exception has been handled, privileged software prepares the core to resume whatever it was doing before taking the exception.

Commonly, interrupt is used to mean interrupt signal. On ARM A-profile processors, that means an IRQ or FIQ interrupt signal. The ARM architecture splits exceptions into two groups, synchronous and asynchronous. The synchronous exception types can have many causes but they are handled in a similar way. The asynchronous exception type is subdivided into three interrupt types, IRQ, FIQ, and SError (System Error).

The following types of action can cause an exception:

Aborts Aborts can be generated either on failed instruction fetches (Instruction Aborts) or failed data accesses (Data Aborts). They can come from the external memory system giving an error response on a memory access (indicating perhaps that the specified address does not correspond to real memory in the system).

Alternatively, the Memory Management Unit (MMU) of the core generates the abort. An OS can use MMU aborts to allocate memory to applications dynamically.

An instruction that cannot be fetched causes an abort. The Instruction Abort exception is only taken if the core then tries to execute it. A Data Abort exception is caused by a load or store instruction and happens after the data read or write has been attempted.

An abort is described as being synchronous if it is generated by direct execution of instructions and the return address indicates the instruction which caused it.

Otherwise, the abort is described as asynchronous.

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

Reset

Reset is treated as a special case because it has its own vector that always targets the highest implemented Exception level. This vector uses an IMPLEMENTATION DEFINED address which is typically set by configuration input signals.

The address can be read from the Reset Vector Base Address Register RVBAR_EL*n*, where n is the number of the highest implemented Exception level.

All cores have a reset input and take the reset exception after they have been reset. It is the highest priority exception and cannot be masked. This exception is used to execute code on the core to initialize it, after the system has powered up.

Exception generating instructions

Execution of these instructions can generate exceptions. They are typically executed to request a service from software that runs at a higher privilege level:•

- The Supervisor Call (SVC) instruction enables User mode programs to request an OS service.
- The Hypervisor Call (HVC) instruction enables the guest OS to request hypervisor services.
- The Secure monitor Call (SMC) instruction enables the Normal world to request Secure world services.

Interrupts

There are three types of interrupts, IRQ, FIQ and SError. IRQ and FIQ are general purpose compared to SError, which is associated specifically with external asynchronous Data Aborts. So typically, the term 'interrupts' refers only to IRQ and FIQ.

FIQ is higher priority than IRQ. Both of these interrupts are typically associated with individual input pins for each core. External hardware asserts an interrupt request line and the corresponding exception type is raised when the current instruction finishes executing (although some instructions, those that can load multiple values, can be interrupted), assuming that the interrupt is not disabled.

On almost all systems, various interrupt sources are connected using an interrupt controller. The interrupt controller arbitrates and prioritizes interrupts, and in turn, provides a serialized single signal that is then connected to the FIQ or IRQ signal of the core.

Because IRQ and FIQ interrupts are not directly related to the software running on the core at any given time, they are classified as asynchronous exceptions.

2 Synchronous and asynchronous exceptions

In AArch64, exceptions can be either synchronous, or asynchronous.

- An exception is described as being synchronous if it is generated by direct execution of instructions and the return address indicates the instruction which caused it.
- Otherwise, the abort is described as asynchronous.

Sources of asynchronous exceptions are IRQ, FIQ, or SError (System Error). System errors have several possible causes, the most common being asynchronous Data Aborts (for example, an abort that is triggered by write-back of dirty data from a cache line to external memory).

2.1 Handling synchronous exceptions

The Exception Syndrome Register (ESR_EL*n*) and The Fault Address Register (FAR_EL*n*) are provided to supply information to exception handlers about the cause of a synchronous exception. The ESR_EL*n* gives information about the reasons for the exception, while the FAR_EL*n* holds the faulting virtual address for all synchronous instruction and Data Aborts and alignment faults.

The Exception Link Register (ELR_EL*n*) also holds the address of the instruction that caused the aborting data access (for Data Aborts). These are updated after a memory fault, but are set in other circumstances, for example, by branching to a misaligned address.

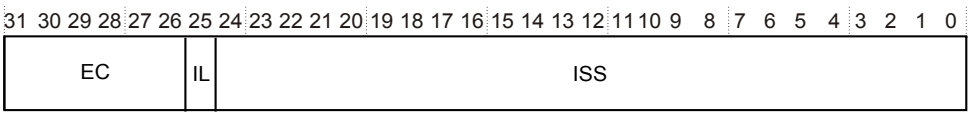
If an exception is taken from an Exception level in AArch32 into an Exception level using AArch64, and the exception writes the Fault Address Register that is associated with the target Exception level, the top 32 bits of the FAR_EL*n* are all set to zero.

