

# Arm® Architecture Reference Manual

## Supplement

**Reliability, Availability, and Serviceability (RAS), for Armv8-A**

The Arm logo, consisting of the lowercase letters 'arm' in a bold, sans-serif font.

## Release information

Date	Version	Changes
2020/Jul/22	D.a	<ul style="list-style-type: none"><li>Initial v8.6 Beta Release, with rewrite of the RAS supplement.</li></ul>
2019/Jul/01	C.b	<ul style="list-style-type: none"><li>Updated v8.4 Release.</li></ul>
2018/Oct/01	C.a	<ul style="list-style-type: none"><li>Initial v8.4 EAC Release.</li></ul>
2017/Dec/01	B.a	<ul style="list-style-type: none"><li>Updated EAC Release.</li></ul>
2017/Sep/01	B	<ul style="list-style-type: none"><li>EAC Release.</li></ul>
2017/Mar/01	A	<ul style="list-style-type: none"><li>First issue.</li></ul>

# 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 © 2017-2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

## Contents

# Arm® RAS Supplement

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

## Preface

Document status . . . . .	vii
About this book . . . . .	viii
Using this book . . . . .	ix
Conventions . . . . .	x
Typographical conventions . . . . .	x
Numbers . . . . .	x
Pseudocode descriptions . . . . .	x
Assembler syntax descriptions . . . . .	x
Rules-based writing . . . . .	xi
Identifiers . . . . .	xi
Examples . . . . .	xi
Additional reading . . . . .	xiii
Feedback . . . . .	xiv
Feedback on this book . . . . .	xiv

## Chapter 1

### Introduction to RAS

1.1	Faults, errors, and failures . . . . .	16
1.2	General taxonomy of errors . . . . .	17
1.2.1	Error detection . . . . .	17
1.2.2	Error propagation . . . . .	17
1.2.3	Infected and poisoned . . . . .	18
1.2.4	Containable and uncontainable . . . . .	18
1.3	Techniques for improving reliability, availability, and serviceability . . . . .	19
1.3.1	Fault prevention and fault removal . . . . .	19
1.3.2	Error handling and recovery . . . . .	19
1.3.3	Fault handling . . . . .	20

## Chapter 2

### Armv8-A RAS Extension

2.1	PE error handling . . . . .	22
2.1.1	PE error detection . . . . .	22
2.1.2	PE error propagation . . . . .	23
2.1.3	Other errors . . . . .	25
2.2	Generating error exceptions . . . . .	26
2.3	Taking error exceptions . . . . .	27
2.3.1	PE error state recording in the exception syndrome . . . . .	29
2.3.2	PE error state classification . . . . .	30
2.3.3	Multiple SError interrupts . . . . .	34
2.3.4	Target Exception level for External abort and SError interrupt exceptions taken to AArch64 state . . . . .	35
2.3.5	Target mode for External abort and SError interrupt exceptions taken to AArch32 state . . . . .	35
2.4	Error synchronization event . . . . .	36
2.4.1	ESB and Virtual SError interrupt exceptions . . . . .	39
2.4.2	Extension for synchronization at exception entry and return . . . . .	39
2.5	Virtual SError interrupts . . . . .	42

2.6	Error records in the PE . . . . .	43
2.6.1	Error record System register view . . . . .	43

## Chapter 3

### RAS System Architecture

3.1	Nodes . . . . .	46
3.1.1	Multiple error records per node . . . . .	47
3.1.2	Detecting and consuming errors . . . . .	48
3.2	Standard error record . . . . .	52
3.2.1	Component error states . . . . .	52
3.2.2	Writing the error record . . . . .	56
3.2.3	Error syndrome . . . . .	60
3.2.4	Security and Virtualization . . . . .	61
3.2.5	Synchronization and error record accesses . . . . .	62
3.2.6	Bridges to other architectures . . . . .	62
3.2.7	Software faults . . . . .	63
3.2.8	Other sources of error and warnings . . . . .	64
3.3	Error recovery interrupt . . . . .	65
3.4	Fault handling interrupt . . . . .	66
3.5	In-band error response signaling (external aborts) . . . . .	67
3.6	Critical error interrupt . . . . .	68
3.7	Standard format Corrected error counter . . . . .	69
3.8	Error recovery, fault handling, and critical error signaling . . . . .	71
3.9	Error recovery reset . . . . .	73
3.10	Timestamp extension . . . . .	74
3.11	Common Fault Injection Model Extension . . . . .	75
3.11.1	Operation of the Common Fault Injection Model Extension . . . . .	75

## Chapter 4

### RAS Extension and RAS System Architecture Registers

4.1	Memory-mapped view . . . . .	78
4.1.1	Access requirements for memory-mapped views of RAS error records . . . . .	79
4.2	Reset values . . . . .	80
4.3	Error record registers, including memory mapped view . . . . .	81
4.3.1	Register index . . . . .	81
4.3.2	ERR<n>ADDR, Error Record Address Register . . . . .	84
4.3.3	ERR<n>CTLR, Error Record Control Register . . . . .	87
4.3.4	ERR<n>FR, Error Record Feature Register . . . . .	95
4.3.5	ERR<n>MISC0, Error Record Miscellaneous Register 0 . . . . .	103
4.3.6	ERR<n>MISC1, Error Record Miscellaneous Register 1 . . . . .	109
4.3.7	ERR<n>MISC2, Error Record Miscellaneous Register 2 . . . . .	111
4.3.8	ERR<n>MISC3, Error Record Miscellaneous Register 3 . . . . .	113
4.3.9	ERR<n>PFGCDN, Pseudo-fault Generation Countdown Register . . . . .	115
4.3.10	ERR<n>PFGCTL, Pseudo-fault Generation Control Register . . . . .	117
4.3.11	ERR<n>PFGF, Pseudo-fault Generation Feature Register . . . . .	122
4.3.12	ERR<n>STATUS, Error Record Primary Status Register . . . . .	127
4.3.13	ERRCIDR0, Component Identification Register 0 . . . . .	143
4.3.14	ERRCIDR1, Component Identification Register 1 . . . . .	144
4.3.15	ERRCIDR2, Component Identification Register 2 . . . . .	145
4.3.16	ERRCIDR3, Component Identification Register 3 . . . . .	146
4.3.17	ERRCRICR0, Critical Error Interrupt Configuration Register 0 . . . . .	147
4.3.18	ERRCRICR1, Critical Error Interrupt Configuration Register 1 . . . . .	149
4.3.19	ERRCRICR2, Critical Error Interrupt Configuration Register 2 . . . . .	151
4.3.20	ERRDEVAFF, Device Affinity Register . . . . .	154
4.3.21	ERRDEVARCH, Device Architecture Register . . . . .	158
4.3.22	ERRDEVID, Device Configuration Register . . . . .	160
4.3.23	ERRERICR0, Error Recovery Interrupt Configuration Register 0 . . . . .	161

*Contents*  
*Contents*

4.3.24	ERRERICR1, Error Recovery Interrupt Configuration Register 1 . . .	163
4.3.25	ERRERICR2, Error Recovery Interrupt Configuration Register 2 . . .	165
4.3.26	ERRFHICR0, Fault Handling Interrupt Configuration Register 0 . . .	168
4.3.27	ERRFHICR1, Fault Handling Interrupt Configuration Register 1 . . .	170
4.3.28	ERRFHICR2, Fault Handling Interrupt Configuration Register 2 . . .	172
4.3.29	ERRGSR, Error Group Status Register . . . . .	175
4.3.30	ERRIIDR, Implementation Identification Register . . . . .	176
4.3.31	ERRIMPDEF< <i>n</i> >, IMPLEMENTATION DEFINED Register <0-191> . . .	178
4.3.32	ERRIRQCR< <i>n</i> >, Generic Error Interrupt Configuration Register . . .	179
4.3.33	ERRIRQSR, Error Interrupt Status Register . . . . .	180
4.3.34	ERRPIDR0, Peripheral Identification Register 0 . . . . .	184
4.3.35	ERRPIDR1, Peripheral Identification Register 1 . . . . .	185
4.3.36	ERRPIDR2, Peripheral Identification Register 2 . . . . .	186
4.3.37	ERRPIDR3, Peripheral Identification Register 3 . . . . .	188
4.3.38	ERRPIDR4, Peripheral Identification Register 4 . . . . .	190

**Chapter 5**

**Appendix**

5.1	Release notes for issue D.a . . . . .	192
-----	---------------------------------------	-----

**Glossary**

# Preface

## Document status

Beta release.

The information contained in this manual is at Beta quality. Beta quality means that all major features of the specification are described in the manual, some details might be missing.

In case of any apparent discrepancy or missing information, please contact Arm Limited.

## About this book

This manual describes the *Armv8-A RAS Extension* and *RAS System Architecture*.

This release of the Arm® RAS Supplement represents a major rewrite of the manual. The most significant change is to use the rules-based writing approach. There are also some changes to the order and flow of the manual, compared to previous releases.

*Release notes for issue D.a* summarizes the main changes and clarifications in the manual.



## Using this book

This manual is intended to be read in conjunction with [\[1\]](#).

# 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.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example <http://developer.arm.com>

## 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.

## Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

## Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a monospace font.

## Rules-based writing

This specification consists of a set of individual rules. Each rule is clearly identified by the letter R.

Rules must not be read in isolation, and where more than one rule relating to a particular feature exists, individual rules are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader. An implementation which is compliant with the architecture must conform to all of the rules in this specification.

Some architecture rules are accompanied by rationale statements which explain why the architecture was specified as it was. Rationale statements are identified by the letter X.

Some sections contain additional information and guidance that do not constitute rules. This information and guidance is provided purely as an aid to understanding the architecture. Information statements are clearly identified by the letter I.

Implementation notes are identified by the letter U.

Software usage descriptions are identified by the letter S.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Rules, rationale statements, information statements, implementation notes and software usage statements are collectively referred to as *content items*.

## Identifiers

Each content item may have an associated identifier which is unique within the context of this specification.

When the document is prior to beta status:

- Content items are assigned numerical identifiers, in ascending order through the document (*0001*, *0002*, ...).
- Identifiers are volatile: the identifier for a given content item may change between versions of the document.

After the document reaches beta status:

- Content items are assigned random alphabetical identifiers (*HJQS*, *PZWL*, ...).
- Identifiers are preserved: a given content item has the same identifier across versions of the document.

## Examples

Below are examples showing the appearance of each type of content item.

R	This is a rule statement.
R <sub>x001</sub>	This is a rule statement.
I	This is an information statement.
X	This is a rationale statement.
U	This is an implementation note.

S            This is a software usage description.

## Additional reading

This section lists publications by Arm and by third parties.

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

[1] *Arm® Architecture Reference Manual for ARMv8-A architecture profile*. (ARM DDI 0487) Arm Limited.

[2] *Basic Concepts and Taxonomy of Dependable and Secure Computing*. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr.

# 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](mailto:errata@arm.com). Give:

- The title (Arm® RAS Supplement).
- The number (ARM DDI 0587 D.a-00bet0).
- 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.

---

## Chapter 1

# Introduction to RAS

I<sub>LMHPD</sub>

*Reliability, Availability, Serviceability (RAS)* are three aspects of the *dependability* of a system:

- *Reliability*, the continuity of **correct service**.
- *Availability*, the readiness for correct service.
- *Serviceability*, the ability to undergo modifications and repairs.

I<sub>HWHJM</sub>

RAS techniques reduce unplanned outages because:

- Transient errors can be detected and corrected before they cause application or system failure.
- Failing components can be identified and replaced.
- Failure can be predicted ahead-of-time to allow replacement during planned maintenance.

## 1.1 Faults, errors, and failures

**R<sub>NVNNC</sub>** *Correct service* is delivered when the service implements the system function.

**I<sub>QWSVK</sub>** **Correct service** might include:

- Producing correct results.
- Producing results within the time allotted to the task.
- Not divulging secret or secure information.

For the purpose of describing the RAS Extension and RAS System Architecture, deviation from **correct service** is defined using the following terms:

**R<sub>NSPJY</sub>** • A *failure* is the event of deviation from correct service. This includes data corruption, data loss, and service loss.

**R<sub>SCKWX</sub>** • An *error* is the deviation from correct service. An incorrect value that has an **error** is *corrupt*.

**R<sub>YRDDR</sub>** • A *fault* is the cause of the error.

**R<sub>JNBDX</sub>** Errors that are present but not detected are *latent errors* or *undetected errors*.

**I<sub>TNQPK</sub>** In a system with no error detection, all errors are **latent errors** and are **silently propagated** by components until they are either masked or cause **failure**.

**I<sub>GRYKV</sub>** The severity of a **failure** can range from minor to catastrophic:

- The harmful consequences of a *minor failure* are of a similar cost to the benefits provided by correct service delivery.
- The harmful consequences of a *catastrophic failure* are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

**I<sub>NMGPO</sub>** There are many sources of faults in a system, including both software and hardware faults:

- *Hardware faults* originate in, or affect, hardware.
- *Software faults* affect software, that is programs or data.

The RAS Extension and RAS System Architecture primarily address errors produced from hardware faults. These fall into two main areas:

- Transient faults.
- Non-transient or *persistent faults*.



## 1.2 General taxonomy of errors

### 1.2.1 Error detection

- R<sub>FHXWP</sub>** When a component accesses memory or other state, an **error** might be detected in that memory or state.
- I<sub>WKPVR</sub>** The error might be **corrected** or **deferred** by the component, or **signaled** to another component as either a **deferred error** or a *detected error*.

### 1.2.2 Error propagation

- R<sub>LRZDN</sub>** A *transaction* occurs when a producer of the transaction passes a value or other signal to a consumer of the transaction.
- I<sub>VYCCX</sub>** Transactions are part of the service provided by the producer for the consumer.
- I<sub>RHGDL</sub>** In many protocols and service interface definitions, a high-level transaction consists of a sequence of operations, for instance between a *Requester* and a *Completer*.
- For the purposes of this manual, the most basic form of a unidirectional transfer between a *producer* and *consumer* is considered as a **transaction**.

That is, each one of the sequence of operations is considered a separate **transaction**. For some operations, such as a request, the Requester is producer and the Completer is the consumer. For other operations, such as a response, the Completer is producer and the Requester is the consumer.

- R<sub>SKZZG</sub>** An error is *propagated* by the producer of a **transaction** when the service interface is incorrect because of the error. The error is propagated to the consumer.

An error is **propagated** by deviations from **correct service**, including when any of the following occurs that would not have been permitted to occur had the fault not been activated:

- R<sub>XDHGD</sub>** • A **corrupt** value is passed from producer to consumer.
- R<sub>ZCNXB</sub>** • A transaction or other operation occurs that should not have occurred.
- R<sub>CFZKP</sub>** • A transaction or other operation that should have occurred does not occur.
- R<sub>MZVNK</sub>** • A loss of uniprocessor semantics or any other loss of coherency in a multiprocessor coherent system is observed.
- R<sub>KCDXV</sub>** • Changing the timing and/or order of transactions or other operations such that the timing and/or order of those transactions or operations is incorrect. In this case the service interface defines acceptable timings and/or orders for transactions and other operations.

The service interface for a **transaction** might include means to *signal* that the transaction is propagating:

- R<sub>VVFYS</sub>** • A **detected error**.
- R<sub>CSVRC</sub>** • A **deferred error**.

- R<sub>BHWVX</sub>** An error is *silently propagated* by the producer of a transaction if the consumer of the transaction cannot detect the error and consumes an **undetected error** because of the transaction. This might be because of one of the following:

- The error is present on the transaction, but was not detected by the producer. The error is **silently propagated** by the producer.

- The error is present on the transaction, but was not [signaled](#) to the consumer as an error. For example, a [corrupt](#) value was passed in the transaction with no indication that it was corrupt. The error is [silently propagated](#) by the producer.

**R<sub>FPBYS</sub>** A latent, possibly detectable, error is [silently propagated](#) by the consumer of an otherwise correct transaction if the transaction causes the error to become [undetectable](#).

**I<sub>XYVCP</sub>** For example, a partial write to a [protection granule](#) that removes [poison](#), leaving the unchanged portion of the location [corrupt](#). To implement a partial write, the consumer logically reads the current value of the location, modifies the value, and then writes the modified value back. These are internal transactions in the consumer that silently propagate the error. In this example there was no error at the producer nor on the transaction.

Errors might be [propagated](#) by components in a system until one of the following occurs:

**I<sub>YZTDY</sub>**

- They are masked and do not affect the outcome of the system.

The error might be masked because a corrupt value is discarded or overwritten, or the error is detected and [removed](#).

**I<sub>VQZPT</sub>**

- They affect the service interface of the system and possibly cause failure. If the error has been [silently propagated](#) to the service interface then:
  - This is a Silent Data Corruption (*SDC*).
  - The rate of such failures, measured as the number of failures per billion device-hours of operation, is called the SDC Failure-in-Time (*FIT*) rate.

Alternatively, the error might have been detected, causing the system to invoke error handling and recovery. See [Error handling and recovery](#).

### 1.2.3 Infected and poisoned

**R<sub>KNHWP</sub>** The state of a component becomes *infected* when the component consumes an [uncorrected error](#) that updates the state.

**R<sub>TZBSW</sub>** A value is *poisoned* in the state of a component if it is marked as being in error, such that a subsequent access of the state will detect the value is so marked and is treated as a [detected error](#).

**I<sub>YBMFK</sub>** [Poison](#) is used to [defer](#) an error.

### 1.2.4 Containable and uncontainable

**R<sub>DXQRB</sub>** An undetected error is *uncontained* at the component that failed to detect it.

**R<sub>RJYRQ</sub>** A [silently propagated](#) error is [uncontained](#) at the component that [silently propagated](#) it.

**R<sub>GJQNR</sub>** A [detected uncorrected error](#) is *uncontainable at the component* if it might be [uncontained](#) at the component. A [detected uncorrected error](#) is *containable at the component* if it is not [uncontainable at the component](#). If the component cannot determine whether a [detected uncorrected error](#) is [uncontainable](#) or [containable at the component](#), it treats it as [uncontainable at the component](#).

**I<sub>MRDMR</sub>** An error that is [uncontainable at a component](#) might be containable at the system level.

---

#### Note

Reporting an error as containable allows software to contain the error. It does not mean that hardware has contained the error.

---

## 1.3 Techniques for improving reliability, availability, and serviceability

**I<sub>TPGKF</sub>** Each device sets its own targets for reliability, availability, and serviceability, and uses different techniques to achieve these targets, including:

- *Fault prevention and fault removal.*
- *Error handling and recovery.*
- *Fault handling.*

**I<sub>DMKGY</sub>** The level of reliability, availability, and serviceability in any implementation, and which parts of the system include RAS, are IMPLEMENTATION DEFINED. The RAS Extension and RAS System Architecture do not prescribe the level of reliability, availability, and serviceability in any implementation, or which parts of the system include RAS.

### 1.3.1 Fault prevention and fault removal

**R<sub>YLVT</sub>** *Fault prevention and fault removal* are two techniques for handling faults. Fault prevention and fault removal mechanisms are IMPLEMENTATION DEFINED.

**I<sub>WZTKF</sub>** *Fault prevention* techniques are outside the scope of the architecture.

**R<sub>JVLNC</sub>** A fault that is removed is a *corrected error* and might be recorded and generate a *fault handling interrupt*, but it is not *propagated*. This means that it is not consumed and does not cause service failure.

**I<sub>WSPBC</sub>** A common technique to detect and correct errors is the use of an *Error Detection and Correction Code* (EDAC), more commonly referred to as simply an *Error Correction Code* (ECC). ECC schemes use mathematical codes to detect and correct an error in a value in memory. The size of the value is the *protection granule* for the ECC scheme.

**I<sub>PBJLC</sub>** The RAS Extension and RAS System Architecture do not require implementation any fault removal schemes, including ECC.

### 1.3.2 Error handling and recovery

**R<sub>XPLVT</sub>** A fault that is not removed gives rise to an *uncorrected error*.

**R<sub>VTXYY</sub>** *Error recovery* is the process by which software and hardware minimize the impact of an *uncorrected error*.

*Error recovery* methods include:

- I<sub>DCGYX</sub>**
- *Deferring* an error from a fault. An error is deferred by hardware if hardware can make forward progress without consuming the error. Deferring the error means:
    - The fault might become masked later (fault removal). For example, because the *corrupt* value is overwritten before it is consumed.
    - If the deferred error is later consumed, then the error is reported at the point of consumption. For example, if the deferred error is consumed by a *Processing element* (PE) then the consumer PE generates an *error exception*. This can give better results in terms of error recovery in the case where the original producer of the data is not known when the error was deferred. For example because a *latent error* was detected.

A common technique to defer an error is to replace the *corrupt* value with a *poisoned* value, for example in memory or in a *transaction*.

- I<sub>YLMTV</sub>**
- Preventing further *propagation* of the error, that is *containing* the error. In particular, preventing *silent propagation* of the error.

I <sub>RFLCM</sub>	<ul style="list-style-type: none"> <li>Reducing the severity of a failure by invoking a <i>service failure mode</i>: <ul style="list-style-type: none"> <li>This is a Detected Uncorrected Error (<i>DUE</i>).</li> <li>The rate of such failures gives the DUE <i>FIT</i> rate.</li> <li>The type of service failure mode depends on what is acceptable to the service.</li> </ul> </li> </ul>
I <sub>BRDMK</sub>	A software error recovery agent is typically invoked when hardware detects an error it cannot correct, defer, or remove.
I <sub>PGXFK</sub>	An error recovery agent also provides information to the operator through error logs to improve serviceability, for example to help with the identification of a Field Replaceable Unit ( <i>FRU</i> ).
I <sub>MFPFY</sub>	The RAS Extension and RAS System Architecture provide optional common programmers' models to record information about an error in an <i>error record</i> .
I <sub>CVFFN</sub>	The RAS Extension describes the behavior of a PE when an error is signaled to it by the system, including invoking a <i>service failure mode</i> by taking an <i>error exception</i> , and optional mechanisms to limit <i>propagation</i> of an error.
I <sub>TLDY</sub>	The RAS Extension and RAS System Architecture do not require systems to implement error recovery mechanisms, including <i>poison</i> , and do not require systems to limit the <i>silent propagation</i> of errors.

