## Release information

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021/May/06</td>
<td>BETA</td>
<td>• BETA quality, for external release.</td>
</tr>
</tbody>
</table>
Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

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

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at http://www.arm.com/company/policies/trademarks.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.


110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0
Contents

Preview of Rules-based Arm® ARM D1 Exception Model

Release information ............................................................ ii
Non-Confidential Proprietary Notice .................................... iii

Part A  Preface

About this book

Conventions ................................................................. viii
Typographical conventions ........................................ viii
Numbers ........................................................................... ix
Rules-based writing ......................................................... x
Additional reading ......................................................... xi
Feedback .......................................................................... xii
Feedback on this book .................................................... xii
Progressive Terminology Commitment ............................. xii

Part B  AArch64 state exception model

Chapter B1  Exception levels

B1.1 Execution state ......................................................... 16
B1.2 Security state .......................................................... 17
B1.3 Effect of not implementing an Exception level .............. 18
B1.3.1 Behavior when EL3 is not implemented .................. 18
B1.3.2 Behavior when EL2 is not implemented ................. 18
B1.3.3 Behavior when EL2 is not implemented and EL3 is implemented .... 19

Chapter B2  Exceptions

B2.1 Exception entry terminology ....................................... 21
B2.1.1 Taken, taken from, and taken to .......................... 21
B2.1.2 Exception producing instructions ......................... 21
B2.1.3 Synchronous and asynchronous exceptions ........... 22
B2.1.4 Definition of a precise exception and imprecise exception 22
B2.1.5 Preferred exception return address ....................... 23
B2.1.6 Exception vectors .................................................. 23
B2.2 Exception entry ......................................................... 25
B2.2.1 Synchronous exception entry ............................... 25
B2.2.2 Asynchronous exception entry .......................... 26
B2.3 Exception return terminology .................................... 28
B2.3.1 Return, return from, return to ............................ 28
B2.4 Exception return ....................................................... 29
B2.4.1 Legal exception returns from AArch64 state .......... 29
B2.4.2 Illegal exception returns from AArch64 state ....... 29
B2.5 Synchronous exception types .................................... 32
B2.5.1 Taking synchronous exceptions from EL0 ............ 33
B2.5.2 Exception levels for taking a synchronous External abort 34
B2.5.3 Prioritization of Synchronous exceptions taken to AArch64 state .... 34
B2.5.4 Trapping of floating-point exceptions .................... 39
B2.6 Asynchronous exception types .................................... 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2.6.1</td>
<td>Virtual interrupts</td>
<td>42</td>
</tr>
<tr>
<td>B2.6.2</td>
<td>Establishing the target Exception level of an asynchronous exception</td>
<td>43</td>
</tr>
<tr>
<td>B2.6.3</td>
<td>Asynchronous exception masking</td>
<td>45</td>
</tr>
<tr>
<td>B2.6.4</td>
<td>Prioritization of interrupts</td>
<td>48</td>
</tr>
<tr>
<td>B2.6.5</td>
<td>Taking an interrupt during a multi-access load or store</td>
<td>49</td>
</tr>
<tr>
<td>B2.7</td>
<td>UNDEFINED instructions</td>
<td>50</td>
</tr>
<tr>
<td>B2.8</td>
<td>Configurable instruction controls</td>
<td>51</td>
</tr>
<tr>
<td>B2.8.1</td>
<td>EL0 and EL1 configurable instruction controls</td>
<td>52</td>
</tr>
<tr>
<td>B2.8.2</td>
<td>EL2 configurable instruction controls</td>
<td>52</td>
</tr>
<tr>
<td>B2.8.3</td>
<td>EL3 configurable instruction controls</td>
<td>53</td>
</tr>
<tr>
<td>B2.9</td>
<td>Exception Producing Instructions</td>
<td>54</td>
</tr>
<tr>
<td>B2.10</td>
<td>Program counter and stack pointer alignment</td>
<td>55</td>
</tr>
<tr>
<td>B2.10.1</td>
<td>PC alignment checking</td>
<td>55</td>
</tr>
<tr>
<td>B2.10.2</td>
<td>SP alignment checking</td>
<td>55</td>
</tr>
</tbody>
</table>
Part A
Preface
About this book

This is the Preview of Rules-based Arm® ARM D1 Exception Model chapter.

This preview includes content from the following chapters of the Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile rewritten in the rules-based writing style:

- Exception levels
- Exception terminology
- Execution state
- Security state
- Registers for instruction processing and Exception handling
- Program counter and stack pointer alignment
- Exception Entry
- Exception Return
- Synchronous Exception types, routing & priorities
- Asynchronous types, routing masking & priorities
- Configurable instruction enables and disables and trap controls
- System Calls
- The effect of implementation choices on Programmer’s model
Conventions

Typographical conventions

The typographical conventions are:

*italic*

Introduces special terminology, and denotes citations.

*bold*

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

*monospace*

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

*SMALL CAPITALS*

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Colored text

Indicates a link. This can be:

• A URL, for example http://developer.arm.com
• A cross-reference to another location within the document
• A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term.

*{ and }*

Braces, { and }, have two distinct uses:

**Optional items**

In syntax descriptions braces enclose optional items. In the following example they indicate that the <shift> parameter is optional:

'ADD <Wd\WSP>, <Wn\WSP>, #[{, }“

Similarly they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

**Sets of items**

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE

Notes

Notes are formatted as:
Note

This is a note.

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note may repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by 0x. In both cases, the prefix and the associated value are written in a monospace font, for example 0xFFFF0000. To improve readability, long numbers can be written with an underscore separator between every four characters, for example 0xFFFF_0000_0000_0000. Ignore any underscores when interpreting the value of a number.
**Rules-based writing**

Rules-based writing differs significantly from the traditional style used in Arm technical documents. Rules-based writing has short statements with unique identifiers.

Requirements are referred to as rules and other information is referred to as information statements. Rules-based architectural documents describe requirements of the architecture and information statements provide additional information. Structured rules also follow a specific structure and specify some keyword terms. The following is an example of a rule, followed by an information statement.

\[ R_{WPBT} \]
If a document has structured rules, by default all rules statements have a specific structure.

\[ I_{NBPP} \]
For architectural specifications the writing style for outcomes is deliberately different from the usual Arm style.

All rules and information statements have unique IDs. IDs start with a designator, followed by a unique string of 4 or 5 SMALL CAPITAL consonants. If the designator is an ‘R’ it is a rule. If the designator is an ‘I’, it is an information statement.

All rules have a specific structure. Information statements may take any structure.

Rules generally start with any conditions that make the rule applicable. These conditions have a limited set of introductory phrases:

- Conditions that begin with if are used to make rules conditional on a state and tend to last for a while.
- Conditions that begin with when are used to make rules conditional on an event happening.
- Conditions that begin with for are used to make a rule apply to a part of the system.

There are three forms of the if-statements:

- If indicates the condition is sufficient to cause the action but might not be necessary. “If X then Y” means the same as “If X then Y happens, but if not X then Y might still happen”.
- Only if indicates the condition is necessary but might not be sufficient. “Only if X then Y” means the same as “If X then Y might happen, if not X then Y cannot happen”.
- If and only if indicates the condition is both necessary and sufficient. “If and only if X then Y” means the same as “If X then Y happens, if not X then Y cannot happen”.

There are also three forms of the when-statements:

- When indicates the condition is sufficient to cause the action but might not be necessary.
- Only when indicates the condition is necessary but might not be sufficient.
- When and only when indicates the condition is both necessary and sufficient.

There may be many preconditions in a rule.

The next part of a rule is either an actor or a subject. When a specific action by a specific entity is defined, the rule will be written in active voice and will have an actor. If the action is performed by an IMPLEMENTATION DEFINED entity, then the rule will be written in passive voice and the rule will have a subject, which is something that is acted on by the action in the rule statement. The action to be carried out follows the actor or subject and is required. The object(s) of the action, followed by the outcome of the rule, are both optional and may be present after the action.
Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer (http://developer.arm.com) for access to Arm documentation.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title (Preview of Rules-based Arm® ARM D1 Exception Model).
- The number (AES0048 BETA).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Progressive Terminology Commitment

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

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com
Part B
AArch64 state exception model
Chapter B1
Exception levels

Describes rules for movement between Exception levels.

The architecture defines four *Exception levels*: EL0, EL1, EL2, and EL3.

EL3 is the highest *Exception level* and EL0 the lowest. Therefore, EL3 is higher than EL2, EL2 is higher than EL1, and EL1 is higher than EL0.

Unprivileged execution is any execution that occurs at EL0.

An implementation includes EL0 and EL1.

It is IMPLEMENTATION DEFINED whether EL2 is implemented, or not.

If implemented, EL2 provides support for the virtualization of EL0 and EL1.