For systems which implement EL2 (Hypervisor) or EL3 (Secure Kernel), Synchronous exceptions are normally taken in the current or a higher Exception level. Asynchronous exceptions can (if necessary), be routed to a higher Exception level to be dealt with by a Hypervisor or Secure kernel. The SCR_EL3 register specifies which exceptions are routed to EL3 and similarly, HCR_EL2 specifies which exceptions are routed to EL2. There are separate bits that allow individual control over routing of IRQ, FIQ, and SError.

2.2 The Exception Syndrome Register

The Exception Syndrome Register, ESR_ELn, contains information that allows the exception handler to determine the reason for the exception. It is updated only for synchronous exceptions and SError. It is not updated for IRQ or FIQ as these interrupt handlers typically obtain status information from registers in the Generic Interrupt Controller (GIC).

The bit coding for the register is:



- Bits [31:26] (ESR_ELn.EC) indicate the exception class which enables the handler to distinguish between the various possible exception causes (such as unallocated instruction, exceptions

from MCR or MRC to CP15, exception from FP operation, SVC, HVC or SMC executed, Data Aborts, and alignment exceptions). For example, EC = 101111 is an SError interrupt.

- Bit [25] (ESR_ELn.IL) indicates the length of the trapped instruction (0 for a 16-bit instruction or 1 for a 32-bit instruction) and is set for certain exception classes.
- Bits [24:0] (ESR_ELn.ISS) form the Instruction Specific Syndrome (ISS) field containing information specific to that exception type. For example, when a system call instruction (SVC, HVC or SMC) is executed, the field contains the immediate value that is associated with the opcode such as 0x123456 for SVC.

2.3 Unallocated instructions

Unallocated instructions cause a Synchronous Abort in AArch64. This exception type is generated when the processor executes one of the following:

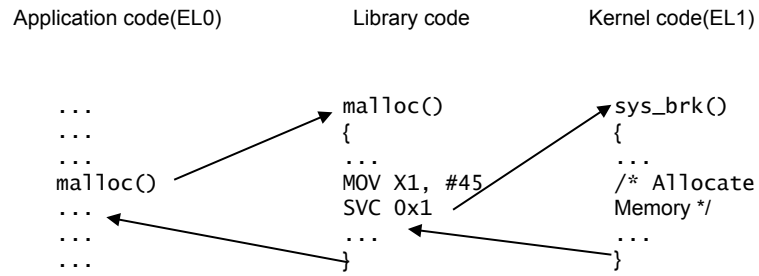
- An instruction opcode that is not allocated.
- An instruction that requires a higher level of privilege than the current Exception level.
- An instruction that has been disabled.
- Any instruction when the PSTATE.IL field is set.

2.4 System calls

Some instructions or system functions can only be carried out at a specific Exception level. For example, if code running at a lower Exception level has to perform a privileged operation, such as when application code requests functionality from the kernel. One way to do this is by using the SVC instruction. This allows applications to generate an exception. Parameters can be passed in registers, or coded within the System call.

For example:

Application code at EL0 requests memory using `malloc()`

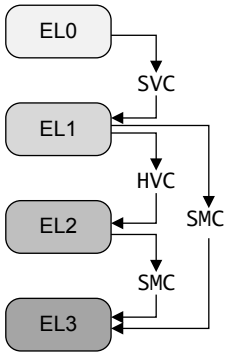


2.5 System calls to EL2/EL3

SVC instructions can be used to call from user applications at EL0 to the kernel at EL1. The HVC and SMC system-call instructions move the processor in a similar fashion to EL2 and EL3. When the processor is executing at EL0 (Application), it cannot call directly into the hypervisor (EL2) or

Secure monitor (EL3). This is only possible from EL1 and above. Applications must therefore use SVC to call into kernel and allow the kernel to call into higher Exception levels on their behalf.

From the OS kernel (EL1), software can call the hypervisor (EL2) with the HVC instruction, or call the Secure monitor (EL3) with the SMC instruction. If the processor is implemented with EL3, the ability to have EL2 trap SMC instructions from EL1 is provided. If there is no EL3, the SMC is unallocated and triggers at the current Exception level.



Similarly, from hypervisor code (EL2), the program can call the Secure monitor (EL3) with the SMC instruction. If you make an SMC call when in EL2 or EL3, it still causes a synchronous exception at the same Exception level, and the handler for that Exception level can decide how to respond.