### 1.3.3 Fault handling

I <sub>SWFLQ</sub>	<p><i>Fault handling</i> by software is the process by which software diagnoses and responds to faults to improve availability.</p> <p><i>Fault handling</i> methods include:</p>
I <sub>GGCDN</sub>	<ul style="list-style-type: none"> <li><i>Predictive Failure Analysis</i> (PFA), using information recorded by hardware to trigger pre-emptive action.</li> </ul>
I <sub>WNHJF</sub>	The RAS Extension and RAS System Architecture provide optional mechanisms to allow the reporting of errors and warnings to a fault handling agent, and to record information about the fault in an <i>error record</i> . It is the responsibility of the <i>error recovery</i> and fault handling processes to collate the <i>error record</i> data and write it to an <i>error log</i> .
I <sub>FQRSQ</sub>	<p>The detailed nature of the fault handling agent is outside the scope of this architecture. Fault handling and <i>error recovery</i> might be independent agents.</p> <p>See also:</p> <ul style="list-style-type: none"> <li><i>Standard error record</i></li> </ul>

## Chapter 2

# Armv8-A RAS Extension

I <sub>LNLHW</sub>	The <i>Reliability, Availability, Serviceability</i> (RAS) Extension is identified as <i>FEAT_RAS</i> .
I <sub>FNVKV</sub>	The RAS Extension is a mandatory extension to the Armv8.2 architecture, and it is an optional extension to the Armv8.0 and Armv8.1 architectures.
I <sub>BQGSC</sub>	ID_AA64PFR0_EL1.RAS in AArch64 state, and ID_PFR0.RAS in AArch32 state, indicate whether the RAS Extension is implemented.
I <sub>LBKPL</sub>	The RAS Extension extends the exception syndrome registers to include fields that allow the <i>Processing element</i> (PE) to report a <a href="#">PE error state</a> when an <a href="#">error exception</a> is taken.
I <sub>DWKZS</sub>	The RAS Extension adds the <a href="#">Error synchronization event</a> and Error Synchronization Barrier instruction, <i>ESB</i> .
I <sub>THGHB</sub>	The RAS Extension defines System registers that are specific to RAS, including to access optional <a href="#">error records</a> defined by the <a href="#">RAS System Architecture</a> . The System register instructions are described in [1]. The format of the error record registers is defined in <a href="#">Error record System register view</a> .
I <sub>KPYCD</sub>	The <i>FEAT_IESB</i> feature provides controls to insert an implicit <a href="#">Error synchronization event</a> at exception entry and exception return.
I <sub>KJCLV</sub>	The <i>FEAT_RASv1p1</i> feature extends the RAS System registers to include support for <a href="#">RAS System Architecture</a> v1.1.
I <sub>GWRVK</sub>	The <i>FEAT_DoubleFault</i> feature provides EL3 controls to change the routing of synchronous External abort exceptions and treat SError interrupts as nonmaskable. <a href="#">FEAT_DoubleFault</a> is defined in [1].
R <sub>YWXWL</sub>	The RAS Extension does not prescribe the level of reliability, availability, and serviceability in the PE. The RAS features that the PE includes, for example to detect, correct, contain, or defer errors, are IMPLEMENTATION DEFINED. The RAS Extension defines a framework for building RAS features in a PE.

## 2.1 PE error handling

### 2.1.1 PE error detection

**I<sub>KRYQW</sub>** When a PE accesses memory or other state, an error might be detected in that memory or state, and **corrected**, **deferred**, or **signaled** to the PE as a **detected error** as an *in-band error response*.

---

#### Note

An error might also be signaled to a PE by means other than an **in-band error response**. See **R<sub>FNVVJ</sub>**.

---

The response from memory or other state is defined by *Detecting and consuming errors* in the *RAS System Architecture*:

**I<sub>DWWQJ</sub>**

- When an error is detected by a component on a read or a cache maintenance operation from the PE:
  - If the error can be **corrected**, it is corrected and corrected data is returned.
  - If the error cannot be **corrected** and can be **deferred**, it is deferred.
  - Otherwise, if **enabled** at the component, the **detected error** is signaled to the PE as an **in-band error response**.

The component might record the error and generate a **fault handling interrupt** and/or **error recovery interrupt**.

**I<sub>BKVQP</sub>**

- When an error is detected by a component consuming a write from the PE:
  - If the error can be **corrected**, it is corrected.
  - If the error cannot be **corrected** and can be **deferred**, it is deferred to the consumer. For example, by **poisoning** the location being written.
  - If **enabled** at the consumer, the **detected error** is signaled to the PE as an **in-band error response**.
  - If **enabled** at the consumer, the consumer generates an **error recovery interrupt**.

**I<sub>VRVFF</sub>** If the component implements the RAS System Architecture, its behavior is defined by *RAS System Architecture*, and depends on the nature of the error and IMPLEMENTATION DEFINED properties of the component. In each of these cases, the component might be a part of the processor, such as a cache, or might be outside of the processor.

The component might also report the error to a RAS System Architecture **node**, which records the error and might generate one or more of a **fault handling interrupt**, **error recovery interrupt**, or **critical error interrupt** depending on the features and configuration of the **node**.

See also *Other errors*.

---

#### Note

An **in-band error response** is sometimes referred to as an *External abort*. To avoid confusion with the External abort *exception*, this manual uses **in-band error response** to describe the response to the PE for a memory access.

See *In-band error response signaling (external aborts)*.

---

**R<sub>QTRKF</sub>** The features that the system and PE include to detect, correct, or defer errors are IMPLEMENTATION DEFINED.

**R<sub>WLTPV</sub>** The size of the **protection granule** for any implemented error detection mechanism in memory is IMPLEMENTATION DEFINED.

- I<sub>JJGW</sub>** A system might implement multiple error detection mechanisms with differing **protection granule** sizes.
- R<sub>FNGVW</sub>** The mechanism for clearing an error or poison from a memory **protection granule** is IMPLEMENTATION DEFINED, and it is IMPLEMENTATION DEFINED whether any such mechanism exists.

---

#### Note

For some systems, a single-copy atomic write of at least the whole **protection granule** can reset the state of the granule and clear any error or poison. In other systems, a **DC ZVA** operation might also clear the error. However, the **protection granule** might be larger than the **DC ZVA** block size and/or the largest single-copy atomic access that the PE can perform.

Systems might require software to stop using the **protection granule**, for example by not using the physical page containing the granule, until the system can be purged of errors, for example at a system reset. The architecture does not set any limit on the size of a **protection granule** and it might be larger than a translation granule.

Any mechanism for purging the system of errors is also IMPLEMENTATION DEFINED.

---

### 2.1.2 PE error propagation

- I<sub>NTXKV</sub>** The program-visible architectural state of the PE, referred to as the *PE state*, includes:
- General-purpose, SIMD&FP, and SVE registers.
  - System registers.
  - Special-purpose registers.
  - PSTATE.
- R<sub>XMBNW</sub>** An error is *consumed by the PE* by any of the following:
- An instruction commits the corruption into the **PE state**.
  - The error is on an instruction fetch and the corrupt instruction is committed for execution.
  - The error is on a translation table walk for a committed load, store, or instruction fetch.
- I<sub>HVFKW</sub>** For a PE, *Error propagation* applies to the propagation of **detected errors** by the PE between the **PE state**, and any other **PE state** or memory.

---

#### Note

*Memory* includes structures that cache the contents of memory, such as an instruction cache, data cache, or TLB.

---

An error is *propagated by the PE* by one or more of the following occurring that would not have been permitted to occur had the fault not been activated:

- R<sub>DQTHR</sub>**
- Consumption of the corrupt value by any instruction, propagating the error to the target(s) of the instruction. This includes:
    - A store of a corrupt value.
    - A write of a corrupt value to a System register, Special-purpose register, or PSTATE. Infecting a System register state might mean that the PE generates transactions that would not otherwise be permitted.
- R<sub>JKPCK</sub>**
- Any operation occurring that should not have occurred, including:

- A load, translation table walk, or instruction fetch that would not have been permitted, including those from hardware speculation or prefetching.
- A store to an incorrect address or a store that would not have been made or not permitted.
- A direct or indirect write to a Special-purpose or System register that would not have been made or not permitted.
- Assertion of any signal, such as an interrupt, that would not have been asserted.

$R_{LLKVP}$  • Any operation not occurring that should have occurred.

$R_{PZNG}$  • Causing the PE to take an imprecise exception, other than an [error exception](#) in response to the error itself. See the section *Definition of a precise exception* in [1].

$R_{PMDF}$  • The PE discarding data that it holds in a modified state.

$R_{DDFY}$  • Any other loss of required uniprocessor semantics, ordering, or coherency.

$R_{NQWB}$  The error [propagated by the PE](#) is *silently propagated by the PE* only if all of the following are true:

- The propagation is not part of the required operation of the PE in taking an [error exception](#) generated by the error.
- The propagation is not part of the required operation of the PE executing an [ESB](#) instruction that [synchronizes](#) the error.
- The error is not [signaled](#) to the consumer as a [detected error](#) or [deferred error](#).
- Any of the following are true:
  - The corrupt value is held in other than the general-purpose, SIMD&FP, or SVE registers.
  - The error is propagated by an instruction in program order before either taking an [error exception](#) generated by the error or executing an [ESB](#) instruction that [synchronizes](#) the error, and is propagated to outside of the general-purpose, SIMD&FP, or SVE registers.
  - The error is propagated other than by an instruction that consumes the corrupt value as an input operand but otherwise behaves correctly.

---

### Note

This means that after taking the [error exception](#) generated by the error, or an [ESB](#), propagating an error by, for example, storing it to memory, is not considered as silent propagation of the error by the PE.

For example, the PE takes an [error exception](#) in response to a load that returns a corrupt value to a general-purpose register. The error is not silently propagated to outside of the general-purpose registers before the [error exception](#) is taken. However:

- Taking the error exception causes the [ESR\\_ELx](#), [ELR\\_ELx](#), and [SPSR\\_ELx](#) registers to be updated. This is part of the required operation of the PE.
- After taking the error exception, software stores the contents of the general-purpose register to memory, and this is not [signaled](#) to memory as a [deferred error](#). This happens in program order after the exception is taken.

Neither of these actions are considered silent propagation of the error by the PE.

---

$R_{DTRFQ}$  The features that a PE includes to prevent silent propagation of an error are IMPLEMENTATION DEFINED.

$I_{NDTTS}$  For example, an implementation might ensure that a corrupt value in a general-purpose or SIMD&FP register is not silently propagated, by [signaling](#) a [deferred error](#) on any write of data to any memory location so that the memory location is [poisoned](#).



### 2.1.3 Other errors

I<sub>KRQMR</sub>

The RAS Extension deals mostly with errors detected by components outside of the PE, such as memory, and [consumed by the PE](#).

Other errors might be detected from within the processor itself. If these are not errors in the [PE state](#) they might be treated like errors detected by another component.

For example:

- A processor cache detects an error in the cache state that cannot be [corrected](#). The cache can be treated as a component outside the PE. If the error is detected in dirty cache data being evicted from the cache when the PE makes an access, it might be [deferred](#) by the cache writing [poison](#) in the evicted cache data. If the PE is performing a partial write that does not completely overwrite the protection granule, it might be [deferred](#) by the cache writing [poison](#) to the cache location, and/or evicting the cache line with [poison](#). Deferring the error means the error is not [consumed by the PE](#). Otherwise, the cache component generates the [in-band error response](#) to the PE.
- A processor detects a [corrupt](#) or [poisoned](#) value being returned from memory that is not being signaled as an [in-band error response](#) and cannot be [corrected](#) or [deferred](#). For example in response to a non-cacheable read or a cache refill. The interface to memory can be treated as a component outside the PE. The memory interface component generates the [in-band error response](#) to the PE.

In each of these cases, the *component* reports these errors to a [RAS System Architecture node](#) that implements [error records](#) and records the errors, and might generate one or more of a [fault handling interrupt](#), [error recovery interrupt](#), or [critical error interrupt](#) depending on the features and configuration of the [node](#).

I<sub>VNTWD</sub>

An example implementation might include error detection logic within the [PE state](#) itself. When the PE detects an error in the [PE state](#), the instruction that uses that state [consumes the error](#), and the PE generates an IMPLEMENTATION DEFINED SError interrupt exception. See [R<sub>FNVVJ</sub>](#).

In this case, the processor that implements the PE includes a [RAS System Architecture node](#) that implements [error records](#) that record these errors.

I<sub>JRQDM</sub>

An example implementation might support poisoning within the [PE state](#). When the PE consumes a [deferred error](#), for example a [poisoned](#) value, from memory into the [PE state](#), the [PE state](#) becomes poisoned. Subsequent operations that read the poisoned value can continue to defer the error by poisoning the result of the operation.

However, if the PE attempts to execute an operation that reads the poisoned value and cannot defer the error further, the PE generates an IMPLEMENTATION DEFINED SError interrupt exception. See [R<sub>FNVVJ</sub>](#).

In this case, the processor that implements the PE includes a [RAS System Architecture node](#) that implements [error records](#) that record these errors.

I<sub>JHQVK</sub>

Components outside of the PE might detect errors that are not [consumed by the PE](#). These components might report such errors to a PE using [error recovery interrupts](#).

R<sub>XJNNT</sub>

For implementations that include the Statistical Profiling Extension, the Statistical Profiling Extension behaves like a separate component.

I<sub>MJQQZ</sub>

Errors from [software faults](#) are outside the scope of the RAS Extension.

See also:

- [RAS System Architecture](#)
- [Software faults](#)

## 2.2 Generating error exceptions

R <sub>VJRKE</sub>	An <i>error exception</i> is generated when a detected error is signaled to the PE as an <a href="#">in-band error response</a> to an architecturally-executed memory access or cache maintenance operation. This includes any explicit data access, instruction fetch, translation table walk, or hardware update to the translation tables made by an architecturally-executed instruction.
I <sub>PJHJS</sub>	<a href="#">Error exception</a> is a term used in this manual to describe a collection of exception types. See <a href="#">Taking error exceptions</a> for more information.
R <sub>MBNBH</sub>	It is IMPLEMENTATION DEFINED whether an <a href="#">error exception</a> can be generated for an error that is consumed by hardware speculation or prefetching by a PE, but that is not committed to the architecturally visible state of the PE.
R <sub>SHKJB</sub>	It is IMPLEMENTATION DEFINED whether an <a href="#">error exception</a> can be generated for a detected error that is deferred.
R <sub>GVWJD</sub>	It is IMPLEMENTATION DEFINED whether an <a href="#">error exception</a> can be generated for a detected error that is corrected.
R <sub>FNVVJ</sub>	<p>An SError interrupt exception can also be generated for IMPLEMENTATION DEFINED causes.</p> <p>For example, when an error is detected and neither corrected nor deferred, and signaled to the PE by means other than an <a href="#">in-band error response</a>, or when an error detected by the PE in the <a href="#">PE state</a> or in the result of a calculation.</p>

## 2.3 Taking error exceptions

**R<sub>VXFYS</sub>** If **FEAT\_DoubleFault** is implemented, an **error exception** is taken as a synchronous External abort exception for all non-speculative:

- Instruction fetches.
- Translation table walks and hardware updates of translation tables on instruction fetches.

It is IMPLEMENTATION DEFINED whether an **error exception** is taken as a synchronous External abort exception or as an asynchronous SError interrupt exception for each non-speculative:

- If **FEAT\_DoubleFault** is not implemented, instruction fetch.
- Explicit access to memory made by an instruction.
- Cache maintenance operation.
- Translation table walk or hardware update of translation tables, other than for on an instruction fetch when **FEAT\_DoubleFault** is implemented.
- If **FEAT\_MTE** is implemented, access to an Allocation Tag in memory made by an instruction.

All **error exceptions** other than those explicitly mentioned in this rule are taken as an asynchronous SError interrupt exception.

**R<sub>WFNJG</sub>** When an **error exception** is taken as an asynchronous SError interrupt exception, the exception is taken in finite time.

**R<sub>BCXKN</sub>** When any of the following exceptions are taken, the PE records the *PE error state* in the exception syndrome register:

- A synchronous External abort taken to AArch64 state.
- An SError interrupt exception taken to either AArch32 or AArch64 state.

See *PE error state classification*.

**R<sub>TYVYR</sub>** When a synchronous External abort is taken to AArch32 state, the PE does not record the **PE error state**.

**I<sub>GGCQQ</sub>** The exception type and target execution state determines the set of **PE error state** values the PE can record. See *PE error state recording in the exception syndrome*.

**I<sub>WSYXB</sub>** The recorded **PE error state** informs software whether it can **recover execution** and, if so, whether any action by the recovery software to **locate and repair** the error is necessary first.

**I<sub>NFDSM</sub>** Software is only able to successfully *recover execution* and make progress from a restart address for the exception by executing an Exception Return instruction to branch to the instruction at this restart address if all of the following are true:

- The error has not been **silently propagated by the PE**.
- At the point when the Exception Return instruction is executed, the PE state and memory system state are consistent with the PE having executed all of the instructions up to but not including the instruction at the restart address, and none afterwards. That is, at least one of the following *restart conditions* is true:
  - The error has been not architecturally **consumed by the PE** and infected the **PE state**.
  - Executing the instruction at the restart address will not consume the error and will correct any corrupt state by overwriting it with the correct value or values.

**R<sub>DCKHJ</sub>** On taking an **error exception**, the PE determines that software is able to **recover execution** at the point where the exception is taken, with no additional action from software, if and only if all of the following are true:

- The error has not been [silently propagated by the PE](#).
- The [restart conditions](#) are met because all of the following are true:
  - Either the error does not remain [latent](#) or executing the instruction at the restart address will not consume the error and will correct any corrupt PE state.
  - The restart address is the preferred return address for the exception.
- The PE has not elected to determine that software is not able to [recover execution](#), and has not elected to determine that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error.

R<sub>JBHWY</sub>

On taking an [error exception](#), the PE determines that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error, to get the PE state and memory system state into this consistent state before attempting recovery, if and only if all the following are true:

- The error has not been [silently propagated by the PE](#).
- The [restart conditions](#) can be met because the restart address is the preferred return address for the exception and at least one of the following is true:
  - The error remains [latent](#) and executing the instruction at the restart address will access the corrupt state. If the error is removed then executing the instruction at the restart address will correct any corrupt PE state and/or corrupt memory state. For example, the instruction at the restart address is a load that will consume the error and corrupts PE state.
  - The error does not remain [latent](#) and the PE has elected to determine that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error.
  - Executing the instruction at the restart address will not consume the error and the PE has elected to determine that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error.
- The PE has not elected to determine that software is not able to [recover execution](#).

R<sub>GJQWN</sub>

On taking an [error exception](#), the PE determines that software is not able to [recover execution](#) if and only if one or more of the following are true:

- The error has been [silently propagated by the PE](#).
- The [restart conditions](#) cannot be met even if software takes action to [locate and repair](#) the error. This is because at least one of the following is true:
  - The error remains [latent](#) and executing the instruction at the restart address will consume the error and corrupt PE state. Either the error cannot be removed or executing the instruction at the restart address will not correct any corrupt PE state.
  - The restart address is not the preferred return address for the exception.
- The PE has elected to determine that software is not able to [recover execution](#).

I<sub>XMCCR</sub>

That the PE determines that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error does not mean that software can [locate and repair](#). For example, the error in memory might be one which cannot be located or cannot be repaired.

---

#### Note

Error recovery software might instead make the PE state and memory system state consistent with an *alternative execution* of the program.

For example, if the error is located in a clean page of memory and the error exception is generated by a load from the location infected with the error, then software might be able to repair the error by:

- Reloading the page from a backing store. This makes the memory system state consistent with the uncorrupted view. Executing the instruction at the restart address will load the uncorrupted value into the PE state.
- Invalidating the clean page and marking it page as inaccessible. Executing the instruction at the restart address will result in a Translation fault being generated when the program tries to access the page. The target of the load will contain an UNKNOWN value, which is permitted by the architecture. The MMU fault handler can then reload the page from the backing store, as it would for a page that has not been previously accessed or has been paged out.

Either approach might result in the virtual address to physical address mapping for the page being changed by software, meaning the memory system state is not consistent with the previously executed instructions. However, the memory system state is consistent with a valid alternative view of the execution of the program that allows software to [recover execution](#).

I<sub>RHPPV</sub>

A PE might include additional IMPLEMENTATION DEFINED mechanisms to aid software [locate and repair](#) the error.

If software has to use IMPLEMENTATION DEFINED mechanisms to [locate and repair](#) the error, then the PE reports that it has determined that software is not able to [recover execution](#). The PE might use IMPLEMENTATION DEFINED additional syndrome registers to report that software is able to [recover execution](#) if software takes action to [locate and repair](#) the error using the IMPLEMENTATION DEFINED mechanisms.

### 2.3.1 PE error state recording in the exception syndrome

R<sub>WPXYM</sub>

When an asynchronous SError interrupt exception is taken to AArch64 state, the PE records the [PE error state](#) in the exception syndrome register as the applicable one of:

- An applicable one of the following, in ESR\_ELx.AET, setting ESR\_ELx.IDS to 0b0 and ESR\_ELx.DFSC to the applicable nonzero fault status code:
  - [Uncontainable \(UC\)](#).
  - [Unrecoverable state \(UEU\)](#).
  - [Recoverable state \(UER\)](#).
  - [Restartable state \(UEO\)](#).
  - [Corrected \(CE\)](#).
- [Uncategorized error](#), by setting ESR\_ELx.ISS to all zeros.
- [IMPLEMENTATION DEFINED syndrome](#), by setting ESR\_ELx.IDS to 0b1.

R<sub>FKHHF</sub>

When a synchronous External abort exception is taken to AArch64 state, the PE records the [PE error state](#) in ESR\_ELx.SET as the applicable one of:

- [Uncontainable \(UC\)](#).
- [Recoverable state \(UER\)](#).
- [Restartable state \(UEO\)](#).

Other values for the [PE error state](#) are not supported by synchronous External abort exceptions taken to AArch64 state.

R<sub>PWKBL</sub>

When an asynchronous SError interrupt exception is taken to AArch32 state, the PE records the [PE error state](#) in DFSR.AET or HSR.AET as appropriate, as the applicable one of:

- [Uncontainable \(UC\)](#).
- [Unrecoverable state \(UEU\)](#).
- [Recoverable state \(UER\)](#).
- [Restartable state \(UEO\)](#).

Other values for the [PE error state](#) are not supported by asynchronous SError interrupt exceptions taken to AArch32 state.

I<sub>QVVS</sub>M

Table 2.1 summarizes the supported **PE error state** syndrome values for each type of **error exception**.

**Table 2.1: Summary of **error exception** types and supported **PE error state** syndrome values**

<b>PE error state</b>	<b>External abort to AArch64 state</b>	<b>SError interrupt to AArch64 state</b>	<b>External abort to AArch32 state</b>	<b>SError interrupt to AArch32 state</b>
<i>Recorded in:</i>	ESR_ELx.SET	ESR_ELx.AET	No syndrome	DFSR.AET
Uncategorized error	No	Yes (ISS==0)	-	No
IMPLEMENTATION DEFINED syndrome	No	Yes (IDS==1)	-	No
Uncontainable (UC)	Yes	Yes	-	Yes
Unrecoverable state (UEU)	No	Yes	-	Yes
Recoverable state (UER)	Yes	Yes	-	Yes
Restartable state (UEO)	Yes	Yes	-	Yes
Deferred (DE)	No	No	-	No
Corrected (CE)	No	Yes	-	No

### 2.3.2 PE error state classification

I<sub>CCKWK</sub>

The PE determines which **PE error state** to record based on the following criteria:

- The **PE error state** syndrome values supported by the type of **error exception** being taken. See *PE error state recording in the exception syndrome*.
- The following implementation-specific properties and behaviors of the PE on taking the exception:
  - Whether the error has been **silently propagated by the PE**.
  - Whether the PE determines that software is able to **recover execution** at the point where the exception is taken.
  - If the PE determines that software can recover execution, whether software needs **locate and repair** the error before attempting to recover. If software does not locate and repair the error, then attempting to recover execution might cause the **error exception** to be generated again.
  - If the PE determines that software cannot recover execution, whether the error is **synchronized by Error synchronization events**.
- Whether the implementation elects to record the **PE error state** as another state. The PE only does this when the criteria for the other, recorded state are met. The conditions under which the PE elects to record the **PE error state** as another state are IMPLEMENTATION DEFINED.

The recorded **PE error state** is defined by the rules in this section.

R<sub>QKZLB</sub>

If and only if all of the following are true, then on taking an **error exception** the **PE error state** is recorded as *Uncontainable (UC)*:

- One or more of the following are true:
  - The error has been **silently propagated by the PE**.
  - The PE determines that software is not able to **recover execution** from the preferred return address of the exception and the error is not **synchronized by Error synchronization events**.
  - The PE determines that software is not able to **recover execution** from the preferred return address of the exception and the **error exception** is taken as a synchronous External abort to AArch64 state. (That is, the type of **error exception** does not support reporting the **PE error state** as **Unrecoverable state (UEU)**.)
  - The implementation has elected to record the **PE error state** as **Uncontainable (UC)**.
- The **error exception** is not taken as a synchronous External abort to AArch32 state.