It is IMPLEMENTATION DEFINED whether EL3 is implemented, or not.

A *PE* is not required to implement a contiguous set of *Exception levels*.

The current *Exception level* changes only when any of the following occur:

- Taking an exception.
- Returning from an exception.
- Processor reset.
- Exiting from *Debug state*.
- If in *Debug state*, executing a DCPSx instruction.
- If in *Debug state*, executing a DRPS instruction.

The “target *Exception level*” is the *Exception level* to which an exception is taken.
Chapter B1. Exception levels

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

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

An exception cannot be taken to EL0.

An exception cannot cause entry to a lower *Exception level*. 
The architecture defines two Execution states, AArch32 state and AArch64 state.

The architecture defines two Execution states. The AArch32 Execution state uses 32-bit processing. The AArch64 Execution state uses 64-bit processing.

AArch32 Execution state is compatible with Armv7-A operation.

An Exception level is described as:

- “Using AArch64” when execution in that Exception level is in the AArch64 Execution state.
- “Using AArch32” when execution in that Exception level is in the AArch32 Execution state.

The PE can only change Execution state at Reset, or when changing Exception level.

If an Exception level is using AArch32, then all lower Exception levels are using AArch32.

If an Exception level is using AArch64, then all higher Exception levels are using AArch64.

Interaction between the AArch64 and AArch32 Execution states is called “Interprocessing”.

In this chapter, the described behavior assumes the highest implemented Exception level is using AArch64.
B1.2 Security state

The PE physical address space that the PE can access is determined by the current Security state.

The architecture defines two Security states, called Secure state and Non-secure state.

The architecture defines two physical address spaces, Secure and Non-secure.

If in Non-secure state, the PE can only access the Non-secure physical address space.

If in Secure state, the PE can access both the Secure physical address space and the Non-secure physical address space.

EL3 is in Secure state.

The Security state of EL2, EL1, and EL0 is determined by the Effective value of the SCR_EL3.NS. If the Effective value of SCR_EL3.NS is 0, the PE is in Secure state. If the Effective value or SCR_EL3 is 1, the PE is in Non-secure state.

EL2 can only be in Secure state if all the following are true:

• FEAT_SEL2 is implemented.
• The Effective value of SCR_EL3.EEL2 is 0b1.
• The Effective value of SCR_EL3.NS is 0b0.

Otherwise, EL2 is in Non-secure state.

The state of SCR_EL3.NS can only change in EL3.
Chapter B1. Exception levels
B1.3. Effect of not implementing an Exception level

B1.3 Effect of not implementing an Exception level

The decision to exclude either EL2, EL3, or both EL2 and EL3 from an implementation affects the Execution states, Security state and virtualization.

An exception cannot be taken to an unimplemented Exception levels.

Any configurable instruction control that defines an exception to an unimplemented Exception level does not cause that exception to the unimplemented Exception level. The effective value of the configurable exception control is the value that does not cause an exception to that Exception level.

System calls made to an unimplemented Exception level from a lower Exception level are UNDEFINED.

The translation regime associated with an unimplemented Exception level is not implemented.

Any exception return that targets an unimplemented Exception level is an illegal return.

Every accessible register associated with an unimplemented Exception level is RES0 unless the register is associated with the Exception level only to provide the ability to transfer execution to a lower Exception level.

B1.3.1 Behavior when EL3 is not implemented

If EL3 is not implemented, the Effective value of SCR_EL3.NS still impacts behavior in an implementation.

If EL3 is not implemented, SCR_EL3.NS has a fixed Effective value that is IMPLEMENTATION DEFINED.

An implementation can provide a configuration input that determines the Effective value of SCR_EL3.NS at reset.

If EL3 is not implemented, and the Effective value of SCR_EL3.NS is 0b0, all of the following are true:

- The Effective value of MDCR_EL3.EPMAD is 0b1.
- The Effective value of MDCR_EL3.EDAD is 0b1.
- The Effective value of MDCR_EL3.SPME is 0b1.
- The Effective value of MDCR_EL3.NSPB is 0b01.
- The Effective value of MDCR_EL3.SPD32 is 0b11.
- The Effective value of MDCR_EL3.STE is 0b1.

If EL3 is not implemented, the Effective value of MDCR_EL3.STE is the inverse of the Effective value of SCR_EL3.NS.

If all of the following are true, the Effective value of SCR_EL3.EEL2 is 0b1:

- EL3 is not implemented.
- EL2 is implemented.
- The Effective value of SCR_EL3.NS is 0b0.

B1.3.2 Behavior when EL2 is not implemented

If EL2 is not implemented, virtual interrupts are disabled and the Effective value of certain bits are affected.

If EL2 is not implemented, all of the following are true:

- Virtual interrupts are disabled.
- The Effective value of CNTHCTL_EL2[1:0] is 0b11.
- The Effective value of HCR_EL2.E2H is 0b0.
Chapter B1. Exception levels

B1.3. Effect of not implementing an Exception level

• The **Effective value** of HCR_EL2.TGE is 0b0.

• The **Effective value** of MDCR_EL2.HPMN is the value of PMCR_EL0.N

B1.3.3 Behavior when EL2 is not implemented and EL3 is implemented

If EL2 is not implemented and EL3 is implemented, most EL2 registers are RES0 and some instructions are UNDEFINED.

If EL2 is not implemented and EL3 is implemented, all of the following are true:

• Except for any of the following registers, every accessible register associated with EL2 is RES0:
  - If EL1 can use AArch32 then the following registers are not RES0:
    » DACR32_EL2.
    » DBGVCR32_EL2.
    » FPEXC32_EL2.
    » IFSR32_EL2.
  - For VPIDR_EL2:
    » Reads return the value of MIDR_EL1.
    » Writes to VPIDR_EL2 are ignored.
  - For VMPIDR_EL2:
    » Reads return the value of MPIDR_EL1.
    » Writes to VMPIDR_EL2 are ignored.

• The **Effective value** of HCR_EL2.RW is the value of SCR_EL3.RW.

• SCR_EL3.HCE is RES0.

• The following Address translation and TLB invalidation instructions are UNDEFINED:
  - AT S1E2R.
  - AT S1E2W.
  - TLBI VAE2.
  - TLBI VALE2.
  - TLBI VAE2IS.
  - TLBI VALE2IS.
  - TLBI VAE2OS.
  - TLBI VALE2OS.
  - TLBI ALLE2.
  - TLBI ALLE2IS.
  - TLBI ALLE2OS.
  - TLBI RVAE2.
  - TLBI RVALE2.
  - TLBI RVAE2IS.
  - TLBI RVALE2IS.
Chapter B1. Exception levels

B1.3. Effect of not implementing an Exception level

- TLBI RVAE2OS.
- TLBI RVALE2OS.
Chapter B2
Exceptions

B2.1 Exception entry terminology

The following subsections define key terms used to describe exception entry.

B2.1.1 Taken, taken from, and taken to

When an exception is taken, it is taken from one Exception level and is taken to the same or a higher Exception level.

R\textsubscript{VDFHD} When the PE responds to an exception, an exception is “taken”.

R\textsubscript{FXGFX} The PE state immediately before taking the exception is the state the exception is “taken from”.

R\textsubscript{WTMN} The PE state immediately after taking the exception is the state the exception is “taken to”.

B2.1.2 Exception producing instructions

When executed, some instructions intentionally produce an exception.

R\textsubscript{JMKVX} Instructions that intentionally cause an exception to be taken are called exception producing instructions.

I\textsubscript{MGYJT} The following are exception producing instructions:

• HVC.

• SMC.
B2.1. Exception entry terminology

- SVC.

Exception producing instructions produce a precise exception at the point in the instruction stream immediately after the exception producing instruction.

B2.1.3 Synchronous and asynchronous exceptions

The architecture defines two types of exception: synchronous and asynchronous.

If all of the following apply, an exception is “synchronous”:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The preferred exception return address has an architecturally defined relationship with the instruction that caused the exception.
- The exception is precise.

An exception is “asynchronous” if it is not synchronous.

Asynchronous exceptions take to AArch64 state are also known as interrupts.

B2.1.4 Definition of a precise exception and imprecise exception

Exceptions, both synchronous and asynchronous, are classified as either precise or imprecise.

An exception is “precise” if on taking the exception, the PE state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. However, for an instruction executing immediately at the point in the instruction stream that the exception was taken from, the definition of precise also permits any of the following:

- For synchronous Data Abort and Watchpoint exceptions that are taken to AArch64 state generated by an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.

- For synchronous Data Abort and Watchpoint exceptions that are generated from load or store instructions executed in AArch64 state, all the following can occur:
  - If the instruction was a load to either the base address register or the offset register, that register is restored to the original value, and any other destination registers become UNKNOWN.
  - If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.