3 Exception handling

The ARMv8-A architecture has four Exception levels, EL0, EL1, EL2, and EL3. Processor execution can only move between Exception levels by taking, or returning from, an exception.

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

An exception causes a change of program flow. Execution restarts in the Exception level to which the Exception is taken, from the exception vector that corresponds to the exception taken. That is, the exception vector holds the first instruction of the exception handler.

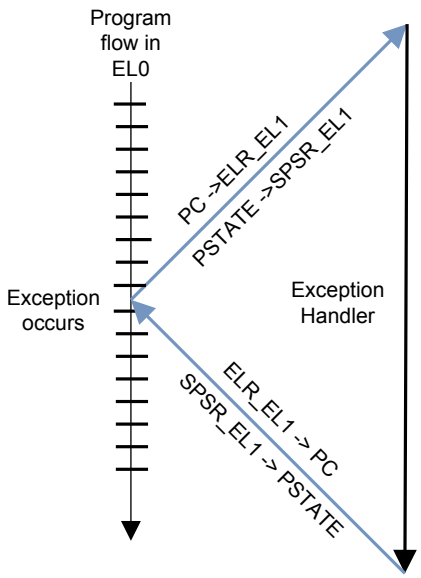
When an event that causes an exception occurs, the processor hardware automatically performs several actions:

1. The SPSR_ELn is updated (where n is the Exception level where the exception is taken), to store the PSTATE information that is required to correctly return at the end of the exception.

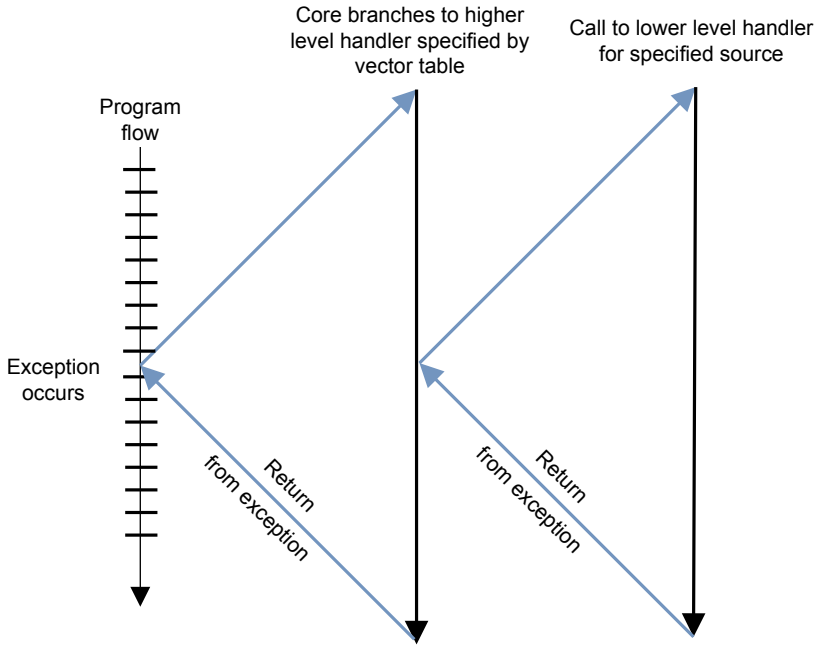
2. PSTATE is updated to reflect the new processor status (and this can mean that the Exception level is raised, or it can stay the same).

3. The address to return to at the end of the exception is stored in ELR_EL*n*.

   The _ELn suffix on register names denotes that there are multiple copies of these registers existing at different Exception levels. This means, for example, that SPSR_EL1 is a different physical register to SPSR_EL2.



The processor branches to a vector table which contains entries for each exception type. The vector table contains a dispatch code, which typically identifies the cause of the exception, and selects and calls the relevant function to handle it.



When the handler code completes execution it returns to the high-level handler, which then executes the ERET instruction to return to the application.

The target Execution State (that is, AArch32 or AArch64) and Exception level for asynchronous exceptions are configured using two system registers, SCR_EL3 and HCR_EL2.

4 Example exception handlers

The following code provides examples of how exception handlers might be structured.