- The implementation has not elected to record the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error, or the type of error exception does not support reporting the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error.

R<sub>QGN</sub>YD

If and only if all of the following are true, then on taking an error exception the PE error state is recorded as *Unrecoverable state (UEU)*:

- The error has not been silently propagated by the PE.
- The error exception is taken as an SError interrupt exception.
- One or more of the following are true:
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the error is synchronized by Error synchronization events.
  - The implementation has elected to record the PE error state as Unrecoverable state (UEU).
- The implementation has not elected to record the PE error state as Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

I<sub>FJC</sub>ZP

*Error synchronization event* defines synchronized by Error synchronization events.

R<sub>JHN</sub>VT

If and only if all of the following are true, then on taking an error exception the PE error state is recorded as *Recoverable state (UER)*:

- The error has not been silently propagated by the PE.
- The error exception is not taken as a synchronous External abort to AArch32 state.
- The PE determines that software is able to recover execution from the preferred return address of the exception.
- One or more of the following are true:
  - The PE determines that software must take action to locate and repair the error to successfully recover execution. This might be because the exception was taken before the error was architecturally consumed by the PE, at the point when the PE was not be able to make correct progress without either consuming the error or otherwise making the state of the PE unrecoverable.
  - The implementation has elected to record the PE error state as Recoverable state (UER).
- The implementation has not elected to record the PE error state as Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

R<sub>MBV</sub>CF

If and only if all of the following are true, then on taking an error exception the PE error state is recorded as *Restartable state (UEO)*:

- The error has not been silently propagated by the PE.
- The error exception is not taken as a synchronous External abort to AArch32 state.
- The PE determines that software can recover execution from the preferred return address of the exception without the need for software to take action to locate and repair the error first.
- One or more of the following are true:
  - The error is an uncorrected error. This includes a deferred error.
  - The error is a corrected error and the error exception is not taken as an SError interrupt taken to AArch64 state.
  - The implementation has elected to record the PE error state as Restartable state (UEO).
- The implementation has not elected to record the PE error state as any of Recoverable state (UER), Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.



$R_{LFXRD}$	<p>If and only if all of the following are true, then on taking an <a href="#">error exception</a> the <a href="#">PE error state</a> is recorded as <i>Corrected (CE)</i>:</p> <ul style="list-style-type: none"> <li>• The error has been corrected and not <a href="#">silently propagated by the PE</a>.</li> <li>• The <a href="#">error exception</a> is taken as an SError interrupt taken to AArch64 state.</li> <li>• Software can <a href="#">recover execution</a> from the preferred return address of the exception. Because the error has been corrected, software does not need to take action to <a href="#">locate and repair</a> the error.</li> <li>• The implementation has not elected to record the <a href="#">PE error state</a> as any other type.</li> </ul>
$R_{NZYRP}$	<p>If and only if all the following are true, then on taking an <a href="#">error exception</a> the <a href="#">PE error state</a> is recorded as an <i>Uncategorized error</i>:</p> <ul style="list-style-type: none"> <li>• The <a href="#">error exception</a> is taken as an asynchronous SError interrupt taken to AArch64 state.</li> <li>• The implementation has elected to record the <a href="#">PE error state</a> as an <a href="#">Uncategorized error</a>.</li> </ul>
$R_{VHWD}$	<p>If and only if all the following are true, then on taking an <a href="#">error exception</a> the <a href="#">PE error state</a> is recorded as an <i>IMPLEMENTATION DEFINED syndrome</i></p> <ul style="list-style-type: none"> <li>• The <a href="#">error exception</a> is taken as an asynchronous SError interrupt taken to AArch64 state.</li> <li>• The implementation has elected to record the <a href="#">PE error state</a> as an <a href="#">IMPLEMENTATION DEFINED syndrome</a>.</li> </ul>
$I_{SRPJD}$	<p>The <a href="#">IMPLEMENTATION DEFINED syndrome</a> type might provide additional <i>IMPLEMENTATION DEFINED syndrome</i> recorded in the exception syndrome register. Software might be able to determine the state of the PE from this syndrome, or other <i>IMPLEMENTATION DEFINED syndrome</i> registers.</p>
$I_{WLZRP}$	<p><a href="#">Uncategorized error</a> and <a href="#">IMPLEMENTATION DEFINED syndrome</a> are defined for backwards compatibility with previous versions of the architecture. Arm does not recommend use of these <a href="#">PE error state</a> values in new implementations that include other RAS features.</p>
$I_{VKMZB}$	<p>The <a href="#">PE error states</a> are summarized by <a href="#">Figure 2.1</a>. <a href="#">Figure 2.1</a> assumes the type of <a href="#">error exception</a> supports the resulting <a href="#">PE error state</a>, never elects to record an error as a different <a href="#">PE error state</a> when permitted, and does not show <a href="#">Uncategorized error</a> or <a href="#">IMPLEMENTATION DEFINED syndrome</a>.</p>



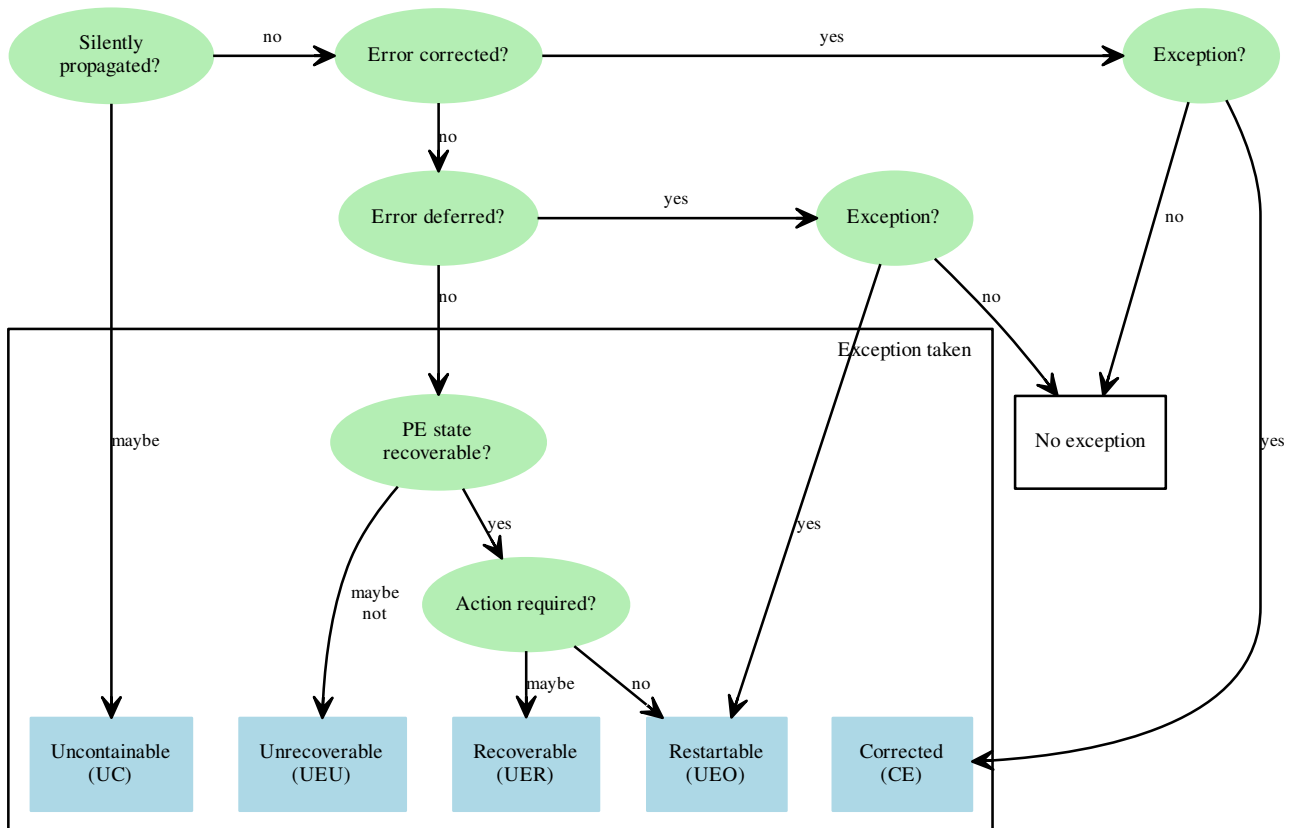


Figure 2.1: PE error states

I<sub>ZQ</sub>RGL

If the **PE error state** reports that software can **recover execution**, or that software isolation might be possible because the error is **synchronized by Error synchronization events**, this does not necessarily mean that the error can be recovered from because the error in the system might be one which does not allow software to recover the operation. Rather, software *might* be able to recover if it can repair the error and continue.

For example, the component that originally detected the error and signaled it to the PE might record in a **RAS System Architecture node** that the error is uncontainable at the component, meaning the system has to be shut down to avoid catastrophic failure. The **in-band error response** to the PE is not required to signal the severity of the error to the PE. The recorded **PE error state** refers only to the PE, not the system error state.

If the **in-band error response** can signal the severity of the error to the PE, the PE might use this information to elect to report the **PE error state** as other than **Recoverable state (UER)**. For example, if a processor cache detects an uncontainable tag RAM error, the PE might report the **PE error state** as **Uncontainable (UC)**, even though the state of the PE itself is recoverable. However, this is not required, and software must not rely on this behavior and should determine from the system whether the error is recoverable at the system level.

See also:

- **PE error propagation**
- **Error synchronization event**

## Using the PE error state classification

S<sub>XSKNS</sub>

When the **PE error state** is recorded as **Uncontainable (UC)**:

- Software must assume that either:

- The error has been [silently propagated by the PE](#).
  - Software is not able to [recover execution](#) from the preferred return address of the exception and the error was not [synchronized by Error synchronization events](#).
- If the error cannot be otherwise isolated to an application or VM, or both, the system must be shut down by software to avoid catastrophic failure.

S<sub>HYWFL</sub>When the [PE error state](#) is recorded as [Unrecoverable state \(UEU\)](#):

- Software can assume the error has not been [silently propagated by the PE](#).
- Software cannot safely [recover execution](#) from the preferred return address of the exception, even if it takes action to [locate and repair](#) the error. The state of the affected software, or both, is unrecoverable. However, if the software includes [Error synchronization events](#), software can use the properties of the Error synchronization event to determine which software is affected by the error.
- The affected software cannot continue and must be isolated by software.

S<sub>LSFYM</sub>When the [PE error state](#) is recorded as [Recoverable state \(UER\)](#):

- The uncorrected error might remain [latent](#) in the system.
- If the exception handler takes action to [locate and repair](#) the uncorrected error, it can safely [recover execution](#) from the preferred return address of the exception. Otherwise on restart of the affected software the PE might attempt to consume the error again, causing a further [error exception](#). If software cannot [locate and repair](#) the error, the affected software must be isolated by software.

S<sub>GLPZY</sub>When the [PE error state](#) is recorded as [Restartable state \(UEO\)](#):

- The error might remain [latent](#) in the system.
- Software might take action to [locate and repair](#) the error before it is consumed. However, the affected software can be safely restarted by the exception handler without software taking any action to [locate and repair](#) the error.

For example, the error was signaled when the PE speculatively accessed corrupt data.

S<sub>GRZQS</sub>

When the [PE error state](#) is recorded as [IMPLEMENTATION DEFINED syndrome](#) or [Uncategorized error](#), if software is not able to determine the actual state of the PE and memory, it should treat [IMPLEMENTATION DEFINED syndrome](#) and [Uncategorized error](#) as [Uncontainable \(UC\)](#).

### 2.3.3 Multiple SError interrupts

I<sub>CPJLW</sub>

Multiple SError interrupt conditions might be pending together. The architecture does not define relative priorities for asynchronous exceptions.

R<sub>DKHQZ</sub>

If multiple SError interrupt conditions are pending, it is [IMPLEMENTATION DEFINED](#) whether the multiple pending SError interrupt conditions are taken as a single SError interrupt exception.

I<sub>JBQSC</sub>

On taking an SError interrupt exception for more than one SError interrupt condition:

- If the exception is taken to AArch64 state and one or more pending SError interrupt conditions would be reported as [IMPLEMENTATION DEFINED syndrome](#), then the syndrome recorded in ESR\_ELx.ISS is [IMPLEMENTATION DEFINED](#).
- Otherwise, the recorded [PE error state](#) applies recorded by combined effect of the errors.

I<sub>GNHXJ</sub>

Any pending SError interrupt conditions that are not taken with other SError interrupts as a single SError interrupt exception remains pending after the SError interrupt exception is taken.

### 2.3.4 Target Exception level for External abort and SError interrupt exceptions taken to AArch64 state

I<sub>NRZXZ</sub> This section is included for completeness. It repeats the definitions from [1] and so is *non-normative*.

The default target Exception level for SError interrupt and synchronous External abort exceptions taken to AArch64 state is:

- EL1, if taken from EL0 or EL1.
- EL2, if taken from EL2.
- EL3, if taken from EL3.

However:

- If EL3 is implemented and SCR\_EL3.EA is 0b1, all SError interrupt and synchronous external abort exceptions are taken to EL3.
- Otherwise, if EL2 is implemented and enabled in the current Security state, then:
  - If HCR\_EL2.AMO is 0b1 or HCR\_EL2.TGE is 0b1, all SError interrupts from EL0 and EL1 are taken to EL2.
  - If HCR\_EL2.TEA is 0b1 or HCR\_EL2.TGE is 0b1, all synchronous External abort exceptions from EL0 and EL1 are taken to EL2.

### 2.3.5 Target mode for External abort and SError interrupt exceptions taken to AArch32 state

I<sub>BMBXM</sub> This section is included for completeness. It repeats the definitions from [1] and so is *non-normative*.

For SError interrupt and synchronous External abort exceptions taken to AArch32 state, the default target mode is:

- Abort mode, if taken from EL0, EL1 or EL3, including from Secure Monitor mode.
- Hyp mode, if taken from EL2.

However:

- If EL3 is implemented and using AArch32 and SCR.EA is 0b1:
  - All SError interrupt and synchronous external Data Abort exceptions are taken to Secure Monitor mode, using vector offset 0x10.
  - All synchronous external Prefetch Abort exceptions are taken to Secure Monitor mode, using vector offset 0x0C.
- Otherwise, if EL2 is implemented and using AArch32 and the PE is in Non-secure state:
  - If HCR.AMO is 0b1 or HCR.TGE is 0b1, all SError interrupts from EL0 and EL1 are taken to Hyp mode, using vector offset 0x14.
  - If HCR.TEA is 0b1 or HCR.TGE is 0b1, all synchronous External abort exceptions from EL0 and EL1 are taken to Hyp mode, using vector offset 0x14.

## 2.4 Error synchronization event

I <sub>YGRDP</sub>	The RAS Extension defines the <a href="#">Error synchronization event</a> and the <code>ESB</code> instruction.
R <sub>GRJVN</sub>	<p>An <i>Error synchronization event</i> is generated by all of the following:</p> <ul style="list-style-type: none"> <li>• Executing an <code>ESB</code> instruction.</li> <li>• When <a href="#">FEAT_IESB</a> is implemented, and one of the following is true, taking an exception to an Exception level, ELx, using AArch64: <ul style="list-style-type: none"> <li>– The appropriate <code>SCTLR_ELx.IESB</code> bit is 0b1.</li> <li>– <a href="#">FEAT_DoubleFault</a> is implemented, the Exception level is EL3, and <code>SCTLR_EL3.NMEA</code> is 0b1.</li> </ul> </li> </ul> <p>In Debug state this also applies to executing a <code>DCPSx</code> instruction to ELx.</p> <ul style="list-style-type: none"> <li>• When <a href="#">FEAT_IESB</a> is implemented, and one of the following is true, executing an exception return instruction at an Exception level, ELx, using AArch64: <ul style="list-style-type: none"> <li>– The appropriate <code>SCTLR_ELx.IESB</code> bit is 0b1.</li> <li>– <a href="#">FEAT_DoubleFault</a> is implemented, the Exception level is EL3, and <code>SCR_EL3.NMEA</code> is 0b1.</li> </ul> </li> </ul> <p>In Debug state this also applies to executing a <code>DRET</code> instruction at ELx.</p>
I <sub>PMGVM</sub>	In addition to generating an <a href="#">Error synchronization event</a> , the <code>ESB</code> instruction might additionally record and then clear a masked pending asynchronous SError interrupt exception.
I <sub>NWLTG</sub>	For details of the operation and encoding of <code>ESB</code> , see [1].
I <sub>HQZRM</sub>	The <a href="#">FEAT_IESB</a> feature and <code>SCTLR_ELx.IESB</code> bits are described by [1]. See also <a href="#">Extension for synchronization at exception entry and return</a> .
I <sub>NNXVF</sub>	The <a href="#">FEAT_DoubleFault</a> feature and <code>SCR_EL3.NMEA</code> bit are described by [1]. See also <a href="#">Extension for synchronization at exception entry and return</a> .
R <sub>YZPBD</sub>	<p>An error is <i>synchronized by Error synchronization events</i> if and only if all the following are true for each Error synchronization event:</p> <ul style="list-style-type: none"> <li>• The error is generated by an instruction on the same PE as the Error synchronization event. This includes any memory accesses, instruction fetch, translation table walk, or hardware update to the translation tables made by the instruction.</li> <li>• If the <a href="#">error exception</a> for the error is <a href="#">taken in program order after the Error synchronization event completes</a>, and either physical SError interrupt exceptions are unmasked when the Error synchronization event occurs or the <a href="#">error exception</a> is taken synchronously, then all of the following are true: <ul style="list-style-type: none"> <li>– The instruction that generated the error is in program order after the Error synchronization event.</li> <li>– On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.</li> </ul> </li> <li>• If the <a href="#">error exception</a> for the error is taken asynchronously as an SError interrupt, physical SError interrupt exceptions are masked when the Error synchronization event occurs, and the SError interrupt is not pending when the Error synchronization event completes, then all of the following are true: <ul style="list-style-type: none"> <li>– The instruction that generated the error is in program order after the Error synchronization event.</li> <li>– On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.</li> </ul> </li> </ul> <p>The SError interrupt is not pending when the Error synchronization event completes if a subsequent read of <code>ISR_EL1.A</code> or <code>ISR.A</code> returns 0b0.</p> <ul style="list-style-type: none"> <li>• If the <a href="#">error exception</a> for the error is taken asynchronously as an SError interrupt, the Error synchronization</li> </ul>

event is generated by an `ESB` instruction executed when physical SError interrupt exceptions are masked, and the `ESB` instruction does not set `DISR_EL1.A` or `DISR.A` to `0b1`, then all of the following are true:

- The instruction that generated the error is in program order after the `ESB`.
- On completion of the `ESB`, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the `ESB`.

`R_NFKMQ`

*Taken in program order after the Error synchronization event completes* means:

- For an Error synchronization event generated by an `ESB` instruction, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception return instruction when `FEAT_IESB` implemented, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception entry when `FEAT_IESB` is implemented, one of the following is true:
  - The exception is taken in program order strictly after the first instruction of the exception handler at the exception vector address.
  - The exception is taken from the first instruction of the exception handler at the exception vector address and the `ESR_ELx.IESB` syndrome bit is recorded as `0b0`.

`I_QZSHG`

The definition of *synchronized by Error synchronization events* means that if the error that is *synchronized by Error synchronization events* is generated by an instruction in program order before the Error synchronization event, then either the *error exception* is taken before the Error synchronization event, or on executing the Error synchronization event the following apply:

- If physical SError interrupt exceptions are unmasked or the error exception is taken synchronously, the Error synchronization event ensures that the error exception is not taken in program order after the Error synchronization event. This allows isolation of the software affected by the error.
- If physical SError interrupt exceptions are masked and the error exception is taken asynchronously:
  - If the Error synchronization event was generated by an `ESB`, the error is recorded in `DISR_EL1` or `DISR`. Software can use the *PE error state* recorded in `DISR_EL1` or `DISR` to determine what recovery is possible.
  - Otherwise, the error exception is pending when the Error synchronization event completes.

The SError interrupt might have been pending before or made pending by the Error synchronization event.

The definition does not mean that if the error is generated by a instruction in program order after the Error synchronization event, then the *error exception* will only be taken after the Error synchronization event. The error exception might be taken before the Error synchronization event, if the PE speculated past the Error synchronization event and speculatively executed the instruction that generated the error. This might cause software to generate a false failure. Error synchronization events are not speculation barriers.

`I_SQCFG`

It is implementation-specific which physical errors are *synchronized by Error synchronization events*. However, the criteria for the *PE error state* mean that if the PE reports the *PE error state* as one of the following, the error must be either explicitly or implicitly *synchronized by Error synchronization events*:

- *Unrecoverable state (UEU)*.
- *Recoverable state (UER)*.
- *Restartable state (UEO)*.

This is because *synchronized by Error synchronization events* is a criterion for *Unrecoverable state (UEU)*, and the criteria for *Recoverable state (UER)* and *Restartable state (UEO)* satisfy the definition of *synchronized by Error synchronization events*.

For other physical errors:

- An error that has been [silently propagated by the PE](#) must be reported as [Uncontainable \(UC\)](#) and is not containable even if [synchronized by Error synchronization events](#). Software must assume the error has been silently propagated even if the error is [synchronized by Error synchronization events](#).
- The following errors have not been consumed by the PE:
  - A *Deferred error*.
  - A *Corrected error*.
  - An error exception from a read by hardware speculation that does not corrupt the state of the PE.

Software can [recover execution](#) from these errors regardless of whether the error is [synchronized by Error synchronization events](#).

- An implementation might have other IMPLEMENTATION DEFINED sources of SError interrupt, see [R\\_FNVVJ](#). If an IMPLEMENTATION DEFINED SError interrupt is generated by a level-sensitive interrupt signal, it cannot be [synchronized by Error synchronization events](#).

[I\\_VFFYW](#) An [Error synchronization event](#) might operate as follows:

- (1) The PE ensures that any error [synchronized by Error synchronization events](#) and generated by an instruction in program order before the Error synchronization event has caused a physical SError interrupt exception to become pending.
- (2) If a physical SError interrupt is pending for an error [synchronized by Error synchronization events](#) and generated by an instruction in program order before the Error synchronization event, and physical SError interrupt exceptions are not masked at the current Exception level, then the physical SError interrupt exception is taken before completion of the Error synchronization event. The SError interrupt might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

[I\\_RDWTF](#) The prioritization of asynchronous interrupts is IMPLEMENTATION DEFINED. This means the PE might take another exception before an SError interrupt made pending by the [Error synchronization event](#). In this case, the SError interrupt remains pending.

Arm recommends the SError interrupt is prioritized over other exceptions.

[R\\_NPPGJ](#) If an SError interrupt for an error [synchronized by Error synchronization events](#) is pending after completing the [Error synchronization event](#) generated by an ESB instruction, and physical SError interrupt exceptions are masked at the current Exception level, the ESB instruction performs the following steps:

- (1) The pending physical SError interrupt is recorded in DISR\_EL1 or DISR. This includes the [PE error state](#) that the pending error exception would record if taken.
- (2) The DISR\_EL1.A bit or DISR.A bit is set to 0b1.
- (3) The pending state of the physical SError interrupt is cleared.

The SError interrupt might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

[R\\_BLRMT](#) The criteria for ESB recording the [PE error state](#) in DISR\_EL1 or DISR are the same as for that for recording the [PE error state](#) in ESR\_ELx or DFSR when an SError interrupt exception taken to the current execution state.