- For implementations that include synchronous exception generation for floating-point exceptions, when a floating-point exception is taken, it is permitted that the cumulative floating-point exception bits are not restored.

- For a synchronous exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.

An exception is “imprecise” if it is not a precise exception.
Chapter B2. Exceptions
B2.1. Exception entry terminology

Except for SError interrupts, all exceptions taken to AArch64 state are precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION_DEFINED.

An asynchronous Data Abort exception generated by a load that causes an SError exception to be taken at some point later in the instruction stream than the load that generated the exception, is usually imprecise. The SError exception is usually imprecise because the data returned from the load is UNKNOWN, and so can have corrupted the state that is presented at the time that the exception is taken.

B2.1.5 Preferred exception return address

For each exception, a “preferred exception return address” is stored in the Exception Link Register, ELR_ELx.

The “preferred exception return address” is an address that software might return to after handling an exception in order to resume execution.

The preferred exception return address is determined by the type of the exception.

For an exception taken to an Exception level, ELx, using AArch64, the Exception Link Register for that Exception level, ELR_ELx, is set to the preferred exception return address.

For asynchronous exceptions, the preferred exception return address is the address of the instruction following the instruction boundary at which the interrupt occurs.

For synchronous exceptions other than exception producing instructions, the preferred exception return address is the address of the instruction that generates the exception.

For an exception producing instruction that is executed, the preferred exception return address is the address of the instruction that follows the exception producing instruction.

For an exception producing instruction that is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the exception producing instruction.

When an exception is taken from an Exception level using AArch32 to an Exception level, ELx, using AArch64, ELR_ELx[63:32] are 0.

B2.1.6 Exception vectors

When an exception is taken to an Exception level using AArch64, the exception vector is the address for that exception from which execution starts.

When an exception is taken to an Exception level that is using AArch64, execution starts at the “exception vector”.

The memory at the exception vector for an exception is expected to contain the handler code that corresponds to that exception category.

The vector base address for an Exception level, ELx, is defined by the Vector Base Address Register, VBAR_ELx.

Each exception category has an exception vector at a fixed offset from the vector base address.

Exceptions taken to AArch64 are categorized, based on the following information, for the purpose of assigning exception vectors:

- Whether the exception is one of the following:
  - Synchronous exception.
  - SError.
  - IRQ.
  - FIQ.
• Information about the *Exception level* that the exception came from, combined with information about the SP in use, and the state of the register file.

The following table describes the offsets that are added to the vector base address to describe the exception vector:

<table>
<thead>
<tr>
<th>Exception taken from</th>
<th>Synchronous (including synchronous External aborts if SCR_EL3.EASE is 0)</th>
<th>Offset for exception type</th>
<th>Synchronous External aborts if SCR_EL3.EASE is 1</th>
<th>IRQ or vIRQ</th>
<th>FIQ or vFIQ</th>
<th>SEError or vSEError</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Exception level with SP_EL0.</td>
<td>0x000</td>
<td>0x080</td>
<td>0x100</td>
<td>0x180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current Exception level with SP_ELx, x&gt;0.</td>
<td>0x200</td>
<td>0x280</td>
<td>0x300</td>
<td>0x380</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.</td>
<td>0x400</td>
<td>0x480</td>
<td>0x500</td>
<td>0x580</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.</td>
<td>0x600</td>
<td>0x680</td>
<td>0x700</td>
<td>0x780</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the preceding table, the implemented *Exception level* immediately lower than the target *Exception level* is defined by **RFSZTB**.

If EL3 is implemented, the *Exception level* immediately lower than the target *Exception level* is determined by whether the exception was taken from an *Exception level* in Secure or Non-secure state.

For an exception taken to EL3 from a lower *Exception level*, if EL2 is implemented and enabled for the security state of *Exception level* the exception was taken from, the *Exception level* immediately lower than EL3 is EL2.

For an exception taken to EL3 from a lower *Exception level*, if EL2 is not implemented and enabled for the security state of *Exception level* the exception was taken from, the *Exception level* immediately lower than EL3 is EL1.
Chapter B2. Exceptions

B2.2. Exception entry

When an exception is taken to AArch64 state, a series of actions occur in a specific order to handle 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 return address is written to ELR_ELx.
- The contents of PSTATE immediately after the exception is taken is as described in rule RWTXBY.
- For synchronous exceptions and SError interrupts, exception syndrome information is written to ESR_ELx.
- Execution starts from the exception vector at the target Exception level.

When an exception is taken to an Exception level, ELx, that is using AArch64 state, after recording the previous values in SPSR_ELx, the following PSTATE bits are set:

- PSTATE.EL is set to the target Exception level.
- All of PSTATE.{D, A, I, F, SP, TCO} are set to 1.
- PSTATE.SSBS is set to the value of SCTLR_ELx.DSSBS.
- PSTATE.{IL, nRw, UAO} are set to 0.
- PSTATE.BTYPE is set to 0x00
- PSTATE.SS is set according to the rules in AArch64 Self-hosted Debug[1].
- For any of the following situations, PSTATE.PAN is set to 1:
  - The target Exception level is EL1 and SCTLR_EL1.SPAN is 0.
  - The exception is taken from EL0, the target Exception level is EL2 using AArch64, Secure EL2 is enabled, HCR_EL2.{TGE,E2H} is {1,1}, and SCTLR_EL2.SPAN is 0.

If the Effective value of SCTLR_ELx.IESB is 1 at the target Exception level, an exception taken to AArch64 state is an error synchronization event.

If SCTLR_ELx.EIS is 1, exception entry is a context synchronization event. If SCTLR_ELx.EIS is 0, exception entry is not a context synchronization event, but the indirect writes to ESR_ELx, FAR_ELx, SPSR_ELx, ELR_ELx, and HPFAR_EL2 due to exception entry are synchronized so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.

The ordering of error synchronization and context synchronization on exception entry is described in the pseudocode.

B2.2.1 Synchronous exception entry

For some types of synchronous exceptions, when an exception is taken to AArch64 state, there are actions that occur in addition to those described in the Exception entry section.

For any of the following synchronous exceptions taken from an Exception level using AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0:

- Software Step exception.
- PC Alignment Fault exception.
- Instruction Abort exception.
- Breakpoint exception.
- Address Matching Vector Catch exception.
Chapter B2. Exceptions

B2.2. Exception entry

- Illegal Execution state exception.
- Software Breakpoint exception.
- Branch Target exception.

When taking any other synchronous exception from an Exception level using AArch64, it is constrained unpredictable whether:

- SPSR_ELx.BTYPE is set to the value of PSTATE.BTYPE.
- SPSR_ELx.BTYPE is set to 0.

PSTATE.BTYPE is then set to 0.

For any of the following synchronous exception types, when an exception is taken to an Exception level, ELx, a virtual address that characterizes the exception is captured in FAR_ELx:

- An Instruction Abort exception.
- A Data Abort exception.
- A PC alignment fault exception.
- A Watchpoint exception.

For synchronous External aborts that are not caused by translation table walks, it is constrained unpredictable whether the FAR_ELx contains a virtual address that characterizes the exception. The ESR_ELx.FnV bit in the ISS encoding for the External abort indicates the validity of the virtual address in FAR_ELx:

- If ESR_ELx.FnV in the ISS encoding for the External abort is 1, the Faulting Virtual Address in FAR_ELx is UNKNOWN.
- If ESR_ELx.FnV in the ISS encoding for the External abort is 0, the Faulting Virtual Address in FAR_ELx is valid.

For all other exceptions taken to ELx, FAR_ELx is UNKNOWN.

For Instruction Abort or Data Abort exceptions caused by any of the following faults, when an exception is taken to EL2, an intermediate physical address that characterizes the exception is captured in HPFAR_EL2:

- A Translation fault on a stage 2 translation.
- An Access Flag fault on a stage 2 translation.
- A stage 2 Address Size fault.
- A fault on stage 2 translation of an address accessed in a stage 1 translation table walk.

For all other exceptions taken to EL2 using AArch64, HPFAR_EL2 is UNKNOWN.

B2.2.2 Asynchronous exception entry

For some types of asynchronous exceptions, when an exception is taken to AArch64 state, there are actions that occur in addition to those described in the Exception entry section.

When an asynchronous exception is taken from an Exception level using AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0.

If any of the following apply, when a physical SError interrupt exception is taken to AArch64 state, the pending state of the physical SError is cleared:

- FEAT_DoubleFault is implemented.
• If the Reliability, Availability, and Serviceability Extension (RAS Extension) is implemented, and on taking the SError interrupt, the syndrome recorded in ELR_ELx indicates an SError other than IMPLEMENTATION DEFINED or uncategorized SError interrupt syndrome.

Otherwise, it is IMPLEMENTATION DEFINED whether the pending state of the physical SError is cleared. This IMPLEMENTATION DEFINED behavior might vary according to the type of the SError interrupt.