4.1 A simple exception handler

The following code demonstrates a simple exception handler for non-nested interrupts:

```
ASM_IRQ_Handler:
   ; Stack all corruptible registers            //
   STP X0, X1, [SP, #-16]!                      // save PCS corruptible.
   STP X2, X3, [SP, #-16]!                      // interrupts
   ...                                          // PUSH the rest of the
                                                // corruptible registers
```

Corruptible Registers are X0-X15.

```
   BL     identify_and_clear_source
   BL     C_IRQ_Handler

   ; Restore corruptible registers
   ...
   LDP X2, X3, [SP], #16                        // Restore PCS corruptible
   LDP X0, X1, [SP], #16                        // interrupts

   ; Return from exception                      //
   ERET                                         // Return from exception
```

4.2 A nested exception handler

The following code demonstrates a simple exception handler for nested interrupts:

```
ASM_IRQ_Handler:
   ; Stack all corruptible registers                        // Save
   ...
   ; Read SPSR_EL1 and ELR_EL1 into GP registers       // PCS
   MRS X0, SPSR_EL1                                   // Corruptible
   MRS X1, ELR_EL1                                    // registers
   ; Stack SPSR_EL1 and ELR_EL1                       // SPSR_EL1
   STP X0, X1, [SP, #-16]!                            // ELR-EL1
   BL     identify_and_clear_source
   ; Unmask IRQs
   MSR DAIFClr, #0b0010                               // Unmask
   BL     C_IRQ_Handler                               // IRQs
   ; Mask IRQs
   MSR DAIFSet, #0b0010                               // Mask
   ; Restore SPSR_EL1 and ELR_EL1                     // IRQs
```

```
LDP X0, X1, [SP], #16                    // Restore PCS
MSR SPSR_EL1, X0                         // corruptible
MSR ELR_EL1, X1                          // registers, SPSR_EL1
; Restore corruptible registers          // and ELR_EL1
...
ERET                                     // Exception return
```

5 AArch64 exception vector table

When an exception occurs, the processor must execute handler code that corresponds to the exception. The location in memory where the handler is stored is called the *exception vector*. In the ARM architecture, exception vectors are stored in a table, called the *exception vector table*.

Each Exception level has its own vector table, that is, there is one for each of EL3, EL2, and EL1. The table contains instructions to be executed, rather than a set of addresses. These would normally be branch instructions that direct the core to the full exception handler.

The exception vector table for EL1, for example, holds instructions for handling all types of exception that can occur at EL1, Vectors for individual exceptions are at fixed offsets from the beginning of the table. The virtual address of each table base is set by the Vector Base Address Registers: VBAR_EL3, VBAR_EL2 and VBAR_EL1.

Each entry in the vector table is 16 instructions long (in ARMv7-A and AArch32, each entry is only 4 bytes). This means that in AArch64 the top-level handler can be written directly in the vector table.

The base address is given by VBAR_EL*n* and each entry has a defined offset from this base address. Each table has 16 entries, with each entry being 128 bytes (32 instructions) in size. The table effectively consists of 4 sets of 4 entries. Which entry is used depends on several factors:

- The type of exception (SError, FIQ, IRQ, or Synchronous)
- If the exception is being taken at the same Exception level, the stack pointer to be used (SP0 or SP*n*).
- If the exception is being taken at a lower Exception level, the Execution state of the next lower level (AArch64 or AArch32).

A typical vector table is shown below.

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	

+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Considering an example might make this easier to understand.

If kernel code is executing at EL1 and an IRQ interrupt is signaled, an IRQ exception occurs. This particular interrupt is not associated with the hypervisor or secure environment and is also handled within the kernel, and the SPSel bit is set, so SP_EL1 is used.

Execution takes place, therefore, from address VBAR_EL1 + 0x280.

In the absence of `LDR PC, [PC, #offset]` in the ARMv8-A architecture, more instructions must be used for the destination to be read from a table of registers. The spacing of the vectors is designed to avoid cache pollution for typical sized instruction cache lines from vectors that are not being used. The Reset Address is a separate address, which is IMPLEMENTATION DEFINED, and is typically set by hardwired configuration within the core. This address is visible in the RVBAR_EL1/2/3 register.

Having a separate exception vector for each exception gives the flexibility for the OS or hypervisor to determine the AArch64 and AArch32 state of the lower Exception levels. The SP_ELn is used for exceptions generated from lower levels. However, the software can switch to use SP_EL0 inside the handler. When this mechanism is used, it facilitates access to the values from the thread in the handler.