[R\\_KNWB](#) If an SError interrupt is taken as part of an [Error synchronization event](#) generated by an ESB instruction, the ESB instruction address is the *preferred return address* of the exception.

---

#### Note

See [1] for the definition of the *preferred return address* for an exception.

---

<code>R<sub>SFHDS</sub></code>	On executing an <code>ESB</code> instruction when SError interrupt exceptions are masked, any pending SError interrupt generated by an error that is not <a href="#">synchronized by Error synchronization events</a> : <ul style="list-style-type: none"> <li>Remains pending after completion of the Error synchronization event.</li> <li>Does not update <code>DISR_EL1</code> or <code>DISR</code>.</li> </ul>
<code>I<sub>CQKXL</sub></code>	The <a href="#">error recovery</a> , <a href="#">fault handling</a> , and <a href="#">critical error</a> interrupts described by <a href="#">RAS System Architecture</a> are asynchronous interrupts, not errors, and so are not <a href="#">synchronized by Error synchronization events</a> .
<code>I<sub>BBGXN</sub></code>	If multiple SError interrupt conditions are pending, an <a href="#">Error synchronization event</a> synchronizes all errors that are <a href="#">synchronized by Error synchronization events</a> .
<code>S<sub>VFHGT</sub></code>	Software must be aware that an SError interrupt taken at an <a href="#">Error synchronization event</a> or recorded in the <code>DISR_EL1</code> or <code>DISR</code> register by an <code>ESB</code> instruction might have been generated by hardware speculation of an instruction in program order after the Error synchronization event.

### 2.4.1 ESB and Virtual SError interrupt exceptions

<code>R<sub>LLLV</sub></code>	If all of the following are true, then an <code>ESB</code> instruction executed at EL0 or EL1 also synchronizes a pending virtual SError interrupt: <ul style="list-style-type: none"> <li>EL2 is implemented and enabled in the current Security state.</li> <li>Any of the following are true: <ul style="list-style-type: none"> <li>EL2 is using AArch64, <code>HCR_EL2.AMO</code> is set to <code>0b1</code>, and <code>HCR_EL2.TGE</code> is set to <code>0b0</code>.</li> <li>EL2 is using AArch32, <code>HCR.AMO</code> is set to <code>0b1</code>, and <code>HCR.TGE</code> is set to <code>0b0</code>.</li> </ul> </li> </ul> <p>In these cases, if a virtual SError interrupt is pending when the <code>ESB</code> instruction is executed:</p> <ul style="list-style-type: none"> <li>If the virtual SError interrupt is unmasked at the current Exception level, it is taken before the completion of the <code>ESB</code> instruction.</li> <li>If the virtual SError interrupt is masked at the current Exception level: <ul style="list-style-type: none"> <li><code>HCR_EL2.VSE</code> or <code>HCR.VA</code> is cleared to <code>0b0</code>.</li> <li>The virtual SError interrupt syndrome from <code>VSESR_EL2</code> or <code>VDFSR</code> is recorded in <code>VDISR_EL2</code> or <code>VDISR</code>.</li> <li><code>VDISR_EL2.A</code> or <code>VDISR.A</code> is set to <code>0b1</code> to indicate the SError interrupt was pending prior to the execution of the <code>ESB</code> instruction.</li> </ul> </li> </ul>
-------------------------------	---

---

#### Note

This happens in parallel with the [Error synchronization event](#) for physical SError interrupt exceptions.

---

### 2.4.2 Extension for synchronization at exception entry and return

<code>I<sub>DYZRN</sub></code>	The <a href="#">FEAT_IESB</a> feature adds a control bit to each AArch64 <code>SCTLR_ELx</code> System register to insert an implicit <a href="#">Error synchronization event</a> at exception entry and exception return. For the register field descriptions, see [1].
<code>R<sub>DPSJR</sub></code>	The rules in this section apply when <a href="#">FEAT_IESB</a> is implemented.
<code>I<sub>WDSBL</sub></code>	An implicit <a href="#">Error synchronization event</a> has no effect on <code>DISR_EL1</code> or <code>VDISR_EL2</code> .
<code>R<sub>KJWNS</sub></code>	When <a href="#">FEAT_DoubleFault</a> is implemented, and the Effective value of <code>SCR_EL3.NMEA</code> is <code>0b1</code> , <code>SCTLR_EL3.IESB</code> is ignored and its Effective value is <code>0b1</code> .



## Synchronization on exception entry

- R<sub>RNZWY</sub>** For each value of ELx in EL1, EL2, EL3, if all of the following are true, then each exception that is taken to ELx generates an [Error synchronization event](#):
- ELx is using AArch64.
  - The Effective value of SCTLR\_ELx.IESB is 0b1.
- R<sub>RPBWR</sub>** For each value of ELx in EL1, EL2, EL3, if all of the following are true, then executing a DCPSx instruction generates an [Error synchronization event](#):
- The PE is in Debug state.
  - ELx is using AArch64.
  - The Effective value of SCTLR\_ELx.IESB is 0b1.
- R<sub>JQSKQ</sub>** If an SError interrupt exception is taken to the Exception level ELy as a result of the [Error synchronization event](#) generated on exception entry by the [FEAT\\_IESB](#) mechanism, then all the following occur:
- The PE sets the ESR\_ELy.IESB bit in the SError interrupt exception syndrome to 0b1.
  - The preferred return address for the SError interrupt exception is the exception vector address for the original exception.

---

### Note

ELy might be the same Exception level as ELx.

---

- I<sub>FWZHV</sub>** If SError interrupt exceptions are masked at ELx, any SError interrupt made pending by the [Error synchronization event](#) stays pending.
- U<sub>MMVJW</sub>** The prioritization of asynchronous interrupts is IMPLEMENTATION DEFINED. This means that an implementation might choose to behave as if the SError interrupt was taken before the implicit [Error synchronization event](#), if the SError interrupt was not masked, taking the SError interrupt in place of the exception.
- In this case, ESR\_ELy.IESB is set to 0b0 and the reported [PE error state](#) correctly indicates, for instance, whether software can [recover execution](#) from the preferred return address for the SError interrupt in ELR\_ELy.
- When [FEAT\\_DoubleFault](#) is implemented, Arm recommends that the implicit Error synchronization event is inserted before taking an exception to EL3.

## Synchronization on exception return

- R<sub>SKRCR</sub>** For each value of ELx in EL1, EL2, EL3, if all of the following are true, then executing an exception return instruction at ELx generates an [Error synchronization event](#):
- The instruction does not generate any exception.
  - ELx is using AArch64.
  - The Effective value of SCTLR\_ELx.IESB is 0b1.

---

### Note

On an illegal return event the exception return instruction sets PSTATE.IL to 0b1, which causes the next instruction to generate an Illegal State exception. The exception return instruction does not generate the exception.

---

- R<sub>CVPDN</sub>** For each value of ELx in EL1, EL2, EL3, if all of the following are true, then executing an DRPS instruction at



ELx generates an [Error synchronization event](#):

- The PE is in Debug state and the instruction does not generate any exception.
- ELx is using AArch64.
- The Effective value of SCTLR\_ELx.IESB is 0b1.

**R<sub>GXQYD</sub>** Any SError interrupt exception taken as part of the [Error synchronization event](#) terminates execution of the instruction.

**R<sub>LPKVM</sub>** If an SError interrupt exception is taken to an Exception level, ELy, as a result of the [Error synchronization event](#) generated on exception return by the [FEAT\\_IESB](#) mechanism, then all the following occur:

- The PE sets the ESR\_ELy.IESB bit in the SError interrupt exception syndrome to an IMPLEMENTATION DEFINED choice of 0b0 or 0b1.
- The preferred return address for the SError interrupt is the address of the `ERET` instruction.

**I<sub>JZHDB</sub>** If SError interrupt exceptions are masked at ELx, any SError interrupt made pending by the [Error synchronization event](#) stays pending.

## 2.5 Virtual SError interrupts

ℙ<sub>LSSCN</sub>

When implemented, EL2 provides a virtual SError interrupt. The RAS Extension provides:

- Virtual syndrome registers, VESR\_EL2 and VDFSR, that allow a hypervisor to specify the ESR\_EL1 or DFSR value reported to a guest Operating System on taking a virtual SError interrupt. See [1] for descriptions of VESR\_EL2 and VDFSR.
- Support for EL0 or EL1 to isolate a virtual SError interrupt as if it were a physical SError interrupt. See [ESB and Virtual SError interrupt exceptions](#).

## 2.6 Error records in the PE

<code>I_KMMPK</code>	A component that records detected errors is called a <b>node</b> by the <i>RAS System Architecture</i> . Each <b>node</b> implements one or more <b>error records</b> .
<code>R_VNLPC</code>	It is IMPLEMENTATION DEFINED whether the processor that implements a PE implements any <b>nodes</b> .
<code>R_XKDRX</code>	A PE implementing the RAS Extension might implement the System register interface to <b>nodes</b> .
<code>I_SCVSB</code>	The System register interface to <b>nodes</b> is not restricted to accessing only PE <b>nodes</b> .
<code>I_ZRKKQ</code>	When an error is recorded by a PE <b>node</b> , one or more of the following might be generated, according to the configuration of the <b>node</b> : <ul style="list-style-type: none"> <li>• A <b>fault handling interrupt</b>.</li> <li>• An <b>error recovery interrupt</b>.</li> <li>• A <b>critical error interrupt</b>.</li> <li>• An <b>in-band error response</b>.</li> </ul> <p>See also:</p> <ul style="list-style-type: none"> <li>• <i>Error record System register view</i></li> <li>• <i>RAS System Architecture</i></li> <li>• <i>Nodes</i></li> </ul>

### 2.6.1 Error record System register view

<code>I_SLVDW</code>	If the System register interface to a <b>node</b> is implemented, software accesses the <b>error records</b> of the <b>node</b> using Error record System registers.
<code>R_BYLZQ</code>	The number of <b>error records</b> that can be accessed using the System registers is IMPLEMENTATION DEFINED, and might be zero. The <code>ERRIDR_EL1</code> and <code>ERRIDR</code> registers indicate the highest numbered index of the <b>error records</b> that can be accessed using System registers, plus one.
<code>I_NWBNQ</code>	The AArch64 Error record System registers are those registers with an <code>ERX*_EL1</code> mnemonic. See <i>Using AArch64 System registers</i> .  The AArch32 Error record System registers are those registers with an <code>ERX*</code> mnemonic. See <i>Using AArch32 System registers</i> .  These registers are defined in [1].
<code>I_VVMCQ</code>	The <b>error record</b> register contents are described by <i>Error record registers, including memory mapped view</i> .
<code>R_ZBCFZ</code>	If <code>FEAT_RASv1p1</code> is implemented, all <b>error records</b> accessible through System registers implement RAS System Architecture v1.1.
<code>S_VBBNY</code>	To access an <b>error record</b> , software: <ol style="list-style-type: none"> <li>(1) Sets the error selection register, <code>ERRSELR_EL1.SEL</code> or <code>ERRSELR.SEL</code>, to the index of the record being accessed.</li> <li>(2) Accesses the <b>error record</b> using the <code>ERX*_EL1</code> or <code>ERX*</code> System registers.</li> </ol>
<code>I_WKXSB</code>	The <b>error records</b> accessed through the System registers might be accessible only to the PE associated with those System registers, or they might be shared and therefore accessible to other PEs through either System registers or as a memory-mapped component.  See also:

- [Synchronization and error record accesses.](#)
- [Error record registers, including memory mapped view](#)

### Fields in VESR\_EL2, VDFSR, DISR(\_EL1), and VDISR(\_EL2)

ISLNMV

ESR\_ELx, HSR, DFSR, VESR\_EL2, VDFSR, DISR\_EL1, DISR, VDISR\_EL2, and VDISR are error syndrome registers that are written with either a syndrome by hardware on taking or deferring a physical SError interrupt, or with a virtual syndrome value provided by software for a virtual SError interrupt, as applicable.

For a given implementation:

- If ESB never synchronizes any errors, then DISR\_EL1.A and DISR.A might be RES0.
- The error syndrome registers are capable of storing any syndrome value that might be reported by hardware on taking a physical error exception.
- If any of ESR\_ELx[24:0], HSR[11:9], and DFSR[15:14,12] is not used and always set to zero by hardware on taking a physical SError interrupt exception or synchronous External Abort exception, it can be RES0 in that syndrome register.
- A bit that is not used and always set to zero or always set to one by hardware on taking a physical SError interrupt is permitted to be RES0 or RES1 respectively in the corresponding other syndrome registers. See [Table 2.2](#).

In [Table 2.2](#), the bit described in the left-hand column is permitted to be RES0 or RES1 if the corresponding bit is always set to zero or always set to one (respectively) on taking an SError interrupt in all of the registers listed in the other columns marked *Yes* on that row.

**Table 2.2: Permitted relaxations for bits in error syndrome registers**

Bit that is permitted to be RES0 or RES1	ESR_ELx[x], x ∈ [24:0]	HSR[x], x ∈ [11:9]	DFSR[x], x ∈ [15:14,12]
VESR_EL2[x]	Yes	-	Yes
VDISR_EL2[x]	Yes	-	Yes
DISR_EL1[x]	Yes	-	-
VDFSR[x]	-	-	Yes
VDISR[x]	-	-	Yes
DISR[x]	-	Yes	Yes

## Chapter 3

# RAS System Architecture

I <sub>XKHGG</sub>	The <i>Reliability, Availability, Serviceability</i> (RAS) System Architecture provides a framework for building RAS features in a system. It provides a reusable component architecture for components that can detect and record errors, and signal them to a <i>Processing element</i> (PE).
R <sub>DKJPB</sub>	A <i>node</i> is a RAS System Architecture component that records an error detected or consumed by a system component.
I <sub>NTRXQ</sub>	A RAS System Architecture implementation includes one or more <a href="#">nodes</a> . The RAS System Architecture does not require that all components in a system implement the RAS System Architecture or appear as a <a href="#">node</a> .
I <sub>FPMKF</sub>	The RAS System Architecture does not prescribe the level of reliability, availability, and serviceability in the system. The RAS features that the system includes, for example to detect, correct, contain, or defer errors, are IMPLEMENTATION DEFINED.
I <sub>LJWMZ</sub>	The RAS features and behavior of components that do not implement the RAS System Architecture are IMPLEMENTATION DEFINED.
I <sub>QTZCK</sub>	Arm recommends that all errors are reported to a RAS System Architecture <a href="#">node</a> to enable error recovery and fault handling.
I <sub>HTDRT</sub>	This section describes the behavior of RAS System Architecture <a href="#">nodes</a> and other components that implement the RAS System Architecture.

## 3.1 Nodes

R<sub>RDHHP</sub> A component might implement one or more [nodes](#).

The RAS System Architecture defines the following common features for a [node](#):

R<sub>XMFKF</sub> **Error detection and correction**

The level of error correction and detection implemented at a component is IMPLEMENTATION DEFINED.

A [node](#) might include the control to disable error reporting and recording of detected errors, for example while software initializes the component.

It is IMPLEMENTATION DEFINED whether the [node](#) fully disables error detection and correction when reporting and recording are disabled.

See [Detecting and consuming errors](#).

R<sub>FNBYQ</sub> **Fault handling interrupt**

Asynchronous reporting of all or some recorded errors by an interrupt, that is, [Corrected errors](#), [Deferred errors](#), and [Uncorrected errors](#). It is IMPLEMENTATION DEFINED whether a [node](#) provides a single control for all errors, or a first control for [Corrected errors](#) and a second control for all other detected errors.

See [Fault handling interrupt](#).

R<sub>QQRSQ</sub> **Corrected error counter**

It is IMPLEMENTATION DEFINED whether a [node](#) implements a counter for counting [Corrected errors](#). Software might poll the error counter or initialize the counter with a threshold value and receive an interrupt when the counter overflows. A counter overflows when incrementing the counter results in unsigned integer overflow.

It is IMPLEMENTATION DEFINED which [Corrected errors](#) are counted.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether [Deferred errors](#) and [Uncorrected errors](#) are counted by the Corrected error counter.

See [Standard format Corrected error counter](#).

R<sub>WFWCL</sub> **Timestamps**

It is IMPLEMENTATION DEFINED whether a [node](#) records a timestamp in each [error record](#).

See [Timestamp extension](#).

R<sub>ZMMBH</sub> **In-band error response (external abort)**

In-band signaling of detected [Uncorrected errors](#) to the consumer of the error. It is also referred to as an external abort. [Corrected errors](#) and [Deferred errors](#) are not reported by such means.

See [In-band error response signaling \(external aborts\)](#).

R<sub>VHDZW</sub> **Error recovery interrupt**

Asynchronous (out-of-band) reporting of recorded [Uncorrected errors](#) by an interrupt. The interrupt can be used for error recovery, fault handling, or both. [Corrected errors](#) are not reported by this means. It is IMPLEMENTATION DEFINED whether the [node](#) provides the control to enable [Deferred errors](#) to be reported in this way. If the control is not provided, [Deferred errors](#) are not reported by this means.

See [Error recovery interrupt](#).

R<sub>BJNDJ</sub> **Critical Error interrupt**

Critical error interrupts provide a mechanism for a [node](#) to report a critical error condition to a system controller for error recovery.

See [Critical error interrupt](#).

R<sub>REFNHX</sub>**Records**

A **node** implements one or more standard **error records**. When an error is detected or consumed, syndrome about the error is written to an **error record**.

See *Standard error record*.

I<sub>WRWMK</sub>

A **node** might implement some or all of these features.

R<sub>YHBGJ</sub>

The first standard **error record** for a **node** contains:

- An identification register, **ERR<n>FR**, that describes the implemented features of the **node**.
- The **ERR<n>CTLR** register to enable or disable the features.

R<sub>JMRML</sub>

A **node** has a single **ERR<n>FR** and a single **ERR<n>CTLR** register.

R<sub>CWWXN</sub>

If the **node** implements multiple **error records**, each **error record** has the same features and all **error records** share the controls.

**Note**

If a component requires multiple sets of controls, the component implements multiple **nodes**.

R<sub>GSGNZ</sub>

For each **node**, it is IMPLEMENTATION DEFINED whether the fault and error reporting mechanisms apply to both reads and writes, or whether the mechanisms can be individually controlled for reads and writes.

**3.1.1 Multiple error records per node**R<sub>MRKT</sub>

Each **node** contains at least one **error record**.

I<sub>YKNTD</sub>

A **node** might implement multiple **error records** for one or more of the following purposes:

- To record different types of error in different **error records**.
- To record errors from different components, or different **FRUs** accessed by a component, in different **error records**.
- To record multiple errors.

U<sub>GMKHP</sub>

Using a single **error record** is efficient for the implementation.

However, consider an example **node** for an SoC memory controller component that records errors detected within both:

- An internal buffer that acts as a queue for memory accesses.
- An external memory module, that is, an external *Field Replaceable Unit (FRU)*.

In this **node**, using a single error record for errors from either source might lead to the following scenarios:

- (1)
  - A **Corrected error** is detected in the internal buffer and recorded in the error record.
  - Before software processes the error record, an **Uncorrected error** is detected in the external **FRU**.
- (2)
  - A **Corrected error** is detected in the external **FRU** and recorded in the error record.
  - Before software processes the error record, an **Uncorrected error** is detected in the external **FRU**.

In both scenarios, the second error overwrites the syndrome for the first error, because *Writing the error record* requires this. It is IMPLEMENTATION DEFINED what information, if any, is retained for the first error in the IMPLEMENTATION DEFINED parts of the syndrome.

This means the two scenarios might be indistinguishable to software. In particular any indication of where the **Corrected error** was detected in the syndrome for the first error might be overwritten by the second error. If this

is the case, software will need to treat the two scenarios identically, that is, as if there was a corrected internal error.

However, an internal error might be considered more significant than an external FRU error. For example, because the external FRU is *field-replaceable* whereas the SoC is not. Implementing separate error records for the internal buffer and external FRU would avoid this issue.

Implementations should therefore consider the impact such choices might have on the serviceability and availability of the system.

If a single node implements multiple error records, then all of the following are true:

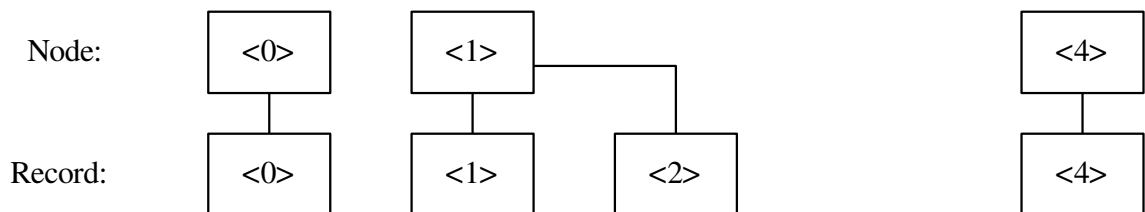
- $R_{VRMSL}$  • The error records are indexed sequentially within a group of error records starting from the first error record for the node.
- $R_{HCXWW}$  • For each error record other than the first error record for the node, the following are true:
  - The  $ERR<n>FR.ED$  field is 0b00.
  - $ERR<n>FR[63:2]$  are RES0.
  - The  $ERR<n>CTLR$  register is RES0.

$R_{RFPVW}$  A group of error records consists of the error records of one or more nodes.

$R_{DBPFH}$  A group of error records might be sparsely populated. Locations relating to unimplemented error records are RAZ/WI, meaning that they have an  $ERR<n>FR$  register that reads as zero.

See *Nodes*.

$I_{TPDYQ}$  An example of a group of error records containing five error records owned by three nodes might be arranged as shown in Figure 3.1:



**Figure 3.1: A group containing five error records owned by three nodes**

- Node <0> owns a single error record: <0>.  $ERR0FR$  describes the features for this node, and  $ERR0CTLR$  contains the controls for this node.
- Node <1> owns two error records: <1> and <2>.
  - $ERR1FR$  describes the features for this node, and  $ERR1CTLR$  contains the controls for this node.
  - $ERR2FR.ED$  is 0b00 and  $ERR2CTLR$  is not implemented.
- Error record <3> is not implemented.  $ERR3FR.ED$  is 0b00 and  $ERR3CTLR$ ,  $ERR3STATUS$ ,  $ERR3ADDR$ , and  $ERR3MISC<m>$  are not implemented.
- Node <4> owns a single error record: <4>.  $ERR4FR$  describes the features for this node, and  $ERR4CTLR$  contains the controls for this node.
- If the group of error records is accessed using a memory-mapped view then  $ERRDEVID.NUM$  is 5.
- If the group of error records is accessed using System registers then  $ERRIDR_EL1.NUM$  is 5.