When a virtual SError interrupt exception is taken to AArch64 state, HCR_EL2.VSE is set to 0.
B2.3 Exception return terminology

The following subsection define key terms used to describe exception returns.

B2.3.1 Return, return from, return to

When an exception return occurs, it returns from one Exception level and might return to the same or a lower Exception level.

A exception return is caused by the execution of an exception return instruction.

The PE state immediately before an exception return instruction is executed is the state the exception “returns from”.

The PE state immediately after the execution of an exception return instruction is the state the exception “returns to”.

Chapter B2. Exceptions
B2.4. Exception return

B2.4 Exception return

Exception returns are caused by executing an ERET instruction. Exception returns are always to the same or a lower Exception level.

In AArch64 state, an ERET instruction causes an exception return.

The ERET instruction is UNDEFINED in EL0.

An exception return is either legal or illegal.

B2.4.1 Legal exception returns from AArch64 state

On a legal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, the PE inserts an error synchronization event before the ERET instruction.
- The PC is set to the value in ELR_ELx.
- If returning to an Exception level using AArch32 state, all of the following apply:
  - If SPSR_ELx.T is 0, ELR_ELx[1:0] are treated as being 0 for setting the PC.
  - If SPSR_ELx.T is 1, ELR_ELx[0] is treated as being 0 for setting the PC.
- The contents of PSTATE are set to the values held in SPSR_ELx.
- If the PSTATE.IL bit copied from SPSR_ELx is 1 and the target Exception level for the return is using AArch32 state, the PSTATE.{IT,T} bits are determined by an IMPLEMENTATION DEFINED choice of one of the following:
  - Set to 0.
  - For returns initiated by an ERET or DRPS instruction, copied from SPSR_ELx.
  - For debug exits, copied from DSPSR_EL0.

The IMPLEMENTATION DEFINED choice might vary dynamically within the implementation. Software must regard the value as being an UNKNOWN choice between the two values.

- The Event Register for the PE executing the ERET instruction is set.
- The local Exclusives monitor for the PE executing the ERET instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.
- After the PC is set to the value held in ELR_ELx and the contents of PSTATE are set to the values held in SPSR_ELx, ELR_ELx and SPSR_ELx become UNKNOWN.
- If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a context synchronization event.
- If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a context synchronization event.

B2.4.2 Illegal exception returns from AArch64 state

If an exception attempts to return to an illegal mode or state, an illegal exception return is generated.

If in AArch64 state, any of the following situations can cause an illegal exception return:

- A return is made to an Exception level higher than the current Exception level.
- A return is made to an Exception level that is not implemented.
- If all of the following are true, and a return is made to EL1:
– EL2 is implemented and enabled in the current Security state
– HCR_EL2.TGE is 1.

• If all of the following are true, and a return is made to EL2:
  – EL3 is implemented.
  – SCR_EL3.NS is 0.
  – FEAT_SEL2 is not implemented.

• If all of the following are true, and a return is made to EL2:
  – EL3 is implemented.
  – SCR_EL3.NS is 0.
  – FEAT_SEL2 is implemented.
  – SCR_EL3.EEL2 is 0.

• A return where the saved PSTATE.M[4] is 0 and at least one of the following is true:
  – M[1] is 1.
  – M[3:0] are 0b0001.
  – The Exception level being returned to is using AArch32 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.

• A return where the saved PSTATE.M[4] bit is 1, indicating a return to AArch32 state, and at least one of the following are true:
  – AArch32 state is not supported at any Exception level.
  – The M field value is not a valid AArch32 state PE mode.
  – The Exception level being returned to is using AArch64 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.

On an illegal exception return from an Exception level, ELx, all of the following occur:

• If SCTLR_ELx.IESB is 1, the PE inserts an error synchronization event before the ERET instruction.
• The PC is set to the value in ELR_ELx. If the saved PSTATE.M[4] bit is 1, for illegal exception returns to AArch32 state, all of the following are true:
  – The PC bits[31:2] are set to the value in ELR_ELx.
  – The PC bits[63:32, 1:0] are UNKNOWN.
• PSTATE.IL is set to 1.
• PSTATE.{EL, nRW, SP} are unchanged.
• PSTATE.{ N, Z, C, V, D, A, I, F, PAN} are set to the associated values in SPSR_ELx.
• PSTATE.SS is handled as if the return is a legal exception return.
• PSTATE.{TCO, DIT, UAO, SSBS, BTYPE} are UNKNOWN.
• If PSTATE.nRW is 1, indicating a return to AArch32 state, then the following PSTATE bits are also set:
  – PSTATE.{ Q, IT, GE, E} are set to the associated values in SPSR_ELx.
  – It is constrained unpredictable whether PSTATE.T is 0 or set to the contents of SPSR_ELx.
• The Event Register for the PE executing the ERET instruction is set.
• The local Exclusives monitor for the PE executing the ERET instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.

• After the PC has been set to the value held in ELR_ELx and the contents of PSTATE have been set to the value held in SPSR_ELx, the ELR_ELx and SPSR_ELx become UNKNOWN.

• If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a context synchronization event.

• If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a context synchronization event.
B2.5 Synchronous exception types

Synchronous exceptions can be referred to by specific names. The names are determined by what generated the exception.

All of the following are Synchronous exceptions:

- Any exception generated by attempting to execute an instruction that is UNDEFINED, including:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instructions when they are disabled.
  - Attempts to execute instruction bit patterns that are allocated.

  These are called Undefined Instruction exceptions.

- Any exception caused by attempts to execute an instruction when the value of PSTATE.IL is 1. These are called Illegal Execution State exceptions.

- Any exception caused by the use of a misaligned SP. These are called SP Alignment Fault exceptions.

- Any exception caused by attempting to execute an instruction with a misaligned PC. These are called PC Alignment Fault exceptions.

- If executing inside a guarded memory region and PSTATE.BTYPE does not equal 0, any exception caused by executing an instruction that is not compatible with the current value of PSTATE.BTYPE. These are called Branch Target exceptions.

- Any exception caused by a pointer authentication instruction authentication failure. These are called PAC Fail exceptions.

- Any exception caused by the exception producing instructions SVC, HVC, or SMC. These are respectively called Supervisor Call (SVC) exceptions, Hypervisor Call (HVC) exceptions, or Secure Monitor Call (SMC) exceptions.

- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. These are called Trap exceptions.

- Any exception caused by Instruction Aborts that were generated by the memory address translation system that was associated with attempts to execute instructions from areas of memory that generate faults. These are called Instruction Abort exceptions.

- Any exception caused by Data Aborts that were generated by the memory address translation system that are associated with attempts to read or write memory that generate faults. These are called Data Abort exceptions.

- Any exception caused by Data Aborts because of a misaligned address. These are called Data Abort exceptions.

- If FEAT_MTE is implemented, any exception caused by a Data Abort as a result of a Tag Check Fault. These are called Data Abort exceptions.

- All of the debug exceptions:
  - Breakpoint Instruction exceptions.
  - Breakpoint exceptions.
  - Watchpoint exceptions.
  - Vector Catch exceptions.
  - Software Step exceptions.
• In an implementation that supports the trapping of floating-point exceptions, any exception caused by trapped floating-point exceptions. These are called Trapped Floating-point exceptions.
• An IMPLEMENTATION DEFINED defined set of exceptions caused by External aborts.

The architecture permits, but does not require, synchronous or asynchronous handling of External aborts.

The RAS architecture specifies some situations that must be handled synchronously.

### B2.5.1 Taking synchronous exceptions from EL0

If EL2 is implemented and enabled, some synchronous exceptions taken from EL0 can be taken to either EL1 or EL2 depending on the value of HCR_EL2.TGE and MDCR_EL2.TDE.

If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 1, when any of the following exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 0, when any of the following exceptions are taken from EL0, they are taken to EL1:

• Stage 1 Data Abort.
• Stage 1 Instruction Abort.
• PC Alignment Fault.
• SP Alignment Fault.
• Tag Check Fault.
• Branch Target Exception.
• Illegal Execution State Exception.
• Trapped Floating-point Exception.
• Supervisor Monitor Call.
• Undefined Instruction Exception.
• SVE Access Trap.
• PAC Fail Exception.
• WFE Trap.
• WFI Trap.
• Advanced SIMD and Floating-point access Trap.
• Synchronous External Aborts.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when any of the following debug exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, when any of the following debug exceptions are taken from EL0, they are taken to EL1:

• Breakpoint Exception.
• Software Breakpoint Exception.
• Software Step Exception.
• Watchpoint Exception.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when a Vector Catch exception is taken from EL0, it is taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, a Vector Catch exception cannot be taken.
For an exception that is taken from EL0 to EL2 because the value of HCR_EL2.TGE is 1:

- If the exception would have been reported in ESR_EL1 using any EC value other than 0x07, then the EC value and corresponding ISS encoding that would have been used to report the exception in ESR_EL1 are used to report the exception in ESR_EL2.
- If the exception would have been reported in ESR_EL1 using the EC value 0x07, then the EC value 0x00 and ISS encoding value 0x00 are reported in ESR_EL2.