The following code is for a typical vector table for EL3 vectors for exceptions in AArch64.

```
// Typical exception vector table code.
.balign 0x800
Vector_table_el3:
curr_el_sp0_sync:        // The exception handler for a synchronous
                         // exception from the current EL using SP0.
.balign 0x80
curr_el_sp0_irq:         // The exception handler for an IRQ exception
                         // from the current EL using SP0.
.balign 0x80
curr_el_sp0_fiq:         // The exception handler for an FIQ exception
                         // from the current EL using SP0.
.balign 0x80
curr_el_sp0_serror:      // The exception handler for the system error
                         // exception from the current EL using SP0.
.balign 0x80
curr_el_spx_sync:        // The exception handler for a synchronous
                         // exception from the current EL using the
                         // current SP.
.balign 0x80
curr_el_spx_irq:         // The exception handler for an IRQ exception from
                         // the current EL using the current SP.

.balign 0x80
```

```
curr_el_spx_fiq:         // The exception handler for an FIQ from
                         // the current EL using the current SP.

.balign 0x80
curr_el_spx_serror:      // The exception handler for a System Error
                         // exception from the current EL using the
                         // current SP.

 .balign 0x80
lower_el_aarch64_sync:   // The exception handler for a synchronous
                         // exception from a lower EL (AArch64).

.balign 0x80
lower_el_aarch64_irq:    // The exception handler for an IRQ from a lower EL
                         // (AArch64).

.balign 0x80
lower_el_aarch64_fiq:    // The exception handler for an FIQ from a lower EL
                         // (AArch64).

.balign 0x80
lower_el_aarch64_serror: // The exception handler for a System Error
                         // exception from a lower EL(AArch64).

.balign 0x80
lower_el_aarch32_sync:   // The exception handler for a synchronous
                         // exception from a lower EL(AArch32).
.balign 0x80
lower_el_aarch32_irq:    // The exception handler for an IRQ exception
                         // from a lower EL (AArch32).
.balign 0x80
lower_el_aarch32_fiq:    // The exception handler for an FIQ exception from
                         // a lower EL (AArch32).
.balign 0x80
lower_el_aarch32_serror: // The exception handler for a System Error
                         // exception from a lower EL(AArch32).
```

6 Returning from an exception

The processor has to be told when to return from an exception by software. This is done in code using the ERET instruction. This restores the pre-exception PSTATE from SPSR_EL*n* and returns program execution back to the original location by restoring the PC from ELR_EL*n*.

The architecture provides separate link registers for subroutines and exception returns.

In the A64 instruction set, register X30 is used (with the RET instruction) to return from subroutines. Its value is updated with the address of the instruction to return to whenever a branch with link instruction (BL or BLR) is executed.

The ELR_EL*n* register is used to store the return address from an exception. The value in this register is automatically written on entry to an exception and is written to the PC as one of the effects of executing the ERET instruction that is used to return from exceptions.

> **Note**
>
> When returning from an exception, you see an error if the value in the SPSR conflicts with the settings in the System Registers.

ELR_EL*n* contains the return address, which depends on the specific exception type. Typically, this is the address of the instruction after the one that generated the exception.

For example, when an SVC (system call) instruction is executed, you want to return to the following instruction in the application. In other cases, however, you might want to re-execute the instruction that generated the exception.

For asynchronous exceptions, the ELR_EL*n* points to the address of the first instruction that has not been executed, or executed fully, because of taking the interrupt. Handler code is permitted to modify the ELR_E*n* if, for example, it was necessary to return to the instruction after aborting a synchronous exception. The ARMv8-A AArch64 model is simpler than that used in AArch32 or ARMv7-A, where for backward compatibility reasons, it was necessary to subtract 4 or 8 from the Link register value when returning from certain types of exception.

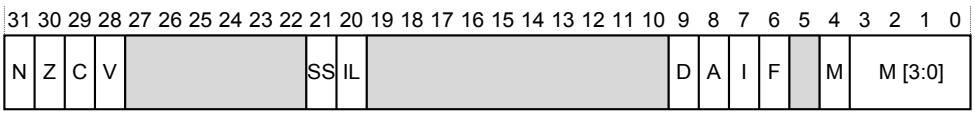
In addition to the SPSR and ELR registers, each Exception level has its own dedicated stack pointer register. These are SP_EL0, SP_EL1, SP_EL2 and SP_EL3. These registers are used to point to a dedicated stack. The stack can, for example, be used to store registers that are corrupted by the exception handler, so that they can be restored to their original value, before returning to the original code.

Handler code may switch from using SP_ELn to SP_EL0. For example, SP_EL1 might point to a piece of memory that holds a small stack that the kernel can always guarantee to be valid. SP_EL0 might point to a kernel task stack that is larger, but not guaranteed to be safe from overflow. This switching is controlled by writing to the SPSel register.