### 3.1.2 Detecting and consuming errors



<code>R_QZHDT</code>	<p>A component detects an error when it detects that a deviation from <a href="#">correct service</a> has occurred or will occur. For example, including but not limited to when any of the following occurs that would not be permitted to occur had the fault not been activated:</p> <ul style="list-style-type: none"> <li>• A <a href="#">corrupt</a> value has been or will be passed to a consumer.</li> <li>• A transaction or other operation occurs or will occur that should not occur.</li> <li>• A transaction or other operation that should occur does not occur or will not occur.</li> <li>• A loss of uniprocessor semantics or any other loss of coherency in a multiprocessor coherent system is or will be observed. See <code>I_SVZKY</code>.</li> <li>• The timing and/or order of transactions or other operations has been or will be changed.</li> <li>• A latent error has become or will become undetectable. See <code>I_QXPLK</code>.</li> </ul>
<code>I_SVZKY</code>	<p>Examples of a loss of uniprocessor semantics or other loss of coherency that might occur because of an error include:</p> <ul style="list-style-type: none"> <li>• A cache loses data that it holds in a modified state.</li> <li>• A cache writes back unmodified data.</li> </ul> <p>An example that should not occur is when a partial write to the <a href="#">protection granule</a> of a cache location holding <a href="#">poison</a> occurs, and the cache later invalidates the line without writing back the <a href="#">poison</a> value.</p> <p>For example, if a cache fetches data from memory and receives poison, and subsequently, a partial write to that location is insufficient to clean the location of the poison and the location remains poisoned, then the cache should treat it as modified, even though it appears that the write did not modify the location. That is, the cache should take ownership of the location and write-back poison when the location is evicted from the cache. Otherwise if the original error was transient and later disappears from memory, the location reverts to the unmodified value, silently propagating the error.</p>
<code>I_QXPLK</code>	<p>An example of a latent error becoming undetectable includes when a <a href="#">poison</a> value indicating a <a href="#">deferred error</a> is lost at the interface between domains. For example, because a <a href="#">poison</a> value is passed to a component that does not support poisoning.</p> <p>An example of a latent error becoming undetectable that should not occur is when a <a href="#">poison</a> value is lost by a partial write to the <a href="#">protection granule</a>. In this case, the partial write should leave the <a href="#">protection granule</a> containing <a href="#">poison</a>.</p>
<code>R_LRSMZ</code>	<p>A component consumes an error that is <a href="#">signaled</a> to the component in response to a memory access, cache maintenance operation, or other <a href="#">transaction</a> initiated by the component as one of:</p> <ul style="list-style-type: none"> <li>• An <a href="#">in-band error response</a>.</li> <li>• A <a href="#">deferred error</a>.</li> </ul>
<code>R_WXPDN</code>	<p>When an error is detected or consumed by a component, the error is <i>reported</i> to one or more <a href="#">nodes</a>.</p> <p>It is IMPLEMENTATION DEFINED whether:</p>
<code>R_VYRXT</code>	<ul style="list-style-type: none"> <li>• A Requester that consumes a <a href="#">signaled</a> detected error reports the consumed error.</li> </ul>
<code>R_LRQSG</code>	<ul style="list-style-type: none"> <li>• Errors are reported when a detected error is propagated between components.</li> </ul>
<code>R_WDJGD</code>	<ul style="list-style-type: none"> <li>• All corrected errors are reported.</li> </ul>
<code>R_GVPMK</code>	<ul style="list-style-type: none"> <li>• Errors detected on hardware speculation are reported.</li> </ul>
<code>R_GCDCL</code>	<p>It is IMPLEMENTATION DEFINED whether the <a href="#">node</a> or nodes that an error is reported to are one or more of the following:</p>

- The same component that detected the error.
- The consumer of the [transaction](#) that consumes a detected error [signaled](#) by the producer of the transaction which detected the error. Syndrome information might be passed with the [signaled](#) detected error to the consumer.
- Another component that neither detected nor consumed the error. For example, a [node](#) whose purpose is to record errors for other components. Such a [node](#) might comprise one record for each component for which it is recording an error, or a number of shared records, where each record identifies the originating component, or some other arrangement.

When an error is detected or consumed by a component:

R<sub>LBHMF</sub>

- If the error can be [corrected](#):
  - The error is [corrected](#).
  - Optionally, the detected error is [reported](#) to a [node](#), the node [records](#) a [Corrected error](#), and if implemented and [enabled](#), a [fault handling interrupt](#) is raised.
  - If the error is detected on a read access by a consumer, corrected data is returned to the consumer.

R<sub>LMCVC</sub>

- If the error cannot be [corrected](#) and can be [deferred](#):
  - The error is [deferred](#). For example, the location being accessed is [poisoned](#) or poisoned data is returned to the consumer.
  - The error is [reported](#) to a [node](#) and the node [records](#) a [Deferred error](#).
  - If implemented and [enabled](#), a [fault handling interrupt](#) is raised.
  - If implemented and [enabled](#), an [error recovery interrupt](#) is raised.

Note: An error cannot be deferred to a component that does not accept deferred errors.

R<sub>LKCNC</sub>

- If the error cannot be [corrected](#) and cannot be [deferred](#):
  - The error is [reported](#) to a [node](#) and the node [records](#) an [Uncorrected error](#).
  - If implemented and [enabled](#), a [fault handling interrupt](#) is raised.
  - If implemented and [enabled](#), an [error recovery interrupt](#) is raised.
  - If the error is detected on an access by a consumer, and if implemented and [enabled](#), a [in-band error response](#) is returned to the consumer.
  - If the component is unable to continue operation, it might enter a [service failure mode](#).

I<sub>NJHPF</sub>

The criteria by which a component determines when it can [correct](#) or [defer](#) an error are IMPLEMENTATION DEFINED. For example, if the error is detected in response to an access by a consumer that is incapable of receiving a [deferred error](#) response, then it is not possible to [defer](#) the error to the consumer.

R<sub>LTBDP</sub>

When an error is reported to a [node](#), the [node](#) records syndrome information for the error in a standard [error record](#).

I<sub>SNNZR</sub>

Arm recommends that hardware records sufficient information to:

- Determine whether error recovery is possible, if the error was not corrected by hardware.
- Allow fault analysis to find trends in the faults. This information is IMPLEMENTATION DEFINED but might include the location of the data.
- Allow identification of a [FRU](#).

I<sub>JNMFY</sub>

The [node](#) registers might also contain control registers for error detection, correction and reporting at the component.

I<sub>WMVTN</sub>

[Corrected errors](#) can be recorded by counting each [corrected error](#). Counting might be done by either software or hardware. The fault handling process compares the [corrected error](#) rate with a threshold value to determine whether to take action.

I\_QGNHF

*Standard format Corrected error counter* and *corrected error counter* describe an optional standard hardware mechanism for counting errors.

I\_GGQSR

The details of any *service failure mode* are IMPLEMENTATION DEFINED. For example:

- A component that fetches data from memory and processes that data might halt processing and await servicing by an application processor when it receives an *in-band error response*. This is a form of *service failure mode*.
- When a PE takes an *error exception* and executes an error handler, this is also a form of *service failure mode*.

The component might implement multiple functions, some of which can be in a *service failure mode* while others continue to operate, or the service failure mode might affect multiple or all functions of the component.

See also:

- *Standard error record*
- *Error recovery interrupt*
- *Fault handling interrupt*
- *In-band error response signaling (external aborts)*
- *Standard format Corrected error counter*

## 3.2 Standard error record

R <sub>GTCQJ</sub>	The RAS System Architecture defines a standard <i>error record</i> and a mechanism to access <a href="#">error records</a> as System registers or as a memory-mapped component.
R <sub>XGGTZ</sub>	<p>The standard <a href="#">error record</a> contains:</p> <ul style="list-style-type: none"> <li>• A status register, <a href="#">ERR&lt;n&gt;STATUS</a>, for common status fields, such as the type and coarse characterization of the error.</li> <li>• An optional address register, <a href="#">ERR&lt;n&gt;ADDR</a>.</li> <li>• IMPLEMENTATION DEFINED status registers, referred to as <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a>. Arm recommends these are used for: <ul style="list-style-type: none"> <li>– Identifying a <a href="#">FRU</a>.</li> <li>– Locating the error within the <a href="#">FRU</a>.</li> <li>– Optionally, a <a href="#">corrected error counter</a> or counters for software to poll the rate of <a href="#">Corrected errors</a>.</li> <li>– Optionally, a timestamp value for when the error was recorded.</li> </ul> </li> </ul>
R <sub>MQPFL</sub>	<p>If RAS System Architecture v1.0 is implemented, there are two <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> for each <a href="#">error record</a>:</p> <ul style="list-style-type: none"> <li>• <a href="#">ERR&lt;n&gt;MISC0</a>.</li> <li>• <a href="#">ERR&lt;n&gt;MISC1</a>.</li> </ul>
R <sub>QCKVG</sub>	<p>If RAS System Architecture v1.1 is implemented, there are 4 <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> for each <a href="#">error record</a>:</p> <ul style="list-style-type: none"> <li>• <a href="#">ERR&lt;n&gt;MISC0</a>.</li> <li>• <a href="#">ERR&lt;n&gt;MISC1</a>.</li> <li>• <a href="#">ERR&lt;n&gt;MISC2</a>.</li> <li>• <a href="#">ERR&lt;n&gt;MISC3</a>.</li> </ul>
<hr/> <p><b>Note</b></p> <p>The RAS System Architecture permits the implementation of <a href="#">ERR&lt;n&gt;MISC2</a> and <a href="#">ERR&lt;n&gt;MISC3</a> in implementations of the RAS System Architecture v1.0.</p> <hr/>	
R <sub>DXZPX</sub>	An <a href="#">error record</a> might include additional IMPLEMENTATION DEFINED controls and identification registers.
I <sub>PVYZG</sub>	<a href="#">Error record System register view</a> defines System registers for accessing a group of <a href="#">error records</a> .
I <sub>PBJTL</sub>	<a href="#">Memory-mapped view</a> defines reusable formats for a memory-mapped views of <a href="#">error records</a> . Use of reusable formats by any component in the system is OPTIONAL.
I <sub>BNPZB</sub>	The format of the <a href="#">error record</a> registers is the same for both access mechanisms.
R <sub>WDSFZ</sub>	<a href="#">Error records</a> are preserved over Error Recovery reset. This allows for a diagnosis after system failure.

### 3.2.1 Component error states

R <sub>VWSSX</sub>	When a <a href="#">node</a> records an error, the <i>component error state</i> is recorded in the <a href="#">error record</a> .
R <sub>LBPN</sub>	<p>For a standard <a href="#">error record</a>, the <a href="#">component error state</a> types that can be recorded are:</p> <ul style="list-style-type: none"> <li>• <a href="#">Corrected error</a> (CE).</li> <li>• <a href="#">Deferred error</a> (DE).</li> <li>• <a href="#">Uncorrected error</a>.</li> </ul>

R<sub>KFPDF</sub>

If and only if all of the following are true, then on recording an error, the **component error state** is recorded as *Corrected error* (CE):

- The error was **corrected**.
- The error has not been **silently propagated**.
- The component has not entered as **service failure mode** and continues to operate.
- The implementation has not elected to record the **component error state** as **Deferred error**, or **Uncorrected error**.

In normal circumstances, the error no longer infects the state of the component. However, in the case of a persistent correctable fault, or other rare IMPLEMENTATION DEFINED circumstances, the error might remain **latent** in the component.

R<sub>XJFMG</sub>

If and only if all of the following are true, then on recording an error, the **component error state** is recorded as *Deferred error* (DE):

- At least one of the following are true:
  - The error was not **corrected**, and was **deferred**.
  - The error was **corrected**, and the implementation elected to record the **component error state** as **Deferred error**.
- The error has not been **silently propagated**.
- The error might be **latent** in the system.
- It is IMPLEMENTATION DEFINED whether the error continues to infect the state of the component or whether it has been **deferred** to a consumer.
- The component has not entered as **service failure mode** and continues to operate.
- The implementation has not elected to record the **component error state** as **Uncorrected error**.

---

#### Note

A **Deferred error** might be recorded for an error that cannot be corrected. However, for the purposes of the **component error state** taxonomy, **Deferred error** is classified separately from **Uncorrected error**.

---

R<sub>KJTQQ</sub>

If and only if all of the following are true, then on recording an error, the **component error state** is recorded as *Uncorrected error*:

- At least one of the following are true:
  - The error was not **corrected** and not **deferred**.
  - The error might have been **silently propagated**.
  - The component has entered as **service failure mode** and does not continue to operate the function that consumed the error.
  - The error was either **corrected** or **deferred**, and the implementation elected to record the **component error state** as **Uncorrected error**.
- The error is **latent** in the system.

R<sub>WHGSP</sub>

An **Uncorrected error** is recorded as one of the following sub-types:

- **Uncontainable error** (UC).
- **Unrecoverable error** (UEU).
- **Recoverable error** or **Signaled error** (UER).
- **Restartable error** or **Latent error** (UEO).

**R<sub>PHLQO</sub>** If any of the following are true, then on recording a **Uncorrected error**, the **component error state** is recorded as **Uncontainable error (UC)**:

- The error might have been **silently propagated** by the component.
- The implementation has elected to record the error as **Uncontainable error**.

If the error cannot be isolated, the system must be shut down to avoid catastrophic failure.

**R<sub>CTYHC</sub>** If and only if all of the following are true, then on recording a **Uncorrected error**, the **component error state** is recorded as **Unrecoverable error (UEU)**:

- The error has not been **silently propagated** by the component.
- Either of the following are true:
  - The component has halted operation (entered a **service failure mode**) of the function that consumed the error. The component determines that software will not be able to recover operation of the function.
  - The implementation has elected to record the error as **Unrecoverable error**.
- The implementation has not elected to record the error as **Uncontainable error**.

**R<sub>CNERY</sub>** If and only if all of the following are true, then on recording a **Uncorrected error**, the **component error state** is recorded as **Signaled error (UER)**:

- The error was produced at the component.
- The error has not been **silently propagated** by the component.
- The error has been or might have been consumed, and was not recorded as a **Deferred error**.
- The implementation has not elected to record the error as **Unrecoverable error**, or **Uncontainable error**.

**R<sub>FFTXZ</sub>** If and only if all of the following are true, then on recording a **Uncorrected error**, the **component error state** is recorded as **Latent error (UEO)**:

- The error was produced at the component.
- The error has not been **propagated** by the component, silently or otherwise.
- The implementation has not elected to record the error as **Deferred error**, **Unrecoverable error**, or **Uncontainable error**.

That is, the error was detected but not consumed, and was not recorded as a **Deferred error**.

---

#### Note

The producer is usually unable to determine whether a consumer has architecturally consumed the error. An error might be recorded as **Latent error** if it has definitely not been **propagated** to any consumer, and as **Signaled error** otherwise.

---

**R<sub>QTYFD</sub>** If and only if all of the following are true, then on recording a **Uncorrected error**, the **component error state** is recorded as **Recoverable error (UER)**:

- The error has not been **silently propagated** by the component.
- The component has halted operation (entered a **service failure mode**) of the function that consumed the error.
- Either of the following is true:
  - The component is reliant on consuming the corrupted data to continue operation of the function that

consumed the error. The component determines that software will be able to recover operation of the function if it locates and repairs the error.

- The implementation has elected to record the error as [Recoverable error](#).
- The implementation has not elected to record the error as [Deferred error](#), [Unrecoverable error](#), or [Uncontainable error](#).

R<sub>CFZTH</sub>

If and only if all of the following are true, then on recording a [Uncorrected error](#), the [component error state](#) is recorded as *Restartable error* (UEO):

- The error has not been [silently propagated](#) by the component.
- The component has halted operation (entered a [service failure mode](#)) of the function that consumed the error.
- The component determines that it does not rely on the corrupted data, and so can recover operation even if software does not locate and repair the error.
- The implementation has not elected to record the error as [Deferred error](#), [Unrecoverable error](#), or [Uncontainable error](#).

I<sub>TVJNM</sub>

The [component error state](#) types are summarized by [Figure 3.2](#). [Figure 3.2](#) assumes the component supports the resulting [component error state](#) and the implementation never elects to record an error as a different [component error state](#) when permitted.

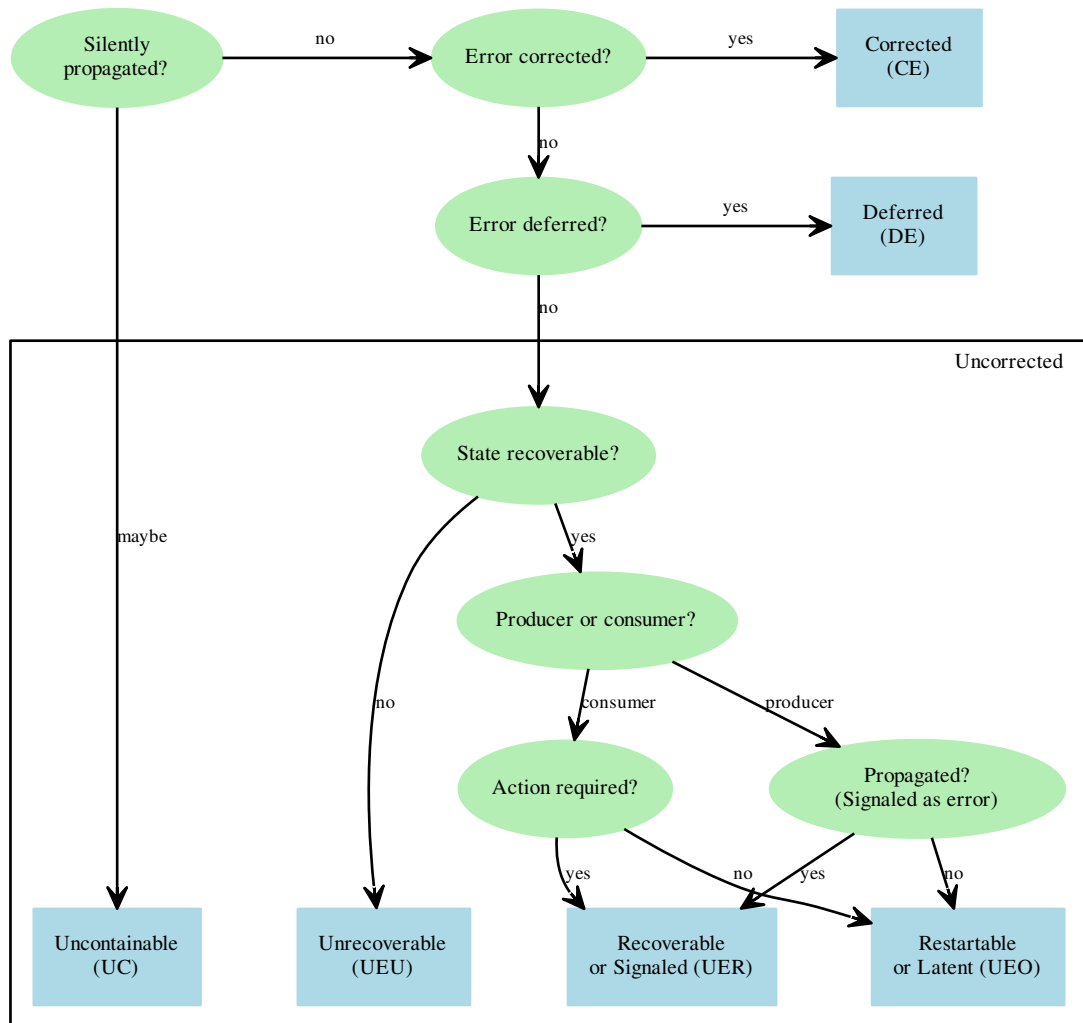


Figure 3.2: Component error state types

### 3.2.2 Writing the error record

R<sub>MDXXV</sub>

When a new error is recorded, the [node](#):

- Does one of the following:
  - [Overwrites](#) the [error record](#) with the syndrome for the new error.
  - [Keeps](#) the syndrome for the previous error.
- Modifies `ERR<n>STATUS.{CE, DE, UE}` to indicate the [component error state](#). See [Component error states and priorities](#).
- Counts the error, if a [corrected error counter](#) is implemented and the error is of a type that the counter counts.

R<sub>BGXQQ</sub>

If counting a [Deferred error](#) or [Uncorrected error](#) causes the counter to overflow, then `ERR<n>STATUS.OF` is set as it would be for a [Corrected error](#) that causes [corrected error counter](#) overflow. However, if the RAS System Architecture requires that recording the [Deferred error](#) or [Uncorrected error](#) sets the `ERR<n>STATUS.OF` flag to `0b1`, then this flag is also set to `0b1` even if the error is counted and the [corrected error counter](#) does not overflow.



## Component error states and priorities

R<sub>PXCDZ</sub>

The highest priority recorded [component error state](#) type is recorded in the [ERR<n>STATUS](#).{V, CE, DE, UE, UET} fields, as shown in [Table 3.1](#).

In [Table 3.1](#), V, CE, DE, UE, UET refer to fields in [ERR<n>STATUS](#).

**Table 3.1: Encoding the highest priority component error state**

V	CE	DE	UE	UET	Highest priority <a href="#">component error state</a> type	Mnemonic
0	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	None	-
1	0b00	0	0	UNKNOWN	None	-
1	!= 0b00	0	0	UNKNOWN	Corrected error	CE
1	X	1	0	UNKNOWN	Deferred error	DE
1	X	X	1	0b10	Uncorrected error: Latent error or Restartable error	UEO
1	X	X	1	0b11	Uncorrected error: Signaled error or Recoverable error	UER
1	X	X	1	0b01	Uncorrected error: Unrecoverable error	UEU
1	X	X	1	0b00	Uncorrected error: Uncontainable error	UC

I<sub>QHBGV</sub>

The [component error state](#) types implemented at a [node](#) are IMPLEMENTATION DEFINED. An implementation might only include a simplified subset of these [component error state](#) types.

A [node](#) can always elect to record:

- UEO as any of UER, UEU, or UC.
- UER as either UEU or UC.
- UEU as UC.

## Prioritizing errors, RAS System Architecture v1.0

R<sub>ZPTXT</sub>

When RAS System Architecture v1.0 is implemented, overwriting depends on the [component error state](#) type of the previous highest priority error and on the [component error state](#) type of the newly recorded error, as shown in [Table 3.2](#).

In [Table 3.2](#):

- Each row corresponds to the highest priority previous [component error state](#) type recorded in the [error record](#).
- Each column corresponds to the [component error state](#) type of the new [detected error](#).

The row and column headings use the mnemonics from [Table 3.1](#), and the following additional abbreviations are used:

- K** Keep. [Keep](#) the previous error syndrome. It is IMPLEMENTATION DEFINED whether [ERR<n>STATUS.OF](#) is set to 0b1 or unchanged.
- O** Overflow. [Keep](#) the previous error syndrome and set [ERR<n>STATUS.OF](#) to 0b1.
- W** Overwrite. [Overwrite](#) with the new error syndrome. It is IMPLEMENTATION DEFINED whether [ERR<n>STATUS.OF](#) is set to 0b0 or unchanged.
- CK** Count and keep. Count the error if a [corrected error counter](#) is implemented, and [keep](#) the previous error syndrome. If the counter overflows, or if no [corrected error counter](#) is implemented, it is IMPLEMENTATION DEFINED whether [ERR<n>STATUS.OF](#) is set to 0b1 or unchanged.

### CWK

Count and overwrite or keep. The behavior is IMPLEMENTATION DEFINED and described by the value of `ERR<q>FR.CEO`, where `<q>` is the index of the first [error record](#) owned by the [node](#):

- 0: Count the error if a [corrected error counter](#) is implemented. [Keep](#) the previous error syndrome.
- 1: Count the error. If `ERR<n>STATUS.OF == 1` before the error is counted, [keep](#) the previous syndrome. Otherwise [overwrite](#) with the new error syndrome.

If counting the error causes unsigned overflow of the counter, or if no [corrected error counter](#) is implemented, `ERR<n>STATUS.OF` is set to `0b1`.

### CW

Count and overwrite. Count the error if a [corrected error counter](#) is implemented, and [overwrite](#) with the new error syndrome. If a [corrected error counter](#) is implemented and counting the error causes unsigned overflow of the counter, `ERR<n>STATUS.OF` is set to an UNKNOWN value. Otherwise, it is IMPLEMENTATION DEFINED whether `ERR<n>STATUS.OF` is set to `0b0` or unchanged.