**B2.5.2 Exception levels for taking a synchronous External abort**

Synchronous external aborts taken from EL0 and EL1 that would otherwise be taken to EL1, can be taken to EL2 or EL3 in specific circumstances.

A synchronous external abort is taken to EL1 if executing at EL0 or EL1 and none of the following scenarios cause the synchronous external abort to be taken to EL2 or EL3.

A synchronous external abort is taken to EL2 only if all of the following are true:

- SCR_EL3.EA is 0.
- EL2 is enabled and implemented in the current Security state.
- Execution is at an Exception level below EL3.
- If any of the following is true:
  - HCR_EL2.TEA is 1.
  - HCR_EL2.TGE is 1.
  - Execution is at EL2.

A synchronous external abort is taken to EL3 if any of the following are true:

- If executing at any Exception level and SCR_EL3.EA is 0b1.
- If executing at EL3.

**B2.5.3 Prioritization of Synchronous exceptions taken to AArch64 state**

Between the fetching of an instruction, its decode and its eventual execution, a single instruction might generate a number of different synchronous exceptions. The following list illustrates the inherent priority of each type of Armv8-A synchronous exception.

The following list shows the priorities for Synchronous exceptions taken to an Exception level using AArch64. The highest priority is 1.

The AArch64 Priority numbers correlate with the equivalent AArch32 and Debug prioritization lists.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Synchronous exception type</th>
<th>Further information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset Catch debug event.</td>
<td>Reset Catch debug events[1]</td>
</tr>
<tr>
<td>2</td>
<td>Exception Catch debug event if it has a priority of 2.</td>
<td>Exception Catch debug event[1]</td>
</tr>
<tr>
<td>3</td>
<td>Halting Step debug event.</td>
<td>Halting Step debug events[1]</td>
</tr>
<tr>
<td>4</td>
<td>Software Step Exception used during Debug</td>
<td>Software Step exceptions[1]</td>
</tr>
<tr>
<td>5</td>
<td>Exception Catch debug events</td>
<td>Exception Catch debug event[1]</td>
</tr>
<tr>
<td>6</td>
<td>PC alignment fault exception</td>
<td>B2.10.1 PC alignment checking</td>
</tr>
</tbody>
</table>
## Chapter B2. Exceptions

### B2.5. Synchronous exception types

<table>
<thead>
<tr>
<th>Priority</th>
<th>Synchronous exception type</th>
<th>Further information</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Instruction Abort exceptions, including Exceptions generated by a Translation Table Walk not prioritized as 29.</td>
<td>AArch64 state prioritization of synchronous aborts from a single stage of address translation[1]</td>
</tr>
<tr>
<td>8</td>
<td>Breakpoint exceptions, or AArch32 Address Matching Vector Catch exceptions.</td>
<td>• Breakpoint exceptions[1]</td>
</tr>
<tr>
<td>9</td>
<td>Illegal Execution state exceptions.</td>
<td>B2.4.2 Illegal exception returns from AArch64 state</td>
</tr>
<tr>
<td>10</td>
<td>Software Breakpoint exceptions, caused by execution of a Breakpoint instruction.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Target exception</td>
<td>About PSTATE.BTYPE[1]</td>
</tr>
<tr>
<td>12</td>
<td>Exceptions taken from EL1 to EL2 because of the configuration of one of the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• HSTR_EL2.Tn</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• HCR_EL2.TIDCP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• If ARMv8.3-NV is implemented:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– HCR_EL2.NV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– HCR_EL2.NV1</td>
<td></td>
</tr>
</tbody>
</table>
 Exceptions that occur as a result of attempting to execute an instruction that is \textit{UNDEFINED} for one or more of the following reasons:

\begin{itemize}
  \item Attempting to execute a UDF instruction.
  \item Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the \textit{PE} implementation.
  \item Attempting to execute an instruction that is defined to never be accessible at the current \textit{Exception level}, regardless of any enables or traps.
  \item Debug state execution of an instruction encoding that is unallocated in Debug state.
  \item Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
  \item If HVC instructions are disabled by SCR\_EL3.HCE or HCR\_EL2.HCD, attempting to execute an HVC instruction.
  \item If SPSel.SP is 0, attempting to execute an MSR or MRS instruction to SP\_EL0.
  \item If HLT instructions are disabled by EDSCR.HDE or halting is prohibited, attempting to execute an HLT instruction.
  \item If in Debug state:
    \begin{itemize}
      \item If HCR\_EL2.TGE is 1, attempting to execute a DCPS1 instruction in Non-secure EL0.
      \item If EL2 is disabled in the current Security state or is not implemented, attempting to execute a DCPS2 instruction in EL1 or EL0.
      \item If EDSCR.SDD is 1 or EL3 is not implemented, attempting to execute a DCPS3 instruction.
      \item If EDSCR.SDD is 1, attempting to execute in EL2, EL1, or EL0 an instruction that is configured by EL3 control registers to trap to EL3. It is \textit{IMPLEMENTATION DEFINED} whether this type of exception is prioritized at this level or has the priority of the original trap exception.
    \end{itemize}
  \item If in AArch32 state:
    \begin{itemize}
      \item If SCTLR\_EL1.ITD is 1, attempting to execute an instruction in an IT block.
      \item If SCTLR\_EL1.SED is 1, attempting to execute a SETEND instructions.
      \item If SCTLR\_EL1.CP15BEN is 0, attempting to execute a CP15DMB, CP15DSB, or CP15ISB barrier instruction.
    \end{itemize}
  \item Attempting to execute an instruction that is \textit{UNDEFINED} because all of the following apply:
    \begin{itemize}
      \item EL0 is using AArch32.
      \item EL1 is using AArch64.
      \item Programming FPCR\.[Stride, Len] to nonzero values is supported.
      \item One or more of FPCR\.[Stride, Len] is nonzero.
    \end{itemize}
\end{itemize}
## Chapter B2. Exceptions

### B2.5. Synchronous exception types

<table>
<thead>
<tr>
<th>Priority</th>
<th>Synchronous exception type</th>
<th>Further information</th>
</tr>
</thead>
</table>
| 14       | If all of the following apply, any exception that is generated by a configurable trap or enable:  
• The exception is not already covered by priorities 4-13,  
• The exception is taken to EL1, or if HCR_EL2.TGE is 1, is taken to EL2. | |
| 15       | As the result of a configuration control in one of the following registers, an exception taken from EL0 to EL2:  
• When the exception is taken to AArch64 state:  
  – HSTR_EL2:Tn  
  – HCR_EL2:TIDCP  
• When the exception is taken to AArch32 state:  
  – HSTR.Tn  
  – HCR.TIDCP | |
| 16       | As the result of a configuration control in one of the following registers, an exception taken to EL2:  
• When the exception is taken to AArch64 state: CPTR_EL2.  
• When the exception is taken to AArch32 state: in HCPTR. | |
| 17       | As the result of a configuration control in one of the following registers, an exception taken to EL2:  
• When the exception is taken to AArch64 state:  
  – HCR_EL2: Other than a setting made in the {TIDCP, NV} fields.  
  – CNTHCTL_EL2: Any setting.  
  – MDCR_EL2: Any setting.  
  – Any of the following fine-grained traps:  
    * HAFGRTR_EL2  
    * HDFGRTR_EL2  
    * HDFGWTR_EL2  
    * HFGITR_EL2  
    * HFGRTR_EL2  
    * HFGWTR_EL2  
• When the exception is taken to AArch32 state:  
  – HCR : Other than the TIDCP bit.  
  – CNTHCTL: Any setting.  
  – HDCR: Any setting.  
  – When EL1 is using AArch64 state, any of the following fine-grained traps:  
    * HAFGRTR_EL2  
    * HDFGRTR_EL2  
    * HDFGWTR_EL2  
    * HFGITR_EL2  
    * HFGRTR_EL2  
    * HFGWTR_EL2 | |
| 18       | Other than an exception defined by priorities 4-17 inclusive, any exception that is the result of a configurable access to instructions, and the exception is taken to EL2. | |
### Chapter B2. Exceptions