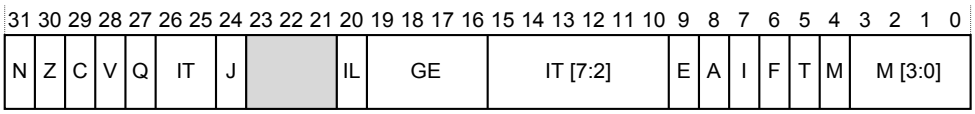
7 Processor state in exception handling

The current state of the processor is stored within separate PSTATE fields. If an exception is taken, the PSTATE information is saved in the Saved Program Status Register (SPSR_ELn) which exists as SPSR_EL3, SPSR_EL2, and SPSR_EL1.

At AArch64;



At AArch32:



When an exception is taken, the core state is saved from PSTATE in the SPSR at the Exception level the exception is taken to. For example, if the core takes an exception to EL1, the core state is saved in SPSR_EL1.

Name	Description	Notes
N	Negative condition flag.	
Z	Zero condition flag.	
C	Carry condition flag.	
V	oVerflow condition flag.	
D	Debug mask bit.	AArch64 only
A	SError mask bit.	
I	IRQ mask bit.	
F	FIQ mask bit.	
SS	Software Step bit.	
IL	Illegal Execution state bit.	
EL (2)	Exception level.	
nRW	Execution state. 0 = 64-bit1 = 32-bit	
SP	Stack pointer selector.0 = SP_EL01 = SP_ELn	AArch64 only
Q	Cumulative saturation (sticky) flag.	AArch32 only
GE (4)	Greater than or Equal flags.	AArch32 only
IT (8)	If-Then execution bits.	AArch32 only
J	J bit.	AArch32 only
T	T32 bit.	AArch32 only

E	Endianness bit.	AArch32 only
M	Mode field.	AArch32 only

The exception bit mask bits (DAIF) allow the exception events to be masked. The exception is not taken when the bit is set.

D Debug exceptions mask.

A SError interrupt Process state mask, for example, asynchronous external abort.

I IRQ interrupt Process state mask.

F FIQ interrupt Process state mask.

The SP field selects whether the current Exception level stack pointer or SP_EL0 is used. This can be done at any Exception level, except EL0.

The IL field, when set, causes execution of the next instruction to trigger an exception. It is used in illegal execution returns, for example, trying to return to EL2 as AArch64 when it is configured for AArch32.

The Software Stepping (SS) bit is used by debuggers to execute a single instruction and then take a debug exception on the following instruction.

PSTATE fields are accessed using special-purpose registers. The Special-purpose registers are read directly using MRS instruction, and written directly using MSR (register) instructions.

The special registers are:

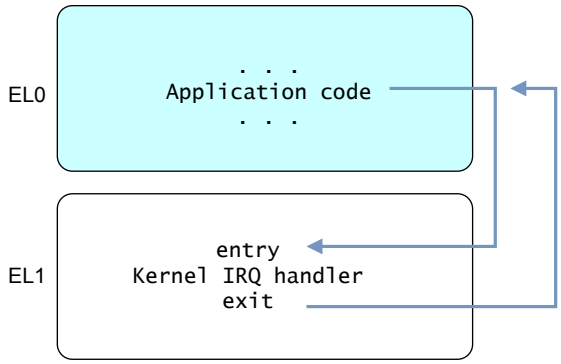
CurrentEL	Holds the current Exception level.
DAIF	Specifies the current interrupt mask bits.
NZCV	Holds the condition flags.
SPSel	At EL1 or higher, this selects between the SP for the current Exception level and SP_EL0.

Some of these separate fields, such as CurrentEL, DAIF, and NZCV, are copied into a compact form in SPSR_EL*n* when taking an exception (and the other way around when returning).

8 Changes to Execution state and Exception level caused by exceptions

When an exception is taken, the processor can change Execution state (from AArch64 to AArch32) or stay in the same Execution state. For example, an external source can generate an IRQ exception while executing an application in AArch32 and then execute the IRQ handler within the OS Kernel in AArch64.

Consider, for example, an application running in EL0, which is interrupted by an IRQ as in the figure below.



The Kernel IRQ handler runs at EL1. The processor determines which Execution state to set when it takes the IRQ exception. It determines the Execution state by reading the RW bit of the control register for the Exception level above the one where the exception is being handled. For example, if the exception is taken in EL1, the Execution state for the handler is controlled by HCR_EL2.RW.

Now consider what Exception level an exception is taken at.

When an exception is taken, the Exception level can stay the same, or it can get higher.