### WO

Overwrite and overflow. [Overwrite](#) with the new error syndrome. `ERR<n>STATUS.OF` is set to `0b1`.

**Table 3.2: RAS System Architecture v1.0 rules for overwriting error records**

	CE	DE	UEO	UER	UEU	UC
-	CW	W	W	W	W	W
CE	CWK	W	W	W	W	W
DE	CK	O	W	W	W	W
UEO	CK	K	O	WO	WO	WO
UER	CK	K	O	O	WO	WO
UEU	CK	K	O	O	O	WO
UC	CK	K	O	O	O	O

## Prioritizing errors, RAS System Architecture v1.1

R<sub>PNFPB</sub>

When RAS System Architecture v1.1 is implemented, overwriting depends on the [component error state](#) type of the previous highest priority error and on the [component error state](#) type of the newly recorded error, as shown in [Table 3.3](#).

In [Table 3.3](#):

- Each row corresponds to the highest priority previous [component error state](#) type recorded in the [error record](#).
- Each column corresponds to the [component error state](#) type of the new [detected error](#).

The row and column headings use the mnemonics from [Table 3.1](#), and the following additional abbreviations are used:

**W** Overwrite. [Overwrite](#) with the new error syndrome. `ERR<n>STATUS.OF` is unchanged.

### WO

Overwrite and overflow. [Overwrite](#) with the new error syndrome. `ERR<n>STATUS.OF` is set to `0b1`.

**O** Overflow. [Keep](#) the previous error syndrome and set `ERR<n>STATUS.OF` to `0b1`.

If no [corrected error counter](#) is implemented, then all of the following apply:

### CW

Behaves the same as **W**.

### CWO and CO

Behave the same as **O**.

Otherwise, a [corrected error counter](#) is implemented, and all of the following apply:

#### CW

Count and overwrite. [Overwrite](#) with the new error syndrome, and count the error. If counting the error causes unsigned overflow of the counter, set [ERR<n>STATUS.OF](#) to 0b1.

#### CWO

Count, overwrite or keep, and overflow. The behavior is IMPLEMENTATION DEFINED and described by the value of [ERR<n>FR.CEO](#):

- 0: The behavior is the same as **CO**.
- 1: Count the error. If [ERR<n>STATUS.OF](#) == 1 before the error is counted, the behavior is the same as **CO**. Otherwise, the behavior is the same as **CW**.

**CO** Count and overflow. [Keep](#) the previous error syndrome, and count the error. If counting the error causes unsigned overflow of the counter, set [ERR<n>STATUS.OF](#) to 0b1.

**Table 3.3: RAS System Architecture v1.1 rules for overwriting error records**

	CE	DE	UEO	UER	UEU	UC
-	CW	W	W	W	W	W
<b>CE</b>	CWO	WO	WO	WO	WO	WO
<b>DE</b>	CO	O	WO	WO	WO	WO
<b>UEO</b>	CO	O	O	WO	WO	WO
<b>UER</b>	CO	O	O	O	WO	WO
<b>UEU</b>	CO	O	O	O	O	WO
<b>UC</b>	CO	O	O	O	O	O

## Overwriting the error syndrome

RRVGRM

When the [node](#) records an error in an [error record](#) and the previous syndrome is *overwritten* with the new error syndrome:

- Modifies [ERR<n>STATUS](#).{V, CE, DE, UE} to indicate the new [component error state](#), as described by [Table 3.1](#):
  - Fields shown as x in [Table 3.1](#) are unchanged.
  - Other [ERR<n>STATUS](#).{V, CE, DE, UE} fields are set to the value given in [Table 3.1](#).

If the [component error state](#) is [Corrected error](#), then the nonzero value written to [ERR<n>STATUS.CE](#) is IMPLEMENTATION DEFINED and depends on the properties of the Corrected error recorded.

- If the new error is a type of [Uncorrected error](#), [ERR<n>STATUS.UET](#) is set to indicate the [component error state](#) sub-type. See [Component error states and priorities](#).
- The [ERR<n>STATUS](#).{ER, PN, IERR, SERR} syndrome fields are written with the syndrome for the new error.
- If there is an address syndrome for the new error, [ERR<n>STATUS.AV](#) is set to 0b1 and the address is written to [ERR<n>ADDR](#). Otherwise [ERR<n>STATUS.AV](#) is set to 0b0 and [ERR<n>ADDR](#) becomes UNKNOWN.
- If the RAS Timestamp Extension is implemented, a timestamp is recorded in [ERR<n>MISC3](#) and [ERR<n>STATUS.MV](#) is set to 0b1.
- If there is other miscellaneous syndrome for the new error, it is written to the [ERR<n>MISC<m>](#) registers and [ERR<n>STATUS.MV](#) is set to 0b1.

- If there is no additional miscellaneous syndrome for the new error written to the ERR<n>MISC<m> registers, then it is IMPLEMENTATION DEFINED whether ERR<n>STATUS.MV is set to 0b0 or unchanged.
  - If software can determine from the ERR<n>MISC<m> contents that the syndrome is not related to the highest priority error, the ERR<n>STATUS.MV bit is unchanged.
  - Otherwise the ERR<n>STATUS.MV bit is cleared to zero.
- ERR<n>STATUS.V is set to 0b1.

S<sub>XFYQK</sub>

After reading an ERR<n>STATUS register, software has to clear the valid bits in the register to allow new errors to be recorded. During this period a new error might **overwrite** the syndrome for the previously read error. To prevent this, a write, or part of a write is ignored by hardware if fields appear to have been updated. For more information see ERR<n>STATUS.

### Keeping the previous error syndrome

R<sub>BGBBD</sub>

When the previous **error record** is *kept*:

- Sets the applicable one of ERR<n>STATUS.{CE, DE, UE} to indicate the new **component error state**:
  - If **Uncorrected error**, then ERR<n>STATUS.UE is set to 0b1.
  - If **Deferred error**, then ERR<n>STATUS.UE is set to 0b1.
  - If **Corrected error**, then the nonzero value written to ERR<n>STATUS.CE is IMPLEMENTATION DEFINED and depends on the properties of the Corrected error recorded.

The remaining ERR<n>STATUS.{UE, DE, CE} fields are unchanged.

- ERR<n>STATUS.UET is unchanged, even if the new error is a type of **Uncorrected error**.
- ERR<n>STATUS.{ER, PN, IERR, SERR}, ERR<n>ADDR, and ERR<n>STATUS.AV are unchanged.
- If the RAS Timestamp Extension is implemented, the timestamp is not recorded.
- It is IMPLEMENTATION DEFINED whether any of ERR<n>MISC<m> are updated. The contents of ERR<n>MISC<m> are IMPLEMENTATION DEFINED. Therefore, it is possible that some of the information about an otherwise discarded error is recorded in these registers. If data is written to any of ERR<n>MISC<m>, then ERR<n>STATUS.MV is set to 0b1.

### Detecting multiple errors

R<sub>RXQWW</sub>

If multiple errors are simultaneously reported to a **node**, it is IMPLEMENTATION DEFINED whether the **node** behaves:

- As if all errors were recorded, in any order. In this case, the prioritization rules mean that the highest priority error is recorded in the syndrome registers. However, the final value of the syndrome registers might depend on the logical order in which the errors were recorded.
- As if the highest priority error was recorded and one or more of the lower priority errors were not recorded.

R<sub>ZJXMD</sub>

If a **corrected error counter** is implemented, and multiple countable errors are detected simultaneously, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether all the errors are counted.

### 3.2.3 Error syndrome

I<sub>YLHWP</sub>

This section provides additional information for some of the error syndrome fields defined in the standard **error record**.

#### Corrected error field

I<sub>BRMPK</sub>

When the syndrome for a **Corrected error** is recorded, the **node** can indicate through the ERR<n>STATUS.CE error type field one of the following:

- The component or node has determined that the error is transient, or likely to be so.
- The component or node has determined that the error is persistent, or likely to be so.
- The component or node does not support making such a determination or is unable to.

**R<sub>FCQDJ</sub>** The mechanism by which a component or node determines whether a [Corrected error](#) is transient or persistent is IMPLEMENTATION DEFINED.

### Poison indicator

**I<sub>DTYHM</sub>** If supported by a [node](#), when the syndrome for a [Deferred error](#) or [Uncorrected error](#) is recorded, the **ERR<n>STATUS.PN** syndrome field is set to indicate that a [poisoned](#) value was detected.

**R<sub>PNKSH</sub>** When the [node](#) records an error and [overwrites](#) the previous error syndrome, if all of the following are true the **ERR<n>STATUS.PN** syndrome field is set to 0b1, and is set to 0b0 otherwise:

- The component checks a value for an error and detects the value indicates a previously deferred error. For example, the value is a [poisoned](#) value.
- The [node](#) does one of the following:
  - Records the error as an [Uncorrected error](#). For example, because the component does one or more of:
    - \* Enters a [service failure mode](#).
    - \* Signals an [in-band error response](#) to the consumer of the data.
    - \* Propagates the value to a component that does not support [poison](#). This is an [Uncontainable error](#).
  - If the component has deferred the error again, records the error as a [Deferred error](#). See also [Bridges to other architectures](#).

**I<sub>JBDPT</sub>** When a component checks a value and detects an [uncorrectable error](#), and defers the error by generating a [poisoned](#) value, the [node](#) records this as a [Deferred error](#) with **ERR<n>STATUS.PN** set to 0b0.

Therefore when software examines the [error records](#), a **ERR<n>STATUS.PN** value of 0b1 indicates that the component was propagating a previously deferred error, and so the fault did not originate in that component. An **ERR<n>STATUS.PN** value of 0b0 indicates that the fault originated at the component.

**I<sub>QLSMY</sub>** In some *Error Detection Code* (EDC) schemes, a [poisoned](#) value is encoded as a reserved value, one that would not be generated by a detectable corruption of valid data. For example, a SECDED scheme where the poisoned value has a Hamming distance greater than 2 bits from any valid value.

For such a scheme it is IMPLEMENTATION DEFINED whether the component can distinguish a corrupt data value from the poison value. The component might accept and store a poisoned value when an error is deferred to it, but treat it as any other [uncorrectable error](#) when it is accessed, meaning **ERR<n>STATUS.PN** is set to 0b0.

### 3.2.4 Security and Virtualization

**I<sub>BQFLD</sub>** Access to the Error System register view of [error record](#) registers can be controlled using Trap exceptions. See [1].

**R<sub>VXDJW</sub>** If a PE implements System register access to [error records](#) for a component that processes [Secure data](#), then either:

- Software has to configure the Trap exception controls to prevent Non-secure access to the [error records](#).
- The component provides reduced functionality to Non-secure state that does not affect operation in Secure state, or does not provide visibility of Secure data, or both.

**I<sub>JTWK</sub>** The definition of *Secure data* is implementation-specific and depends on the how the information encoded in the data relates to the threat model for the system.

For example, in a typical system that supports both Secure and Non-secure memory, data stored in or related to

Secure memory is considered Secure data, and other data is considered *Non-secure data*.

- R<sub>NLZMH</sub>** If a memory-mapped component processes **Secure data**, then one of the following applies:
- The **error records** are visible only to Secure accesses.
  - The **error records** have reduced visibility to Non-secure accesses, that does not affect operation in Secure state, does not provide visibility of Secure data, or both.
- R<sub>DDWHT</sub>** If a memory-mapped component processes only **Non-secure data**, then it is IMPLEMENTATION DEFINED whether:
- The **error records** are visible to both Non-secure and Secure accesses.
  - It is configurable whether the **error records** are visible to Non-secure accesses.
  - The **error records** are visible only to Secure accesses.
- R<sub>BJKZW</sub>** If the memory-mapped component includes registers to generate *message signaled interrupts* (MSIs) and the component can be programmed by Non-secure accesses, the MSIs do not target Secure addresses.

### 3.2.5 Synchronization and error record accesses

- R<sub>VKYVJ</sub>** When a component reports an error to a **node**, the node updates the **error record** registers and might generate one or more of the following:
- A **fault handling interrupt**.
  - An **error recovery interrupt**.
  - A **critical error interrupt**.
  - An **in-band error response**.
- Each of these might generate an exception at a PE.
- R<sub>VYCRY</sub>** If the PE reads the **error record** registers at the **node**, after taking an exception generated by such a signal from a **node**, then the read returns the updated values. This applies for both:
- **Error records** accessed through memory-mapped registers, only if the memory-mapped registers are mapped as a Device type that does not permit read speculation.
  - **Error records** accessed through System registers, only if either the exception is a Context synchronization event or a Context synchronization event occurs in program order after taking the exception and before reading the System registers
- R<sub>NHZBG</sub>** When a component reports an error to **node**, the node updates the **error record** registers in finite time, and the update is globally observed for all observers in the system in finite time.
- I<sub>JMVVD</sub>** Direct reads of the System registers, including **error record** System registers, can occur speculatively and out-of-order relative to other instructions executed on the same PE.
- R<sub>WFPWF</sub>** Direct reads and writes of the **error records** through the **ERX\*\_EL1** AArch64 System registers are indirect reads of ERRSELR\_EL1.
- R<sub>FZBLM</sub>** Direct reads and writes of the **error records** through the **ERX\*** AArch32 System registers are indirect reads of ERRSELR.

### 3.2.6 Bridges to other architectures

- R<sub>LWGCX</sub>** A *bridge* is a component that passes **transactions** between two domains.

I <sub>LHGZM</sub>	For example, between an SoC domain and a <i>Peripheral Component Interconnect Express</i> (PCIe) domain.
I <sub>FKKVY</sub>	As described in <a href="#">Error propagation</a> , a high-level transaction might consist of a sequence of operations passed between the domains by the <a href="#">bridge</a> . For the purposes of this manual, the most basic form of a unidirectional transfer between a <i>producer</i> and <i>consumer</i> is considered as a <a href="#">transaction</a> . That is, each one of the sequence of operations is a <a href="#">transaction</a> .
R <sub>ZXBSX</sub>	Other standards might define mechanisms for RAS error recording and handling in particular domains.
I <sub>YQMB</sub>	<p>In the case of PCIe, the PCIe domain might implement one or more of:</p> <ul style="list-style-type: none"><li>• Simple error recording. Errors are recorded in the PCIe device status register.</li><li>• PCIe <i>advanced error reporting</i> (AER). Errors are recorded in the AER logs.</li><li>• Vendor-specific error recording. Errors are recorded in <i>Designated-Vendor-Specific Extended Capability</i> (DVSEC) logs.</li></ul> <p>In each case, errors detected in the PCIe domain are recorded in the PCIe domain and not in the SoC domain.</p>
U <sub>YTXWG</sub>	<p>For the purposes of tracking the origins of a <a href="#">detected error</a> or a <a href="#">deferred error</a> that has propagated between domains, it may be useful to record when a <a href="#">transaction</a> propagates a <a href="#">detected error</a> or a <a href="#">deferred error</a> to a different domains.</p> <p>Arm recommends that a bridge between domains, where the domains implement different error recording mechanisms, uses a <a href="#">node</a> to record when a <a href="#">transaction</a> that is <a href="#">signaled</a> as propagating either a <a href="#">detected error</a> or a <a href="#">deferred error</a> crosses between the domains, recording the source and direction of the transaction in the IMPLEMENTATION DEFINED syndrome for the <a href="#">error record</a>. The direction is either <i>inbound</i> or <i>outbound</i>.</p> <p>See also:</p> <ul style="list-style-type: none"><li>• <a href="#">Multiple error records per node</a></li></ul>

### 3.2.7 Software faults

I <sub>SSQXP</sub>	<p>Examples of <i>software faults</i> include:</p> <ul style="list-style-type: none"><li>• Access to memory or device register that is not present. This includes cases where Secure and Non-secure memory are physically aliased.</li><li>• Access to a device that is not permitted at the device. For example, a Non-secure access to a Secure register.</li><li>• Access to a device that is in an inaccessible state or other illegal access. For example, the device is powered down, or the value written is not supported.</li></ul>
I <sub>BYWQQ</sub>	<p><a href="#">Software fault</a> handling is outside the scope of the RAS System Architecture. Arm makes the following recommendations for accesses that constitute a <a href="#">software fault</a>:</p> <ul style="list-style-type: none"><li>• Where another standard defines a rule or sets a convention for a device, that should be followed. For example:<ul style="list-style-type: none"><li>– For a PCIe device, certain illegal accesses are RAO/WI.</li><li>– [1] requires that <i>reserved accesses</i> to a component, such as reads and writes of unallocated or unimplemented registers and writes to read-only registers, behave as RAZ/WI.</li><li>– [1] requires that under certain conditions accesses to certain debug registers return an error response.</li></ul></li><li>• Accesses to a memory location that is not present can return an <a href="#">in-band error response</a> when all of the following are true:<ul style="list-style-type: none"><li>– The location is <i>not present</i> due to a configuration of the physical address map that is either static or</li></ul></li></ul>



controlled by trusted software. For example, a configuration choice made by the designer, set during initial system configuration, or reconfigured by trusted software.

It is not because a device has been unexpectedly removed or the address map has been otherwise reconfigured. For example, when a user unplugs a device, or using software controls intended to be available to untrusted software. The split between *trusted* and *untrusted* is implementation-specific, but, for example untrusted would typically include unprivileged software and, in systems that supports virtualization, guest operating systems. *Untrusted* might or might not include Non-secure hypervisors.

- Within the aligned page that contains the not-present location, all other locations are also *not present* and have the same behavior. The size of this page is the largest supported translation granule size of all PEs in the system.

That is, there is never any legitimate reason for software to access the page containing the location, and trusted software should set up the translation tables to prevent accesses from occurring.

For all other cases, the access should do one of the following:

- Return zeros for a read and ignore writes. This is the recommended behavior for reads and writes of unallocated or unimplemented registers, reads of write-only registers, and writes of read-only registers.
- Return all-ones for a read and ignore writes.
- Return an IMPLEMENTATION DEFINED value for a read and ignore writes.

If a device does not support a means to record the [software fault](#), it should not return an [in-band error response](#).

INXCDR

A device might implement a RAS System Architecture [node](#) and [error records](#) for recording [software faults](#), for improved debuggability of the fault.

When a device implements a [node](#) and [error records](#) for recording software faults, software faults can be recorded as an error, and reported with an [in-band error response](#) and/or a [fault handling interrupt](#), referred to as a *software fault interrupt*. Arm recommends that this is configurable through [ERR<n>CTLR](#), allowing software to disable the feature. (For example, if an error exception might cause an unrecoverable software state.)

When the feature is disabled, accesses should behave as recommended above.

The following [ERR<n>STATUS.SERR](#) values can be used to record software faults.

SERR	Description
13	Illegal address (software fault). For example, access to unpopulated memory.
14	Illegal access (software fault). For example, byte write to word register.
15	Illegal state (software fault). For example, device not ready.
25	Error recorded by PCIe error logs. Indicates that the node has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

### 3.2.8 Other sources of error and warnings

INWXQS

Other sources of error and warning are possible in a system. Within the RAS System Architecture these are signaled to a PE using an [error recovery interrupt](#) or [fault handling interrupt](#).



### 3.3 Error recovery interrupt

I_JXHYH	If an <b>error recovery interrupt</b> is implemented by a <b>node</b> , then the set of controls for enabling <b>error recovery interrupts</b> is IMPLEMENTATION DEFINED. Software uses <b>ERR&lt;n&gt;FR</b> to determine what controls are implemented.
R_VYFND	For a <b>node</b> <n>, if an <b>error recovery interrupt</b> is implemented, then a control for enabling the <b>error recovery interrupt</b> on <b>Deferred errors</b> , <b>ERR&lt;n&gt;CTLR.DUI</b> , might be implemented.
R_XGBJV	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.DUI</b> control is implemented, then the <b>error recovery interrupt</b> is <i>enabled</i> for <b>Deferred errors</b> when <b>ERR&lt;n&gt;CTLR.DUI</b> is 0b1, and <i>disabled</i> for <b>Deferred errors</b> when <b>ERR&lt;n&gt;CTLR.DUI</b> is 0b0.
R_KRDFZ	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.DUI</b> control is not implemented, then the <b>error recovery interrupt</b> is always <i>disabled</i> for <b>Deferred errors</b> .
R_QSYLK	For a <b>node</b> <n>, if an <b>error recovery interrupt</b> is implemented, then a control for enabling the <b>error recovery interrupt</b> on <b>Uncorrected errors</b> , <b>ERR&lt;n&gt;CTLR.UI</b> , might be implemented.
R_CBVJB	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.UI</b> control is implemented, then the <b>error recovery interrupt</b> is <i>enabled</i> for <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.UI</b> is 0b1, and. <i>disabled</i> for <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.UI</b> is 0b0.
R_ZLXWQ	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.UI</b> control is not implemented, then the <b>error recovery interrupt</b> is always <i>enabled</i> for <b>Uncorrected errors</b> .
R_BLVMZ	For a <b>node</b> <n>, if an <b>error recovery interrupt</b> is not implemented, then the <b>ERR&lt;n&gt;CTLR.{DUI,UI}</b> controls are not implemented.
R_XWHZR	For each implemented control, it is further IMPLEMENTATION DEFINED whether there is a single control or separate controls for reads and writes.
R_LMFJX	The <b>error recovery interrupt</b> is generated when the <b>node</b> records an error, even if the error syndrome is discarded because the <b>error record</b> already records a higher priority error.

## 3.4 Fault handling interrupt

<code>IDZTCG</code>	If a <b>fault handling interrupt</b> is implemented by a <b>node</b> , then the set of controls for enabling <b>fault handling interrupts</b> is IMPLEMENTATION DEFINED. Software uses <b>ERR&lt;n&gt;FR</b> to determine what controls are implemented.
<code>RWHXMB</code>	For a <b>node</b> <n>, if <b>fault handling interrupt</b> is implemented, then the control for generating the <b>fault handling interrupt</b> on <b>corrected error events</b> , <b>ERR&lt;n&gt;CTLR.CFI</b> , might be implemented.
<code>RQWFKB</code>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.CFI</b> control is implemented, then the <b>fault handling interrupt</b> is <i>enabled</i> for <b>corrected error events</b> when <b>ERR&lt;n&gt;CTLR.CFI</b> is 0b1 and <i>disabled</i> for <b>corrected error events</b> when <b>ERR&lt;n&gt;CTLR.CFI</b> is 0b0.
<code>RTSPRZ</code>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.CFI</b> control is implemented, then the <b>ERR&lt;n&gt;CTLR.FI</b> control is implemented, and the <b>fault handling interrupt</b> is <i>enabled</i> for <b>Deferred errors</b> and <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.FI</b> is 0b1 and <i>disabled</i> for <b>Deferred errors</b> and <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.FI</b> is 0b0.
<code>RTSXYM</code>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.CFI</b> control is not implemented, then the control for generating the <b>fault handling interrupt</b> on all recorded errors, <b>ERR&lt;n&gt;CTLR.FI</b> , might be implemented.
<code>RZCKJL</code>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.FI</b> control is implemented and the <b>ERR&lt;n&gt;CTLR.CFI</b> control is not implemented, then the <b>fault handling interrupt</b> is <i>enabled</i> for <b>corrected error events</b> , <b>Deferred errors</b> , and <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.FI</b> is 0b1 and <i>disabled</i> for <b>corrected error events</b> , <b>Deferred errors</b> , and <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.FI</b> is 0b0.
<code>RMLJNK</code>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.FI</b> control is not implemented, then the <b>fault handling interrupt</b> is always <i>enabled</i> for all <b>corrected error events</b> , <b>Deferred errors</b> and <b>Uncorrected errors</b> .
<code>RWFNLG</code>	For a <b>node</b> <n>, if a <b>fault handling interrupt</b> is not implemented, then the <b>ERR&lt;n&gt;CTLR.{CFI,FI}</b> controls are not implemented.