#### B2.5. Synchronous exception types

<table>
<thead>
<tr>
<th>Priority</th>
<th>Synchronous exception type</th>
<th>Further information</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>An exception caused by the SMC instruction being UNDEFINED because SCR_EL3.SMD is 1.</td>
<td>For AArch64 State, see Branches, Exception generating, and System instructions[1]</td>
</tr>
</tbody>
</table>
| 20       | An exception caused by any of the following Exception producing instructions:  
- HVC  
- SMC  
- SVC | Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32[1] |
| 21       | An exception taken to EL3 as the result of configuration controls in CPTPR_EL3.  
It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception. |  |
| 22       | Exceptions Trapped to EL3 from Secure EL1 using AArch32. |  |
| 23       | An Exception taken to EL3 from EL0, EL1, or EL2 as a result of configuration controls in MDCR_EL3.  
It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception. |  |
| 24       | Other than an exception defined by priorities 4-23, inclusive, any exception taken to EL3 because of a configurable access to an instruction.  
It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception. |  |
| 25       | When ARMv8.3-FPAC is implemented, an exception generated by the failure to authorize a Pointer Authentication instruction. | Faulting on pointer authentication[1]. |
| 26       | When SIMD & Floating Point is implemented, any trapped floating-point exception. | B2.5.4 Trapping of floating-point exceptions |
| 27       | Debug events. |  |
| 28       | An SP alignment fault. | B2.10.2 SP alignment checking |
| 29       | An External abort generated by one of the following Data Abort Exceptions:  
- Any Translation Table walk not prioritized as 7  
- A Page Table entry update  
- When ARMv8.5-MemTag is implemented, any Tag Check Fault.  
It is IMPLEMENTATION DEFINED whether synchronous External Aborts are prioritized here or as Priority 31. | AArch64 state prioritization of synchronous aborts from a single stage of address translation[1] |
| 30       | Watchpoint exception | Watchpoint exceptions[1] |
| 31       | Any Data Abort Exception not defined by Priority 29.  
It is IMPLEMENTATION DEFINED whether synchronous External Aborts are prioritized here or as Priority 29. | • External aborts[1]  
• PE handling of Tag Check Fault*[1] |

For Priorities 29, 30 or 31: When an instruction results in more than one single-copy atomic memory access, the architecture will not prioritize the individual Synchronous exceptions generated as the result of the multiple memory accesses.

---

AES0048  Copyright © 2021 Arm Limited or its affiliates. All rights reserved.  
BETA  Non-confidential
B2.5 Synchronous exception types

B2.5.4 Trapping of floating-point exceptions

Floating-point, SVE and some Advanced SIMD instructions can generate floating-point exceptions. It is IMPLEMENTATION DEFINED whether floating-point exceptions also generate synchronous exceptions.

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a floating-point exception.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

The Armv8-A architecture does not support asynchronous reporting of floating-point exceptions.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements on when that synchronous exception is generated do not apply.

For any of the implemented floating-point exceptions listed in R_{BBSGN}, FPCR provides control bits to enable synchronous exception generation.

The Exception level that a Trapped Floating-point exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

If an implementation includes synchronous exception generation for floating-point exceptions in AArch64 state, when the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception.

  When the trapped floating-point exception is taken, if the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative floating-point exception bits having the value 1.

- In ESR_ELx at the target Exception level, all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
Chapter B2. Exceptions
B2.5. Synchronous exception types

- If any other untrapped floating-point exceptions are generated by the same operation, each untrusted exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.

- If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

For trapped floating-point exceptions from Advanced SIMD instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.
### Chapter B2. Exceptions

#### B2.6 Asynchronous exception types

Asynchronous exceptions taken to AArch64 are also called interrupts. The Arm architecture supports physical and virtual interrupts.

The Arm architecture exception model distinguishes two classes of asynchronous exceptions:

- **Physical Interrupts**
- **Virtual Interrupts**

There are three types of physical interrupt:

- **SError** (also described as a System Error).
- **IRQ**.
- **FIQ**.

There are three types of virtual interrupt:

- **vSError** (also described as a Virtual System Error).
- **vIRQ**.
- **vFIQ**.

IRQ, FIQ, vIRQ and vFIQ interrupts are precise asynchronous exceptions.

Each physical interrupt type can be assigned a target *Exception level* of EL1, EL2, or EL3.

ISR_EL1 shows the pending status of interrupts as follows:

- When read from EL2 or EL3, ISR_EL1 shows the pending status of each of the physical interrupts IRQ, FIQ, and SError.
- When read from EL1, ISR_EL1 shows:
  - The pending status of the virtual interrupts, vIRQ, vFIQ, and vSError if they are enabled by the *Effective values* of the corresponding HCR_EL2.[IMO, FMO, AMO] enables.
  - The pending status of the physical interrupts, IRQ, FIQ, and SError if the corresponding virtual interrupts are not enabled by the *Effective values* of the HCR_EL2.[IMO, FMO, AMO] enables.

An implementation might support other mechanisms for signaling a virtual interrupt.

The mechanism by which physical interrupts are signaled to the *PE* are *IMPLEMENTATION DEFINED* and might be either edge or level sensitive. A common implementation choice is that the IRQ and FIQ interrupts are level sensitive, and this is supported by the Generic Interrupt Controller (GIC).

The physical SError interrupt is often used, amongst other things, for communicating External aborts from the memory system that are to be taken asynchronously.

For an External abort generated by the memory system that is taken asynchronously using the SError interrupt, the SError interrupt always behaves as an edge-triggered interrupt. For any other sources of SError interrupts, it is *IMPLEMENTATION DEFINED* whether they are edge-triggered or level-sensitive.

When taking an SError or a vSError interrupt to an *Exception level* using AArch64, ESR_ELx at the target *Exception level* is updated to describe an SError interrupt. If the RAS extension is implemented, when taking a vSError interrupt to an *Exception level* using AArch64, ESR_ELx at the target *Exception level* is updated with exception syndrome information from VSESR_EL2.

When taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an *Exception level* using AArch64, ESR_ELx at the target *Exception level* is not updated.
B2.6.1 Virtual interrupts

HCR_EL2 contains control bits that enable virtual interrupts and additional control bits to cause virtual interrupts to be pending.

Each virtual interrupt type can be independently enabled from EL2. If a virtual interrupt type is enabled from EL2, the target Exception level for the corresponding physical interrupt is not EL1.

Each virtual interrupt type can be set to the pending state by EL2 using controls in HCR_EL2.

If HCR_EL2.TGE is 0, setting an HCR_EL2.{FMO, IMO, AMO} routing control bit to 1 enables the corresponding virtual interrupt. If HCR_EL2.TGE is 1, all virtual interrupts are disabled and the Effective values of HCR_EL2{FMO, IMO, AMO} are 0.

If a virtual interrupt type is enabled, that type of interrupt can be generated by any one of the following:

- Execution at EL0 or EL1 if the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, is 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller.

If a virtual interrupt is disabled, the virtual interrupt cannot be taken.

The following table describes the bits that enable virtual interrupts and the bits that cause virtual interrupts to be pending in HCR_EL2:

<table>
<thead>
<tr>
<th>Virtual interrupt type</th>
<th>Enable control</th>
<th>Cause a virtual interrupt to be pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>vSError</td>
<td>HCR_EL2.AMO</td>
<td>HCR_EL2.VSE</td>
</tr>
<tr>
<td>vIRQ</td>
<td>HCR_EL2.IMO</td>
<td>HCR_EL2.VI</td>
</tr>
<tr>
<td>vFIQ</td>
<td>HCR_EL2.FMO</td>
<td>HCR_EL2.VF</td>
</tr>
</tbody>
</table>

When taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in HCR_EL2 retains its state.

If the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing at EL2 or EL3 to set their corresponding virtual interrupt pending bits to 0.

When taking a vSError interrupt, HCR_EL2.VSE is cleared to 0.
B2.6.2 Establishing the target Exception level of an asynchronous exception

The target Exception level of physical interrupts is affected by the combination of implemented Exception levels, the configuration of the Hypervisor Configuration Register, and the configuration of the Secure Configuration Register.