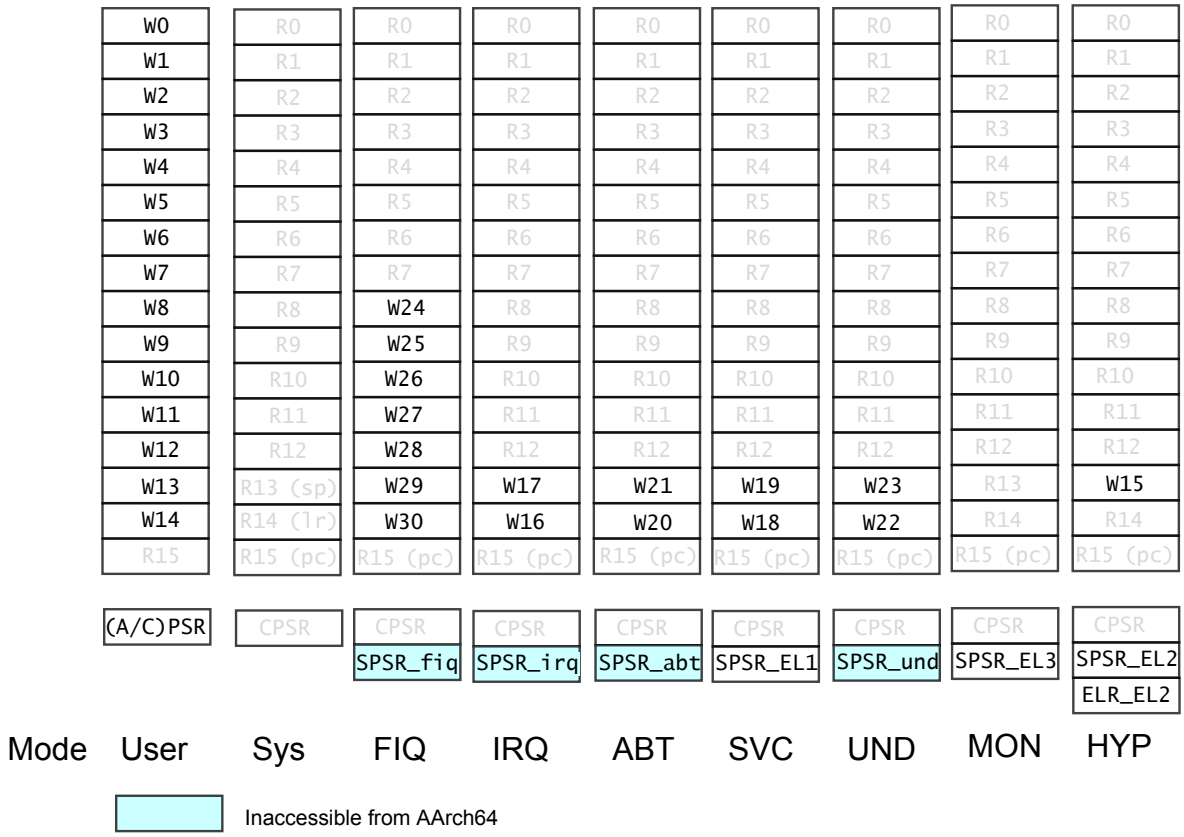
Note

Exceptions are never taken to EL0

Exceptions are typically taken in EL1 by default. When implemented, many classes of exception can instead be routed to EL2 (Hypervisor) using HCR_EL2 or EL3 (Secure Monitor) using SCR_EL3.

In both cases, there are separate bits to control routing of IRQ, FIQ, and SError. The Exception level can never go down by taking an exception. Interrupts are always masked at the Exception level where the interrupt is taken.

When taking an exception from AArch32 to AArch64, there are some special considerations. AArch64 handler code can require access to AArch32 registers and the architecture therefore defines mappings to allow access to AArch32 registers.



Bits [63:32] of the X registers are not available in AArch32 state and contain either 0 or the last value that is written in AArch64. There is no architectural guarantee on which value it is. It is therefore usual to access AArch32 registers as W registers.

9 Interrupt handling

ARM commonly uses interrupt to mean interrupt signal. On ARM A-profile and R-profile processors, that means an external IRQ or FIQ interrupt signal. The architecture does not specify how these signals are used. FIQ is often reserved for secure interrupt sources.

Both A-profile and R-profile processor cores have two physical interrupt pins, IRQ, and FIQ. An FIQ can mask IRQs, but an IRQ cannot mask FIQs. Historically, the architecture had features to permit lower-latency processing of FIQs.