A *Corrected error event* is defined as follows:

<code>RCGZMD</code>	<ul style="list-style-type: none"> <li>If the <b>node</b> implements a <b>corrected error counter</b> then all of the following are true: <ul style="list-style-type: none"> <li>A <b>corrected error event</b> occurs when a counter overflows and sets a counter overflow flag to 0b1.</li> <li>It is UNPREDICTABLE whether a <b>corrected error event</b> occurs when a software write that sets the counter overflow flag to 0b1.</li> <li>It is UNPREDICTABLE whether a <b>corrected error event</b> occurs when a counter overflows and the overflow flag was previously set to 0b1.</li> </ul> </li> </ul>
<code>RXFMTV</code>	<ul style="list-style-type: none"> <li>If the <b>node</b> does not implement <b>Corrected error counters</b> then a <b>corrected error event</b> occurs when the <b>node</b> records an error as <b>Corrected error</b>.</li> </ul>
<code>RYZDHM</code>	For each implemented control, it is further IMPLEMENTATION DEFINED whether there is a single control or separate controls for reads and writes.
<code>RDAQYH</code>	The <b>fault handling interrupt</b> is generated when the <b>node</b> records an error, even if the error syndrome is discarded because the <b>error record</b> already records a higher priority error.

### 3.5 In-band error response signaling (external aborts)

R <sub>QTNMH</sub>	For a <b>node</b> <n>, if support for <b>in-band error response</b> signaling, also referred to as external aborts, is implemented by the <b>node</b> , then the control for enabling <b>in-band error response</b> signaling, <b>ERR&lt;n&gt;CTLR.UE</b> , might be implemented. Software uses <b>ERR&lt;n&gt;FR</b> to determine what controls are implemented.
R <sub>BBFMC</sub>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.UE</b> control is implemented, then <b>in-band error response</b> signaling is <i>enabled</i> for <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.UE</b> is 0b1, and <b>in-band error response</b> signaling is <i>disabled</i> for <b>Uncorrected errors</b> when <b>ERR&lt;n&gt;CTLR.UE</b> is 0b0.
R <sub>XDWP</sub>	For a <b>node</b> <n>, if the <b>ERR&lt;n&gt;CTLR.UE</b> control is not implemented and support for <b>in-band error response</b> signaling is implemented, then <b>in-band error response</b> signaling is always <b>enabled</b> for <b>Uncorrected errors</b> .
R <sub>DMTCY</sub>	For a <b>node</b> <n>, if support for <b>in-band error response</b> signaling is not implemented, then the <b>ERR&lt;n&gt;CTLR.UE</b> control is not implemented.
R <sub>NKMDL</sub>	For the <b>ERR&lt;n&gt;CTLR.UE</b> control, it is further IMPLEMENTATION DEFINED whether there is a single control or separate <b>ERR&lt;n&gt;CTLR.{RUE, WUE}</b> controls for reads and writes.
R <sub>JRYXD</sub>	When the <b>node</b> records an <b>Uncorrected error</b> and signals an <b>in-band error response</b> , it sets <b>ERR&lt;n&gt;STATUS.ER</b> to 0b1.

## 3.6 Critical error interrupt

R <sub>QHJMS</sub>	Support for critical error conditions and <a href="#">critical error interrupts</a> at a <a href="#">node</a> is IMPLEMENTATION DEFINED. Software uses <a href="#">ERR&lt;n&gt;FR</a> to determine what support is implemented.
R <sub>LWHDB</sub>	<a href="#">Critical error interrupts</a> provide a mechanism for a <a href="#">node</a> to report a critical error condition to a system controller for error recovery.
I <sub>WPFSE</sub>	An example of a critical error is one where the <a href="#">node</a> has entered a <a href="#">service failure mode</a> which means that the primary error recovery mechanisms cannot be used. For example, if a memory controller enters a failure mode and stops servicing memory requests from application processors, and application processors host the primary error recovery software, then the error has to be signaled to a secondary error controller that has its own private resources in order to record the error.
R <sub>YQLPR</sub>	For a <a href="#">node</a> <n>, if the <a href="#">critical error interrupt</a> is implemented, then the <a href="#">error recovery interrupt</a> is implemented.
R <sub>LZVMK</sub>	For a <a href="#">node</a> <n>, if the <a href="#">critical error interrupt</a> is implemented, then the <a href="#">critical error interrupt</a> is <i>enabled</i> when <a href="#">ERR&lt;n&gt;CTLR.CI</a> is 0b1 and <i>disabled</i> when <a href="#">ERR&lt;n&gt;CTLR.CI</a> is 0b0.
R <sub>JSVFW</sub>	For a <a href="#">node</a> <n>, if the <a href="#">critical error interrupt</a> is implemented, then when a critical error condition is recorded the <a href="#">node</a> sets <a href="#">ERR&lt;n&gt;STATUS.CI</a> to 0b1, regardless of whether the <a href="#">critical error interrupt</a> is <i>enabled</i> or <i>disabled</i> . <a href="#">ERR&lt;n&gt;STATUS.CI</a> is set to 0b1 in addition to the other syndrome information for the error, which is handled in the normal way.
R <sub>YMGQG</sub>	For a <a href="#">node</a> <n>, if the <a href="#">critical error interrupt</a> is implemented and <i>disabled</i> , then when a critical error condition is detected, the <a href="#">node</a> records the critical error as an <a href="#">Uncontainable error</a> .
I <sub>BNDZW</sub>	Classifying the critical error condition as an <a href="#">Uncontainable error</a> if the <a href="#">critical error interrupt</a> is <i>disabled</i> has the effect of causing the <a href="#">node</a> to generate an <a href="#">error recovery interrupt</a> .
I <sub>VSKSB</sub>	For a <a href="#">node</a> <n>, if the <a href="#">critical error interrupt</a> is implemented and <i>enabled</i> , then it is IMPLEMENTATION DEFINED how the error is classified at the <a href="#">node</a> .

## 3.7 Standard format Corrected error counter

<code>I_FQLPT</code>	The RAS System Architecture defines standard formats for a <a href="#">corrected error counter</a> . Software uses <code>ERR&lt;n&gt;FR</code> to determine whether any standard format <a href="#">corrected error counter</a> is implemented by a <a href="#">node</a> .
<code>R_XYFVB</code>	If a standard format <a href="#">corrected error counter</a> is implemented by a <a href="#">node</a> , then it is IMPLEMENTATION DEFINED whether a single counter or a pair of counters is implemented.
<code>R_SLPQW</code>	For a <a href="#">node</a> <n>, if a standard format <a href="#">corrected error counter</a> is implemented, then the counter or counters are recorded in <code>ERR&lt;n&gt;MISC0</code> .
<code>I_FYBWQ</code>	If a pair of standard format <a href="#">Corrected error counters</a> are implemented by a <a href="#">node</a> , this provides: <ul style="list-style-type: none"> <li>• A first (repeat) error counter to count the first error and any subsequent error detected at the same location.</li> <li>• A second (other) error counter to count errors detected in other locations.</li> </ul>
<code>R_GYPDJ</code>	If a pair of standard format <a href="#">Corrected error counters</a> are implemented by a <a href="#">node</a> , then an <a href="#">error record</a> <n> records a <i>counted-fault location</i> for the error, in one or more of: <ul style="list-style-type: none"> <li>• The <code>ERR&lt;n&gt;ADDR</code> register.</li> <li>• The <code>ERR&lt;n&gt;STATUS.IERR</code> field.</li> <li>• The <code>ERR&lt;n&gt;STATUS.SERR</code> field.</li> <li>• The <code>ERR&lt;n&gt;MISC&lt;m&gt;</code> registers.</li> </ul>

It is IMPLEMENTATION DEFINED which of these or parts thereof describe the [counted-fault location](#).

---

### Note

These registers might contain additional IMPLEMENTATION DEFINED fault location information that is not considered part of the [counted-fault location](#).

---

The [counted-fault location](#) recorded in [error record](#) <n> is either valid or invalid:

<code>R_JCNNX</code>	<ul style="list-style-type: none"> <li>• If the <a href="#">counted-fault location</a> or part of the <a href="#">counted-fault location</a> is held the <code>ERR&lt;n&gt;ADDR</code> register then: <ul style="list-style-type: none"> <li>– This part is valid when <code>ERR&lt;n&gt;STATUS.{V, AV} == {1, 1}</code>.</li> <li>– It is IMPLEMENTATION DEFINED whether this part of the <a href="#">counted-fault location</a> is treated as valid or invalid when <code>ERR&lt;n&gt;STATUS.{V, AV} == {1, 0}</code>.</li> <li>– Otherwise, this part is invalid.</li> </ul> </li> </ul>
<code>R_JMVKQ</code>	<ul style="list-style-type: none"> <li>• If the <a href="#">counted-fault location</a> or part of the <a href="#">counted-fault location</a> is held in the <code>ERR&lt;n&gt;STATUS.IERR</code> field, this part is valid when <code>ERR&lt;n&gt;STATUS.V == 0b1</code> and invalid otherwise.</li> </ul>
<code>R_LTFXM</code>	<ul style="list-style-type: none"> <li>• If the <a href="#">counted-fault location</a> or part of the <a href="#">counted-fault location</a> is held in the <code>ERR&lt;n&gt;STATUS.SERR</code> field, this part is valid when <code>ERR&lt;n&gt;STATUS.V == 0b1</code> and invalid otherwise.</li> </ul>
<code>R_SLYKF</code>	<ul style="list-style-type: none"> <li>• If the <a href="#">counted-fault location</a> or part of the <a href="#">counted-fault location</a> is held in the <code>ERR&lt;n&gt;MISC&lt;m&gt;</code> registers then: <ul style="list-style-type: none"> <li>– This part is valid when <code>ERR&lt;n&gt;STATUS.{V, MV} == {1, 1}</code> and IMPLEMENTATION DEFINED parts of the syndrome data indicate the registers contain a valid <a href="#">counted-fault location</a>.</li> <li>– It is IMPLEMENTATION DEFINED whether this part of the <a href="#">counted-fault location</a> is treated as valid or invalid when <code>ERR&lt;n&gt;STATUS.{V, MV} == {1, 0}</code>.</li> <li>– Otherwise, this part is invalid.</li> </ul> </li> </ul>

R<sub>LSYJ</sub>

- If the [counted-fault location](#) is held across multiple of these registers then the [counted-fault location](#) is valid only if all parts are valid and invalid otherwise.

---

**Note**

- The [counted-fault location](#) is always invalid if [ERR<n>STATUS.V](#) is 0b0, that is, if no error has been recorded by the [error record](#) since [ERR<n>STATUS.V](#) was last cleared to 0b0.
  - The content of IMPLEMENTATION DEFINED syndrome is IMPLEMENTATION DEFINED. This permits, but does not require, for example, the [ERR<n>MISC<m>](#) registers to contain additional valid flags for other parts of the syndrome, or for some parts of [ERR<n>MISC<m>](#) to be valid only for some values of [ERR<n>STATUS.{IERR,SERR}](#).
  - For some implementations, [ERR<n>ADDR](#) is always written when an error is recorded, meaning [ERR<n>STATUS.{V, AV} == {1, 0}](#) is never set by the hardware. Similarly, for some implementations, [ERR<n>STATUS.{V, MV} == {1, 0}](#) is never set by the hardware. For these cases the implementation might ignore the applicable one or ones of the AV and MV bits when determining whether the fault [counted-fault location](#) is valid.
- 

R<sub>JQZZT</sub>

If a pair of standard format [Corrected error counters](#) are implemented by a [node](#), then when a countable error is recorded by [error record <n>](#):

- The first (repeat) error counter counts an error if either of the following are true:
  - The [counted-fault location](#) recorded in error record <n> is invalid.
  - The error being counted is at the same location as the valid [counted-fault location](#) recorded in error record <n>.
- The second (other) counter counts the error otherwise.

I<sub>BYGGW</sub>

When the [counted-fault location](#) recorded in error record <n> is invalid, because this typically means that [ERR<n>STATUS.V](#) is 0b0, the [node](#) typically [overwrites](#) the syndrome, meaning it captures the new [counted-fault location](#). Otherwise, because [ERR<n>STATUS.V](#) is 0b1 the [node keeps](#) the syndrome, meaning the [counted-fault location](#) is unchanged.

R<sub>FYCFY</sub>

If a standard format [corrected error counter](#) is implemented by a [node](#), then if counting an error causes unsigned overflow of the [corrected error counter](#):

- The counter overflow flag is set to 0b1.
- A [corrected error event](#) occurs.

---

**Note**

IMPLEMENTATION DEFINED forms of counters, including other sizes, other overflow models, and other miscellaneous syndrome register locations, might be implemented.

---

See also:

- [Writing the error record](#)
- [Fault handling interrupt](#)

## 3.8 Error recovery, fault handling, and critical error signaling

I <sub>BHBCB</sub>	Error recovery, fault handling, and critical error interrupts are normally routed using an interrupt controller.
R <sub>QTQBJ</sub>	For an Arm <i>Generic Interrupt Controller</i> (GIC), if the error records of the node that generates the interrupts are only accessible via the System registers of one or more PEs, Arm strongly recommends that the interrupt is a <i>Private Peripheral Interrupt</i> (PPI) targeting that PE or one of those PEs.
R <sub>VKLWD</sub>	It is IMPLEMENTATION DEFINED whether each error record has independent interrupt signals for error recovery, fault handling, and critical error interrupts, or whether it shares any of these interrupts with other error records and/or other nodes.
R <sub>WMQZP</sub>	It is IMPLEMENTATION DEFINED whether interrupts are edge-triggered or level-sensitive.
R <sub>SVWPZ</sub>	If the fault handling interrupt is level-sensitive, it is asserted by the node for an error record <n> while any of the following apply: <ul style="list-style-type: none"> <li>Fault handling interrupts on all Deferred errors and Uncorrected errors are enabled, the ERR&lt;n&gt;STATUS.V bit is 0b1, and either or both of the ERR&lt;n&gt;STATUS.{DE,UE} bits are 0b1.</li> <li>Fault handling interrupts on Corrected errors are enabled and either: <ul style="list-style-type: none"> <li>The node implements a corrected error counter, ERR&lt;n&gt;STATUS.V is 0b1, and the counter overflow flag is 0b1.</li> <li>The node does not implement a corrected error counter, ERR&lt;n&gt;STATUS.V is 0b1, and ERR&lt;n&gt;STATUS.CE is nonzero.</li> </ul> </li> </ul>
R <sub>VHSRJ</sub>	If the error recovery interrupt is level-sensitive, it is asserted by the node for an error record <n> while any of the following apply: <ul style="list-style-type: none"> <li>Error recovery interrupts on Uncorrected errors are enabled, ERR&lt;n&gt;STATUS.V is 0b1, and ERR&lt;n&gt;STATUS.UE is 0b1.</li> <li>Error recovery interrupts on Deferred errors are enabled, ERR&lt;n&gt;STATUS.V is 0b1, and ERR&lt;n&gt;STATUS.DE is 0b1.</li> </ul>
R <sub>KTVHF</sub>	If the critical error interrupt is level-sensitive, it is asserted by the node for an error record <n> while critical error interrupts are enabled, ERR<n>STATUS.V is 0b1, and ERR<n>STATUS.CI is 0b1.
R <sub>YPPWB</sub>	If the fault handling interrupt is edge-triggered, it is generated by the node for an error record when any of the following occur: <ul style="list-style-type: none"> <li>Fault handling interrupts on all Deferred errors and Uncorrected errors are enabled, and an error is recorded in the error record as either Deferred error or Uncorrected error.</li> <li>Fault handling interrupts on Corrected errors are enabled and a corrected error event occurs for the error record.</li> </ul>
R <sub>FLWGK</sub>	If the error recovery interrupt is edge-triggered, it is generated by the node for an error record when any of the following occur: <ul style="list-style-type: none"> <li>Error recovery interrupts on Uncorrected errors are enabled, and an error is recorded in the error record as Uncorrected error.</li> <li>Error recovery interrupts on Deferred errors are enabled, and an error is recorded in the error record as Deferred error.</li> </ul>
R <sub>FLPKB</sub>	If the critical error interrupt is edge-triggered, it is generated by the node for an error record <n> when critical error interrupts are enabled, and the node records an error setting ERR<n>STATUS.CI to 0b1.  The critical error interrupt is generated even if the ERR<n>STATUS.CI was already 0b1.

<code>I_MYKYF</code>	An enabled edge-triggered interrupt is generated even if the error syndrome is discarded because the <a href="#">error record</a> already records a higher priority error.
<code>R_XWMLB</code>	It is IMPLEMENTATION DEFINED whether an edge-triggered interrupt is generated by a write to a register that enables an interrupt or otherwise creates the conditions for the interrupt in the other syndrome registers, as defined for a level-sensitive interrupt.
<code>R_GZQWV</code>	The standard <a href="#">error record</a> reserves a set of register locations for programming <i>Message Signaled Interrupts</i> (MSIs). In addition, a recommended layout for these registers is provided.
<code>R_RZDWL</code>	When an error is recorded, or an interrupt becomes enabled, the state of the interrupts is updated in finite time.

See also:

- [ERRIRQCR<n>](#), [ERRERICR0](#), [ERRERICR1](#), [ERRERICR2](#), [ERRCRICR0](#), [ERRCRICR1](#), [ERRCRICR2](#), [ERRFHICR0](#), [ERRFHICR1](#), [ERRFHICR2](#), and [ERRIRQSR](#).
- [Synchronization and error record accesses](#)
- [Error recovery interrupt](#)
- [Fault handling interrupt](#)
- [Critical error interrupt](#)



## 3.9 Error recovery reset

I <sub>FQDBT</sub>	A system comprises multiple power and logical domains, each of which might implement one or more reset signals.
	The RAS System Architecture defines two classes of reset:
R <sub>DKKYC</sub>	<ul style="list-style-type: none"><li>• <i>Cold reset</i> is asserted to a component when it transitions from a powered off state to a powered on state. <a href="#">Cold reset</a> initializes the component to a known initial state. No state is preserved from the previous powered off state.</li></ul>
R <sub>WXRXD</sub>	<ul style="list-style-type: none"><li>• <i>Error Recovery reset</i> is an optional reset that might be applied at any other time. System <a href="#">Error Recovery reset</a> initializes the component to a known state. Unlike <a href="#">Cold reset</a>, any recorded error syndrome information is preserved over a System <a href="#">Error Recovery reset</a>.</li></ul>
R <sub>LPJWX</sub>	The way in which these resets map to other resets is IMPLEMENTATION DEFINED.
R <sub>ZLZDR</sub>	Any mechanisms for asserting resets are IMPLEMENTATION DEFINED.
I <sub>WFGQQ</sub>	For a PE, the <a href="#">Error Recovery reset</a> might be implemented by the architectural Warm reset. If Warm reset is implemented, it preserves the error records in the PE.

## 3.10 Timestamp extension

R <sub>BWYMJ</sub>	The RAS Timestamp Extension is an optional part of RAS System Architecture v1.1.
I <sub>PZVXP</sub>	The RAS Timestamp Extension provides a standard mechanism for timestamping <a href="#">error records</a> .
R <sub>TRHJP</sub>	For a given <a href="#">error record</a> <n>, the timestamp value is recorded in <a href="#">ERR&lt;n&gt;MISC3</a> .
I <sub>FZJMJ</sub>	For a given <a href="#">node</a> <n>, the RAS Timestamp Extension is implemented if <a href="#">ERR&lt;n&gt;FR.TS</a> != 0b00.
R <sub>MHTSQ</sub>	The timestamp uses either the system Generic Timer counter or an IMPLEMENTATION DEFINED timebase.
I <sub>HLDRO</sub>	For a given <a href="#">node</a> <n>, the value of <a href="#">ERR&lt;n&gt;FR.TS</a> defines which timebase is used.
R <sub>XKBJS</sub>	Other than when IMPLEMENTATION DEFINED conditions apply, the following are true: <ul style="list-style-type: none"><li>• The timebase is encoded as a plain binary number.</li><li>• The timebase is monotonically increasing at a fixed rate compared to wallclock time.</li></ul>
I <sub>PQCNK</sub>	The IMPLEMENTATION DEFINED conditions are to allow for the timebase to violate these conditions during initial system configuration.

## 3.11 Common Fault Injection Model Extension

R <sub>CVLDN</sub>	The <i>Common Fault Injection Model Extension</i> is an optional part of RAS System Architecture v1.1.
I <sub>TSWKX</sub>	Other fault injection mechanisms are permitted. For example, if the <i>Common Fault Injection Model Extension</i> is not implemented, the <i>ERRIMPDEF&lt;n&gt;</i> registers might be used for some other IMPLEMENTATION DEFINED fault injection mechanism.
R <sub>YBSBX</sub>	The <i>Common Fault Injection Model Extension</i> can only be implemented for <i>error records</i> accessed through a memory-mapped group of <i>error records</i> if <i>ERRDEVARCH.REVISION</i> $\geq$ 0b0001.
R <sub>PTGZW</sub>	The <i>Common Fault Injection Model Extension</i> fakes the detection of an error at a component.
R <sub>CPYFQ</sub>	A faked error detection results in the <i>node</i> signaling the appropriate ones of the <i>fault handling interrupt</i> , <i>error recovery interrupt</i> , and <i>in-band error response</i> , according to the type of injected error.
I <sub>KHPVH</sub>	The data is not corrupted by the <i>Common Fault Injection Model Extension</i> .
R <sub>RYFQP</sub>	The <i>Common Fault Injection Model Extension</i> supports generating a subset of the <i>component error state</i> types supported by the <i>node</i> .
I <sub>YSQHB</sub>	Arm recommends that the <i>Common Fault Injection Model Extension</i> supports all the <i>component error state</i> types supported by the <i>node</i> .
I <sub>QVLPN</sub>	For a given <i>node</i> <n>, the <i>Common Fault Injection Model Extension</i> is implemented if <i>ERR&lt;n&gt;FR.INJ</i> != 0b00.
I <sub>ZBZHW</sub>	For a given <i>node</i> <n>, the <i>Common Fault Injection Model Extension</i> capabilities are discoverable using <i>ERR&lt;n&gt;PFGF</i> .
I <sub>ZDPWF</sub>	If a <i>node</i> is not capable of recording an <i>component error state</i> type, then it does not support injecting that <i>component error state</i> type.
R <sub>BQCGC</sub>	For a given <i>node</i> <n>, the <i>Common Fault Injection Model Extension</i> is disabled if <i>ERR&lt;n&gt;CTLR.ED</i> is writable and 0b0.
I <sub>YMWNF</sub>	The <i>Common Fault Injection Model Extension</i> registers are: <ul style="list-style-type: none"> <li>• <i>ERR&lt;n&gt;PFGF</i>.</li> <li>• <i>ERR&lt;n&gt;PFGCTL</i>.</li> <li>• <i>ERR&lt;n&gt;PFGCDN</i>.</li> </ul> <p>The <i>Common Fault Injection Model Extension</i> registers are not accessible from AArch32 state. However, when accessed via ERXFR, AArch32 state can access the <i>ERR&lt;n&gt;FR.INJ</i> field described in this section.</p>
I <sub>QFYWD</sub>	Additional constraints might apply if fault injection can affect the operation of Secure state. See <i>Security and Virtualization</i> .

### 3.11.1 Operation of the Common Fault Injection Model Extension

The behaviors in this section apply for a given *node* if the *node* implements the *Common Fault Injection Model Extension*.