The terms used in the table in this section have the following meanings:

- **SCR_EL3**: The Effective value of a field in SCR_EL3.
- **EEL2**: If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1.
- **FIQ IRQ EA**: The Effective value of the field that configures the asynchronous exception type in SCR_EL3. The Effective value of the field is one of the following:
  - If EL3 is implemented, the FIQ/IRQ/EA fields are taken from SCR_EL3.
  - If EL3 is not implemented, the Effective value of these fields is 0.
- **RW**: If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.
- **HCR**: If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.
- **TGE**: If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.
- **FMO IMO AMO**: The Effective value of the mask override field for the asynchronous exception type in HCR or HCR_EL2. The Effective value of the field is one of the following:
  - If EL2 is implemented, the FMO/IMO/AMO fields are taken from HCR_EL2.
  - If EL2 is not implemented, the Effective value of these fields is 0.
- **E2H**: If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.
- **RW**: If EL2 is not implemented, the Effective value of HCR_EL2.RW is the same as the Effective value of SCR_EL3.RW.
- **EL1**: The exception is taken to EL1 using AArch64.
- **EL2**: The exception is taken to EL2 using AArch64.
- **EL3**: The exception is taken to EL3 using AArch64.
- **C**: The interrupt is not taken and remains pending, regardless of the PSTATE.{A, I, F} interrupt masks.
- **FIQ IRQ Abt**: The exception is taken to the AArch32 FIQ mode, the AArch32 IRQ mode or the AArch32 Abort mode according to the type of asynchronous exception.
- **Hyp**: The exception is taken to AArch32 Hyp mode.
- **Mon**: The exception is taken to AArch32 Monitor mode.
- **n/a**: Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

The following table describes the routing of physical interrupts if the highest implemented Exception level is using AArch64.
## Chapter B2. Exceptions

### B2.6. Asynchronous exception types

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>EEL2</th>
<th>EA</th>
<th>IRQ</th>
<th>RW</th>
<th>HCR</th>
<th>AMO</th>
<th>IMO</th>
<th>E2H</th>
<th>RW</th>
<th>Target when taken from</th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
<th>EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>FIQ, IRQ, Abt</td>
<td>FIQ, IRQ, Abt</td>
<td>n/a</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>EL1</td>
<td>EL1</td>
<td>n/a</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FIQ, IRQ, Abt</td>
<td>FIQ, IRQ, Abt</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>EL1</td>
<td>EL1</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>n/a</td>
<td>EL3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>FIQ, IRQ, Abt</td>
<td>FIQ, IRQ, Abt</td>
<td>Hyp</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Hyp</td>
<td>Hyp</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>FIQ</td>
<td>FIQ</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>EL1</td>
<td>EL1</td>
<td>EL1</td>
<td>EL1</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>EL2</td>
<td>n/a</td>
<td>EL2</td>
<td>EL2</td>
<td>EL2</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
B2.6.3 Asynchronous exception masking

A masked interrupt is not taken and remains pending.

A pending interrupt that is masked is not taken but remains pending.

If in AArch64 state and the target Exception level of an interrupt is lower than the current Exception level, the interrupt is masked.

### B2.6.3.1 Physical interrupt masking

If the target Exception level of a physical IRQ or FIQ interrupt is the current Exception level, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>Interrupt type</th>
<th>Control bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>PSTATE.I</td>
</tr>
<tr>
<td>FIQ</td>
<td>PSTATE.F</td>
</tr>
</tbody>
</table>

If the target Exception level of a physical SError interrupt is the current Exception level, the PSTATE.A control determines whether the interrupt is masked. However, if all of the following are true, the PSTATE.A control is ignored, and the interrupt is taken:

- The target Exception level is the current Exception level.
- v8.4-DFE is implemented.
- SCR_EL3.NMEA is 1.

If the target Exception level of a physical interrupt is higher than the current Exception level, all of the following apply:

- If the target Exception level is EL3, the interrupt cannot be masked by the PSTATE.\{A, I, F\} bits.
- If the target Exception level is EL2 and any of the following are true, the interrupt cannot be masked by the PSTATE.\{A, I, F\} bits:
  - HCR_EL2.E2H is 0.
  - HCR_EL2.TGE is 0.
- If the target Exception level is EL2 and all of the following are true, the interrupt can be masked by the PSTATE.\{A, I, F\} bits:
  - HCR_EL2.E2H is 1.
  - HCR_EL2.TGE is 1.
- If the target Exception level is EL1, the interrupt can be masked by the PSTATE.\{A, I, F\} bits.

The ability to execute in EL0 with interrupts taken to EL1 masked is required by some user level driver code.

The terms used in the table in this section have the following meanings:
### B2.6. Asynchronous exception types

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>This is the Effective value of a field in SCR_EL3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEL2</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1.</td>
</tr>
<tr>
<td></td>
<td>If FEAT_SEL2 is not implemented, the Effective value of SCR_EL3.EEL2 is 0.</td>
</tr>
<tr>
<td>FIQ IRQ</td>
<td>The Effective value of the field that configures the asynchronous exception type in SCR_EL3.</td>
</tr>
<tr>
<td>EA</td>
<td>If EL3 is not implemented, then the Effective value of these fields is 0.</td>
</tr>
<tr>
<td>RW</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.</td>
</tr>
<tr>
<td>HCR</td>
<td>If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.</td>
</tr>
<tr>
<td>TGE</td>
<td>If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.</td>
</tr>
<tr>
<td>E2H</td>
<td>If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.</td>
</tr>
<tr>
<td>FMO</td>
<td>The Effective value of the mask override field for the asynchronous exception type in HCR or HCR_EL2.</td>
</tr>
<tr>
<td>IMO</td>
<td>The Effective value of the field is one of the following:</td>
</tr>
<tr>
<td>AMO</td>
<td>• If EL2 is implemented, the FMO/IMO/AMO fields are taken from HCR_EL2.</td>
</tr>
<tr>
<td></td>
<td>• If EL2 is not implemented, the Effective value of these fields is 0.</td>
</tr>
</tbody>
</table>

**A** When the interrupt is pending, it is taken regardless of the value of the PSTATE.{A, I, F} interrupt masks.

**B** When the interrupt is pending, it is subject to the corresponding PSTATE mask. If the mask is 1, then the interrupt is not taken. If the mask is 0, the interrupt is taken.

**A/B** SError interrupts behave as A. FIQ and IRQ interrupts behave as B.

**C** When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F} interrupt masks.

**n/a** Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

The following table describes the target and masking of physical interrupts, if the highest implemented Exception level is using AArch64.
### B2.6.3.2 Virtual interrupt masking

If the target Exception level of a virtual interrupt is the current Exception level, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>EEL2</th>
<th>EA</th>
<th>IRQ</th>
<th>FIQ</th>
<th>RW</th>
<th>NMEA</th>
<th>HCR</th>
<th>E2H</th>
<th>AMO</th>
<th>IMO</th>
<th>FMO</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EL0</td>
</tr>
<tr>
<td>0 0 0 0 x x x x x x x B B n/a C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 x 0 x x x x A A n/a B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 x 1 x x x x A A n/a A/B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x 0 x 0 B B C C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x 0 x 1 A A B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x 1 0 x A n/a B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x 1 1 x B n/a B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 x 0 0 x x A A A B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 x 1 0 x x A A A A/B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 x 0 1 x x A n/a A B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 0 0 x 0 n/a 0 B B B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 0 0 x 0 n/a 1 A A B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 0 0 x 1 n/a x A n/a B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 0 1 x 0 x 0 B B C C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 1 x 0 x 1 A A B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 1 x 1 0 x 0 A A A B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 0 1 x 1 0 x 1 A A A A/B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 1 x 0 1 x x A n/a A B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 1 x 0 1 x x A n/a A A/B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 1 x 1 1 x x A n/a A A/B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Note:** The table above outlines the effect of interrupt mask controls on virtual interrupts based on the current Exception level. The mask is applied to determine whether a virtual interrupt is allowed to proceed.
B2.6. Asynchronous exception types

<table>
<thead>
<tr>
<th>Interrupt type</th>
<th>Control bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>vSError</td>
<td>PSTATE.A</td>
</tr>
<tr>
<td>vIRQ</td>
<td>PSTATE.I</td>
</tr>
<tr>
<td>vFIQ</td>
<td>PSTATE.F</td>
</tr>
</tbody>
</table>

Virtual interrupts can only be taken from EL0 or EL1 and can only be taken to EL1. If EL2 is not enabled in the current Security state, all types of virtual interrupts are always masked. If executing at EL2 or EL3, all types of virtual interrupts are always masked.

The terms used in the table in this section have the following meanings:

- **SCR_EL3**: This is the Effective value of a field in SCR_EL3.
- **HCR**: If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.
- **E2H**: If EL2 is using AArch32, the Effective value of HCR.E2H is 0.
- **B**: When the interrupt is pending, it is subject to the corresponding PSTATE mask. If the mask is 1, then the interrupt is not taken. If the mask is 0, the interrupt is taken.
- **C**: When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F} interrupt masks.
- **n/a**: Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

The following table describes the masking of virtual interrupts when the highest implemented Exception level is using AArch64:

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>HCR</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEL2</td>
<td>NS</td>
<td>EL0</td>
</tr>
<tr>
<td></td>
<td>EA</td>
<td>E2H</td>
</tr>
<tr>
<td>0 0 x x x x</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1 0 x x 0 0 0</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1 0 x x 0 1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1 0 x x 1 x</td>
<td>C</td>
<td>n/a</td>
</tr>
<tr>
<td>x 1 x x 0 0</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>x 1 x x 0 1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>x 1 x x 1 x</td>
<td>C</td>
<td>n/a</td>
</tr>
</tbody>
</table>

B2.6.4 Prioritization of interrupts
The prioritization of physical interrupts and virtual interrupts is IMPLEMENTATION DEFINED.