When the processor takes an exception to AArch64 Execution state, all the PSTATE.DAIF interrupt masks are set automatically. This means that further interrupts are disabled. If software is to support nested interrupts, for example, to allow a higher priority interrupt to interrupt the handling of a lower priority source, then software must explicitly re-enable interrupts using the following instruction:

```
MSR DAIFClr, #imm
```

The immediate value is in fact a 4-bit field. There are also masks for:

- PSTATE.A (for SError)
- PSTATE.D (for Debug)

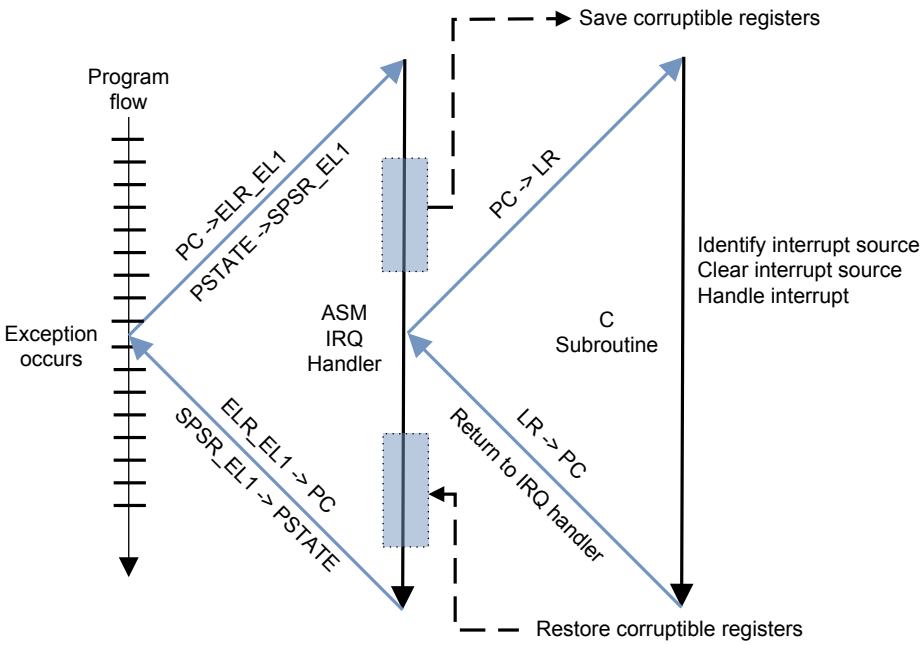


Figure 1 Interrupt handler in C code

An assembly language IRQ handler might look like this:

```
IRQ_Handler
// Stack all corruptible registers and save PCS corruptible interrupts
STP X0, X1, [SP, #-16]!            // SP = SP -16
STP X2, X3, [SP, #-16]!            // SP = SP – 16
...                                // PUSH the rest of the corruptible
                                   // registers
```

```
BL read_irq_source                  // a function to work out why we took an
                                    // interrupt and clear the request
BL C_irq_handler                    // the C interrupt handler
                                    // restore from stack the corruptible
                                    // registers
LDP X2, X3, [SP], #16               // S = SP + 16
LDP X0, X1, [SP], #16               // S = SP + 16
…
ERET
```

However, from a performance point of view, the following sequence can be used:

```
IRQ_Handler
SUB SP, SP, #<frame_size>           // SP = SP - <frame_size>
STP X0, X1, [SP] +16                // Store X0 and X1 at the base of the
                                    // frame
STP X2, X3, [SP] +16                // Store X2 and X3 at the base of the
                                    // frame
// + 16 bytes
...                                 // more register storing
...
// Interrupt handling

BL read_irq_source                  // a function to work out why we took an
                                    // interrupt and clear the request
BL C_irq_handler                    // the C interrupt handler
                                    // restore from stack the corruptible
                                    // registers
LDP X0, X1, [SP, #-16]!             // Load X0 and X1 at the base of the
                                    // frame
LDP X2, X3, [SP, #-16]!             // Load X2 and X3 at the base of the
                                    // frame +
                                    // 16 bytes
...                                 // more register loading
ADD SP, SP, #<frame_size>           // Restore SP at its original value
…
ERET
```

The nested handler requires a little extra code. It must preserve on the stack the contents of SPSR_EL1 and ELR_EL1. IRQs must be re-enabled after determining (and clearing) the interrupt source. However, as the link register for subroutine calls is different to the link register for exceptions, it is possible to avoid having to do anything special with LR or modes.