For all of the following context synchronization events, if an interrupt is pending before the context synchronization event and is not masked after the context synchronization event, the interrupt is taken before the first instruction after the context synchronization event:

- Execution of an ISB instruction.
- If FEAT_ExS is not implemented, exception entry.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set, exception entry.
- If FEAT_ExS is not implemented, exception return.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EOS bit is set, exception exit.
- Exit from Debug state.

If the first instruction after the context synchronizing event generates a synchronous exception, the architecture does not define whether the PE takes the interrupt or the synchronous exception first.

A RAS error synchronization event defines additional requirements for taking an SError interrupt.

Other than the behaviors described in RBBZYL, an unmasked, pending interrupt must be taken in finite time.

If an unmasked interrupt was pending but is changed to not pending before it is taken, it is CONSTRAINED UNPREDICTABLE whether or not the interrupt is taken. If the interrupt is taken, it is taken before the first context synchronization event after the interrupt was changed to be not pending.

**B2.6.5 Taking an interrupt during a multi-access load or store**

If in AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.
B2.7 UNDEFINED instructions

An instruction which is UNDEFINED generates a synchronous exception for that instruction unless there is a higher priority exception generated for that instruction.

The Exception level that the synchronous Undefined Instruction exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.
**B2.8 Configurable instruction controls**

AArch64 state provides configurable instruction controls for enabling, disabling, and trapping instructions. Configurable instruction controls might be referred to as an instruction enable, an instruction disable, or a trap control.

“Configurable instruction controls” are control bits held in System registers that determine whether attempting to execute an instruction generates a synchronous exception at the point in the instruction stream of that instruction, and the instruction is not executed.

Configurable instruction controls might be referred to by any of the following names:
- Instruction enables.
- Instruction disables.
- Trap controls.

The definitions of each type overlap, and in some cases is historical. Describing a register control field as an instruction enable, an instruction disable, or a trap control, gives no indication of how an exception that is generated as a consequence of the value of that field is handled or reported. Each configurable instruction control defines how the exception that is generated as a consequence of the configurable instruction control is handled or reported.

An exception can only be generated as a result of a configurable instruction control if all of the following apply:
- The instruction generating the exception does not also generate a higher priority exception.
- The instruction generating the exception is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

It is UNPREDICTABLE/CONSTRAINED_UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

An implementation might provide more controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all of the following are true:
- The preferred exception return address of the exception is the instruction that generates the exception.
- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

When a configurable instruction control causes a conditional instruction to generate an exception in AArch32 state, it is IMPLEMENTATION DEFINED whether the exception applies to conditional AArch32 instructions that fail their condition code check.
B2.8.1 EL0 and EL1 configurable instruction controls

There are configurable instruction controls at EL0 and EL1.

The following EL0 and EL1 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMUSERENR_EL0</td>
<td>Activity Monitors User Enable Register</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>Architectural Feature Access Control Register</td>
</tr>
<tr>
<td>MDSCR_EL1</td>
<td>Monitor System Debug Control Register</td>
</tr>
<tr>
<td>PMUSERENR_EL0</td>
<td>Performance Monitors User Enable Register</td>
</tr>
<tr>
<td>SCTLR_EL1</td>
<td>System Control Register (EL1)</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>Translation Control Register (EL1)</td>
</tr>
</tbody>
</table>

An exception caused by configurable instruction controls in EL1 can be taken from either AArch64 state or AArch32 state.

B2.8.2 EL2 configurable instruction controls

There are configurable instruction controls at EL2.

The following EL2 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPTR_EL2</td>
<td>Architectural Feature Trap Register, EL2</td>
</tr>
<tr>
<td>HAFGRTR_EL2</td>
<td>Hypervisor Activity Monitors Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HCR_EL2</td>
<td>Hypervisor Configuration Register</td>
</tr>
<tr>
<td>HDFGRTR_EL2</td>
<td>Hypervisor Debug Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HDFGWTR_EL2</td>
<td>Hypervisor Debug Fine-Grained Write Trap Register</td>
</tr>
<tr>
<td>HFGITR_EL2</td>
<td>Hypervisor Fine-Grained Instruction Trap Register</td>
</tr>
<tr>
<td>HFGTR_EL2</td>
<td>Hypervisor Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HFGWTR_EL2</td>
<td>Hypervisor Fine-Grained Write Trap Register</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>Hypervisor System Trap Register</td>
</tr>
<tr>
<td>MDCR_EL2</td>
<td>Monitor Debug Configuration Register, EL2</td>
</tr>
<tr>
<td>SCTLR_EL2</td>
<td>System Control Register, EL2</td>
</tr>
<tr>
<td>TCR_EL2</td>
<td>Translation Control Register, EL2</td>
</tr>
</tbody>
</table>

An exception caused by configurable instruction controls in EL2 can be taken from either AArch64 state or AArch32 state.

If Secure EL2 is implemented and enabled, configurable instruction controls available at EL2 apply in Secure state. If Secure EL2 is not implemented or not enabled, the configurable instruction controls available at EL2 are ignored.
B2.8.3 EL3 configurable instruction controls

There are configurable instruction controls at EL3.

The following EL3 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR_EL3</td>
<td>System Control Register, EL3</td>
</tr>
<tr>
<td>SCR_EL3</td>
<td>Secure Configuration Register</td>
</tr>
<tr>
<td>CPTR_EL3</td>
<td>Architectural Feature Trap Register, EL3</td>
</tr>
<tr>
<td>MDCR_EL3</td>
<td>Monitor Debug Configuration Register, EL3</td>
</tr>
<tr>
<td>TCR_EL3</td>
<td>Translation Control Register, EL3</td>
</tr>
</tbody>
</table>

An exception caused by configurable instruction controls in EL2 can be taken from either AArch64 state or AArch32 state.
B2.9 Exception Producing Instructions

A system call is another name for the exception producing instructions, HVC, SMC, and SVC.

The exception producing instructions are commonly called system calls and refer to any of the following synchronous exception types:

• Supervisor Call exception.
• Hypervisor Call exception.
• Secure Monitor Call exception.

A Supervisor Call is generated by executing an SVC instruction.

A Supervisor Call exception is taken to the following Exception levels:

• If executing at EL0:
  – If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  – If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
• If executing at EL1, the exception is taken to EL1.
• If executing at EL2, the exception is taken to EL2.
• If executing at EL3, the exception is taken to EL3.

A Supervisor Call enables software executing at EL0 to make a call to an operating system or other software executing at EL1.

If EL2 is implemented, a Hypervisor Call is generated by executing an HVC instruction.

A Hypervisor Call exception is taken to the following Exception levels:

• If Secure EL2 is implemented and enabled in the current Security state, when taken from EL1, the exception is taken to EL2.
• When taken from EL2, the exception is taken to EL2.
• When taken from EL3, the exception is taken to EL3.

If any of the following is true, the HVC instruction is UNDEFINED:

• The PE is executing at EL0.
• If EL2 is not enabled in the current Security state, and the PE is executing at EL1.
• SCR_EL3.HCE is 0.

If EL3 is implemented, a Secure Monitor Call is generated by executing an SMC instruction. A Secure Monitor Call is a synchronous exception that is taken to EL3.

If executing at EL0, the SMC instruction is UNDEFINED.
B2.10 Program counter and stack pointer alignment

B2.10.1 PC alignment checking

A misaligned PC generates a synchronous exception.

If bits [1:0] of the PC are not 0b00, there is a “misaligned PC”.

The execution of an instruction with a misaligned PC generates a synchronous PC Alignment exception on that instruction.

A PC Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1
  - If HCR_EL2.TGE is 1, the exception it taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

When a PC Alignment Fault exception is taken to an Exception level, ELx, using AArch64, the ELR_ELx and the FAR_ELx both hold the entire PC in its misaligned form.

A misalignment of the PC is an indication of a serious error, for example software corruption of an address.

B2.10.2 SP alignment checking

A misaligned SP can generate a synchronous exception.

When the SP is used as the base address of a calculation, regardless of any offset applied by the instruction, if bits [3:0] of the SP are not 0b0000, there is a “misaligned SP”.

If SP alignment checking is enabled, then the execution of a load or store using the SP with a misaligned SP generates a synchronous SP Alignment exception on that load or store.

PRFM instructions that use the SP do not perform stack alignment checking.

The following bits enable SP alignment checking at each Exception level when that Exception level is using AArch64.

- SCTLR_EL1.SA0 controls EL0.
- SCTLR_EL1.SA controls EL1.
- SCTLR_EL2.SA controls EL2.
- SCTLR_EL3.SA controls EL3.

An SP Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1
  - If HCR_EL2.TGE is 1, the exception it taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.