R <sub>VDZSG</sub>	When software writes 1 to <i>ERR&lt;n&gt;PFGCTL.CDNEN</i> : <ul style="list-style-type: none"> <li>• If <i>ERR&lt;n&gt;PFGCDN.CDN</i> is nonzero, then the internal Error Generation Counter is set to <i>ERR&lt;n&gt;PFGCDN.CDN</i>.</li> </ul>
--------------------	--

- If `ERR<n>PFGCDN.CDN` is zero, the behavior is UNPREDICTABLE and is one of:
  - The Error Generation Counter is unchanged.
  - The Error Generation Counter is set to zero.
  - The Error Generation Counter is set to zero and the component enters the *fault injection state*.

<code>I<sub>XDWGY</sub></code>	The current value of the Error Generation Counter is not visible to software.
<code>R<sub>PLZMT</sub></code>	While <code>ERR&lt;n&gt;PFGCTL.CDNEN</code> == 1 and the Error Generation Counter is nonzero, the Error Generation Counter decrements by 1 for each cycle at an IMPLEMENTATION DEFINED clock rate.
<code>I<sub>DMNZX</sub></code>	The rate at which the component decrements the counter is defined by the component. For example, it might be the native clock rate for the component, and this might not be the same as the PE clock rate. Software typically discovers this rate from firmware.
<code>R<sub>DDPMH</sub></code>	When the Error Generation Counter decrements to/past zero, the component enters a <i>fault injection state</i> .
<code>R<sub>YXXWT</sub></code>	When the component is in the <i>fault injection state</i> , on the next access to the component, the component: <ul style="list-style-type: none"> <li>• Fakes detection of the <i>component error state</i> type(s) described by <code>ERR&lt;n&gt;PFGCTL</code>.</li> <li>• Reports the injected error to the <i>node</i>, which records the injected error and signals the appropriate ones of the <i>fault handling interrupt</i>, <i>error recovery interrupt</i>, and <i>in-band error response</i>, according to the type of injected error.</li> <li>• Leaves <i>fault injection state</i>.</li> </ul>
<code>R<sub>GJXGL</sub></code>	When an injected error is recorded, the <i>node</i> writes the <code>ERR&lt;n&gt;STATUS</code> .{V, UE, CE, DE, UET} fields according to the <i>component error state</i> type described by <code>ERR&lt;n&gt;PFGCTL</code> .
<code>R<sub>TSXMT</sub></code>	If <code>ERR&lt;n&gt;PFGCTL</code> defines multiple <i>component error state</i> types, or none, the behavior is UNPREDICTABLE and is one of: <ul style="list-style-type: none"> <li>• No error is injected.</li> <li>• An error is injected with an UNPREDICTABLE choice of <i>component error state</i>.</li> </ul>
<code>R<sub>XLJMM</sub></code>	It is IMPLEMENTATION DEFINED how the <i>node</i> updates the <code>ERR&lt;n&gt;STATUS</code> .{AV, ER, OF, MV, PN, CI, IERR, SERR}, <code>ERR&lt;n&gt;ADDR</code> , and <code>ERR&lt;n&gt;MISC&lt;m&gt;</code> when recording an injected error. <code>ERR&lt;n&gt;PFGF</code> describes the IMPLEMENTATION DEFINED options and the controls available in <code>ERR&lt;n&gt;PFGCTL</code> .
<code>I<sub>CSSDM</sub></code>	For many fields, the implementation has the choice to either set the syndrome register or field according to the access that triggers the injected error, or provide finer-grained control over the field, either by a control bit if <code>ERR&lt;n&gt;PFGCTL</code> or by not updating the register or field when the injected error is recorded meaning software can write the injected syndrome to the register or field ahead of injecting the error.
<code>R<sub>WMDWR</sub></code>	For each of the <code>ERR&lt;n&gt;STATUS</code> .{CI, ER, PN} bits, the behavior is UNPREDICTABLE if all of the following are true: <ul style="list-style-type: none"> <li>• <code>ERR&lt;n&gt;PFGF</code> defines that the value injected is controlled by the corresponding <code>ERR&lt;n&gt;PFGCTL</code> bit.</li> <li>• The corresponding <code>ERR&lt;n&gt;PFGCTL</code> bit is 0b1.</li> <li>• For the ER and PN bits, the definition of the <code>ERR&lt;n&gt;STATUS</code> field defines that the bit is not valid for the <i>component error state</i> requested by <code>ERR&lt;n&gt;PFGCTL</code>. For the CI bit, the <i>component error state</i> requested by <code>ERR&lt;n&gt;PFGCTL</code> is not one of an IMPLEMENTATION DEFINED set of permitted values for critical error conditions.</li> </ul> <p>The UNPREDICTABLE behavior is one of:</p> <ul style="list-style-type: none"> <li>• No error is injected.</li> <li>• An error is injected, but the <i>component error state</i> and syndrome bits do not match the requested error type.</li> <li>• The error is injected as requested, including setting the invalid bit or bits to the requested values.</li> </ul>

I <sub>QSLVZ</sub>	<p>This means that:</p> <ul style="list-style-type: none"> <li>• It is IMPLEMENTATION DEFINED which <b>component error states</b> the CI value can be injected with.</li> <li>• The PN value can be injected with a <b>Uncorrected error</b> or <b>Deferred error</b> and cannot be injected with a <b>Corrected error</b>.</li> <li>• The ER value can be injected with an <b>Uncorrected error</b> and cannot be injected with a <b>Corrected error</b>.</li> <li>• It is IMPLEMENTATION DEFINED whether the ER value can be injected with a <b>Deferred error</b>.</li> </ul>
R <sub>GGFSF</sub>	If a single <b>node</b> has multiple <b>error records</b> , then only the first <b>error record</b> has fault injection registers.
R <sub>RBYRG</sub>	If a single <b>node</b> has multiple <b>error records</b> and any of <b>ERR&lt;n&gt;PFGF</b> .{SYN, AV, MV} for the first <b>error record</b> of the <b>node</b> are non-zero, meaning the fault injection mechanism does not update all or some of the <b>ERR&lt;n&gt;MISC&lt;m&gt;</b> or fields when the injected error is recorded, then the injected fault is recorded in the first <b>error record</b> . Otherwise, the injected error might be recorded in any of the multiple <b>error records</b> .
<hr/> <p><b>Note</b></p> <p>If a single <b>node</b> has multiple <b>error records</b> and any of <b>ERR&lt;n&gt;PFGF</b>.{SYN, AV, MV} for the first <b>error record</b> of the <b>node</b> are zero then a <b>node</b> might define which <b>error record</b> is updated or implement an IMPLEMENTATION DEFINED control to allow this to be specified.</p> <hr/>	
I <sub>BDDZZ</sub>	If the <b>node</b> implements <b>fault handling interrupt</b> , <b>error recovery interrupt</b> , and <b>critical error interrupt</b> as edge-triggered interrupts, then recording an injected error has the same behavior as recording a detected error, for generating the edge-triggered interrupt. That is, the interrupt is generated if the interrupt is enabled for the type of error being injected.
I <sub>TVRDH</sub>	If the <b>node</b> implements <b>fault handling interrupt</b> , <b>error recovery interrupt</b> , and <b>critical error interrupt</b> as level-sensitive interrupts, then the level of the interrupt request is a function of the values of the control and status register fields. The behavior of the interrupt request does not depend on whether the control and status registers were written by the <b>node</b> when detecting an error, or written by error injection.
R <sub>CFTGZ</sub>	<p>If the Error Generation Counter is zero and <b>ERR&lt;n&gt;PFGCTL.R</b> == 1 then:</p> <ul style="list-style-type: none"> <li>• If <b>ERR&lt;n&gt;PFGCDN.CDN</b> is nonzero, then the internal Error Generation Counter is set to <b>ERR&lt;n&gt;PFGCDN.CDN</b>.</li> <li>• If <b>ERR&lt;n&gt;PFGCDN.CDN</b> is zero, the behavior is UNPREDICTABLE and is one of: <ul style="list-style-type: none"> <li>– The Error Generation Counter is unchanged.</li> <li>– The Error Generation Counter is set to zero.</li> <li>– The Error Generation Counter is set to zero and the component reenters the <b>fault injection state</b>.</li> </ul> </li> </ul>

## Chapter 4

# RAS Extension and RAS System Architecture Registers

### 4.1 Memory-mapped view

I <sub>MQDMJ</sub>	<i>Error record registers, including memory mapped view</i> defines the registers for memory-mapped <a href="#">error records</a> .
R <sub>HQQNS</sub>	It is IMPLEMENTATION DEFINED which components in the system, if any, implement memory-mapped <a href="#">error records</a> .
R <sub>WWDBV</sub>	A memory-mapped component might implement several <a href="#">error records</a> in a <a href="#">group</a> , relating to one or more <a href="#">nodes</a> .
The <i>Reliability, Availability, Serviceability</i> (RAS) System Architecture defines the following reusable formats for memory-mapped <a href="#">error records</a> :	
R <sub>TPFWF</sub>	<ul style="list-style-type: none"><li>• <i>Memory-mapped error record group view</i> describes a <a href="#">group of error records</a> accessed via a standard 4KB memory-mapped peripheral.</li></ul>
R <sub>DHYDC</sub>	<ul style="list-style-type: none"><li>• <i>Memory-mapped single error record view</i> describes a format for a memory-mapped component that implements a single <a href="#">error record</a>. This might be implemented as part of the control registers for a memory-mapped component. In this format, the first register, <a href="#">ERR&lt;n&gt;FR</a>, is at an address aligned to a multiple of 64 bytes.</li></ul>
R <sub>NBFYF</sub>	In <i>Memory-mapped error record group view</i> , <a href="#">ERRDEVID</a> indicates the highest numbered index of the <a href="#">error records</a> that can be accessed.
I <sub>GFLXS</sub>	For a 4KB peripheral implementing <i>Memory-mapped error record group view</i> , up to 24 <a href="#">error records</a> can be accessed if the <a href="#">Common Fault Injection Model Extension</a> is implemented, and up to 56 otherwise. Groups containing more records can be defined by increasing the page size for a <a href="#">group</a> . This is not described by current versions of the RAS System Architecture. For more information, contact Arm.

R <sub>YGWDK</sub>	In <i>Memory-mapped error record group view</i> , each <b>error record</b> occupies a set of locations at offsets from an <b>error record</b> base. This <b>error record</b> base is a fixed multiple of the index of the <b>error record</b> from the <b>group</b> base.
R <sub>YFCNK</sub>	<i>Memory-mapped error record group view</i> includes a group status register, <b>ERRGSR</b> .
I <sub>TJYGF</sub>	The <b>Common Fault Injection Model Extension</b> is not supported in the <i>Memory-mapped single error record view</i> format.
R <sub>PCXRD</sub>	The <b>error records</b> in a memory-mapped component might be accessible only through that component, or might be shared and accessible through any of: <ul style="list-style-type: none"> <li>• System registers by one or more PEs.</li> <li>• Other memory-mapped components in the same physical address space, including aliases with the same <b>group of error records</b>.</li> <li>• Other memory-mapped components in other address spaces. For example, in both Non-secure and Secure physical address spaces.</li> </ul>
R <sub>JFZRW</sub>	Arm recommends that each memory-mapped <b>error record</b> is accessible at most once in any given physical address space.

#### 4.1.1 Access requirements for memory-mapped views of RAS error records

The requirements for a memory-mapped view of RAS **error records** are:

R <sub>QRLXV</sub>	<ul style="list-style-type: none"> <li>• Reads and writes of unallocated locations are reserved accesses.</li> </ul>
R <sub>PPFBS</sub>	<ul style="list-style-type: none"> <li>• Reads and writes of locations for features that are not implemented are reserved accesses, including: <ul style="list-style-type: none"> <li>– OPTIONAL features that are not implemented.</li> <li>– <b>error records</b> that are not implemented.</li> </ul> </li> </ul>
R <sub>BNKVL</sub>	<ul style="list-style-type: none"> <li>• Reads of WO locations are reserved accesses.</li> </ul>
R <sub>WNFYH</sub>	<ul style="list-style-type: none"> <li>• Writes to RO locations are reserved accesses.</li> </ul>
R <sub>RZWDM</sub>	Reserved accesses are RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.
R <sub>JXHNT</sub>	The memory access sizes that are supported by the memory-mapped component are as described for other memory-mapped components in [1]. It is IMPLEMENTATION DEFINED whether a word-aligned 32-bit access to either half of a doubleword-aligned 64-bit register is supported if there is no <i>Processing element</i> (PE) in the system that supports AArch32.

## 4.2 Reset values

IPQVFQ

When the [node](#) records an error in an [error record](#), depending on the type of error being recorded, it is IMPLEMENTATION DEFINED whether some fields are set to a zero or unchanged.

In most cases, this is because one of the following applies, and it is IMPLEMENTATION DEFINED which:

- The [node](#) sets the field to zero on Cold reset, meaning the value is not required to be changed when the first error is recorded
- The [node](#) sets the field to zero on recording the first error after Cold reset.

To allow for either implementation, software must clear these fields to zero after logging a recorded error and performing a software reset of the [error record](#).

For more information, see [Accessibility](#) in [ERR<n>STATUS](#).



## 4.3 Error record registers, including memory mapped view

INFQOQ

This section describes the [error record](#) registers. The descriptions in this section apply whether the [error record](#) is accessed:

- Through the indirection mechanism described in [Error record System register view](#).
- As memory-mapped registers, as described in [Memory-mapped view](#).

### 4.3.1 Register index

#### Using AArch32 System registers

Table 4.1: Using AArch32 System registers, System register map

Use	To Access	Access	Description
ERXADDR	<a href="#">ERR&lt;n&gt;ADDR[31:0]</a>	R/W	Error Record Address Register
ERXADDR2	<a href="#">ERR&lt;n&gt;ADDR[63:32]</a>	R/W	Error Record Address Register
ERXCTLR	<a href="#">ERR&lt;n&gt;CTLR[31:0]</a>	R/W	Error Record Control Register
ERXCTLR2	<a href="#">ERR&lt;n&gt;CTLR[63:32]</a>	R/W	Error Record Control Register
ERXFR	<a href="#">ERR&lt;n&gt;FR[31:0]</a>	RO	Error Record Feature Register
ERXFR2	<a href="#">ERR&lt;n&gt;FR[63:32]</a>	RO	Error Record Feature Register
ERXMISC0	<a href="#">ERR&lt;n&gt;MISC0[31:0]</a>	R/W	Error Record Miscellaneous Register 0
ERXMISC1	<a href="#">ERR&lt;n&gt;MISC0[63:32]</a>	R/W	Error Record Miscellaneous Register 0
ERXMISC2	<a href="#">ERR&lt;n&gt;MISC1[31:0]</a>	R/W	Error Record Miscellaneous Register 1
ERXMISC3	<a href="#">ERR&lt;n&gt;MISC1[63:32]</a>	R/W	Error Record Miscellaneous Register 1
ERXMISC4	<a href="#">ERR&lt;n&gt;MISC2[31:0]</a>	R/W	Error Record Miscellaneous Register 2
ERXMISC5	<a href="#">ERR&lt;n&gt;MISC2[63:32]</a>	R/W	Error Record Miscellaneous Register 2
ERXMISC6	<a href="#">ERR&lt;n&gt;MISC3[31:0]</a>	R/W	Error Record Miscellaneous Register 3
ERXMISC7	<a href="#">ERR&lt;n&gt;MISC3[63:32]</a>	R/W	Error Record Miscellaneous Register 3
ERXSTATUS	<a href="#">ERR&lt;n&gt;STATUS[31:0]</a>	R/W	Error Record Primary Status Register

#### Using AArch64 System registers

Table 4.2: Using AArch64 System registers, System register map

Use	To Access	Access	Description
ERXADDR_EL1	<a href="#">ERR&lt;n&gt;ADDR</a>	R/W	Error Record Address Register
ERXCTLR_EL1	<a href="#">ERR&lt;n&gt;CTLR</a>	R/W	Error Record Control Register
ERXFR_EL1	<a href="#">ERR&lt;n&gt;FR</a>	RO	Error Record Feature Register
ERXMISC0_EL1	<a href="#">ERR&lt;n&gt;MISC0</a>	R/W	Error Record Miscellaneous Register 0
ERXMISC1_EL1	<a href="#">ERR&lt;n&gt;MISC1</a>	R/W	Error Record Miscellaneous Register 1
ERXMISC2_EL1	<a href="#">ERR&lt;n&gt;MISC2</a>	R/W	Error Record Miscellaneous Register 2
ERXMISC3_EL1	<a href="#">ERR&lt;n&gt;MISC3</a>	R/W	Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	<a href="#">ERR&lt;n&gt;PFGCDN</a>	R/W	Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	<a href="#">ERR&lt;n&gt;PFGCTL</a>	R/W	Pseudo-fault Generation Control Register
ERXPFGF_EL1	<a href="#">ERR&lt;n&gt;PFGF</a>	RO	Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	<a href="#">ERR&lt;n&gt;STATUS</a>	R/W	Error Record Primary Status Register

## Memory-mapped error record group view

Table 4.3: RAS, error record group, memory-mapped register map

Offset	Access	Size	Register	Description
0x000+64xn	RO	64	ERR<n>FR	Error Record Feature Register
0x008+64xn	R/W	64	ERR<n>CTLR	Error Record Control Register
0x010+64xn	R/W	64	ERR<n>STATUS	Error Record Primary Status Register
0x018+64xn	R/W	64	ERR<n>ADDR	Error Record Address Register
0x020+64xn	R/W	64	ERR<n>MISC0	Error Record Miscellaneous Register 0
0x028+64xn	R/W	64	ERR<n>MISC1	Error Record Miscellaneous Register 1
0x030+64xn	R/W	64	ERR<n>MISC2	Error Record Miscellaneous Register 2
0x038+64xn	R/W	64	ERR<n>MISC3	Error Record Miscellaneous Register 3
0x800+64xn	RO	64	ERR<n>PFGF	Pseudo-fault Generation Feature Register
0x800+8xn	R/W	64	ERRIMPDEF<n>	IMPLEMENTATION DEFINED Register <n>
0x808+64xn	R/W	64	ERR<n>PFGCTL	Pseudo-fault Generation Control Register
0x810+64xn	R/W	64	ERR<n>PFGCDN	Pseudo-fault Generation Countdown Register
0xE00	RO	64	ERRGSR	Error Group Status Register
0xE10	RO	32	ERRIIDR	Implementation Identification Register
0xE80	R/W	64	ERRFHICR0	Fault Handling Interrupt Configuration Register 0
0xE80+8xn	R/W	64	ERRIRQCR<n>	Generic Error Interrupt Configuration Register
0xE88	R/W	32	ERRFHICR1	Fault Handling Interrupt Configuration Register 1
0xE8C	R/W	32	ERRFHICR2	Fault Handling Interrupt Configuration Register 2
0xE90	R/W	64	ERRERICR0	Error Recovery Interrupt Configuration Register 0
0xE98	R/W	32	ERRERICR1	Error Recovery Interrupt Configuration Register 1
0xE9C	R/W	32	ERRERICR2	Error Recovery Interrupt Configuration Register 2
0xEA0	R/W	64	ERRCRICR0	Critical Error Interrupt Configuration Register 0
0xEA8	R/W	32	ERRCRICR1	Critical Error Interrupt Configuration Register 1
0xEAC	R/W	32	ERRCRICR2	Critical Error Interrupt Configuration Register 2
0xEF8	R/W	64	ERRIRQSR	Error Interrupt Status Register
0xFA8	RO	64	ERRDEVAFF	Device Affinity Register
0xFBC	RO	32	ERRDEVARCH	Device Architecture Register
0xFC8	RO	32	ERRDEVID	Device Configuration Register
0xFD0	RO	32	ERRPIDR4	Peripheral Identification Register 4
0xFE0	RO	32	ERRPIDR0	Peripheral Identification Register 0
0xFE4	RO	32	ERRPIDR1	Peripheral Identification Register 1
0xFE8	RO	32	ERRPIDR2	Peripheral Identification Register 2
0xFEC	RO	32	ERRPIDR3	Peripheral Identification Register 3
0xFF0	RO	32	ERRCIDR0	Component Identification Register 0
0xFF4	RO	32	ERRCIDR1	Component Identification Register 1
0xFF8	RO	32	ERRCIDR2	Component Identification Register 2
0xFFC	RO	32	ERRCIDR3	Component Identification Register 3

## Memory-mapped single error record view

Table 4.4: RAS, single error record, memory-mapped register map

Offset	Access	Size	Register	Description
0x000	RO	64	ERR<n>FR	Error Record Feature Register
0x008	R/W	64	ERR<n>CTLR	Error Record Control Register
0x010	R/W	64	ERR<n>STATUS	Error Record Primary Syndrome Register
0x018	R/W	64	ERR<n>ADDR	Error Record Address Register

Offset	Access	Size	Register	Description
0x020	R/W	64	<a href="#">ERR&lt;n&gt;MISC0</a>	Error Record Miscellaneous Register 0
0x028	R/W	64	<a href="#">ERR&lt;n&gt;MISC1</a>	Error Record Miscellaneous Register 1
0x030	R/W	64	<a href="#">ERR&lt;n&gt;MISC2</a>	Error Record Miscellaneous Register 2
0x038	R/W	64	<a href="#">ERR&lt;n&gt;MISC3</a>	Error Record Miscellaneous Register 3

### 4.3.2 ERR<n>ADDR, Error Record Address Register

The ERR<n>ADDR characteristics are:

#### Purpose

If an address is associated with a detected error, then it is written to ERR<n>ADDR when the error is recorded. It is IMPLEMENTATION DEFINED how the recorded address maps to the software-visible physical address. Software might have to reconstruct the actual physical addresses using the identity of the node and knowledge of the system.

#### Configurations

ERR<n>ADDR is present only if all of the following are true:

- Error record <n> is implemented.
- Error record <n> includes an address associated with an error.

ERR<n>ADDR is RES0 otherwise.

ERR<q>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then  $q = n$ .

#### Attributes

When accessed using a System register, ERR<n>ADDR is a 64-bit read/write register accessed using:

- MRC and MCR of ERXADDR for ERR<n>ADDR[31:0] when ERRSELR.SEL is set to  $n$ .
- MRC and MCR of ERXADDR2 for ERR<n>ADDR[63:32] when ERRSELR.SEL is set to  $n$ .
- MRS and MSR of ERXADDR\_EL1 when ERRSELR\_EL1.SEL is set to  $n$ .

When accessed as a memory-mapped register, ERR<n>ADDR is a 64-bit read/write register located at offset  $0 \times 018 + 64 \times n$ .

### Field descriptions

The ERR<n>ADDR bit assignments are:

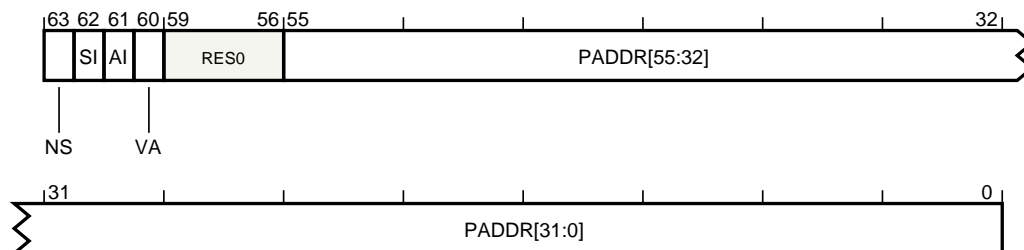


Figure 4.1: ERR<n>ADDR

#### NS, bit [63]

Non-secure attribute. The possible values of this bit are:

0	ERR<n>ADDR.PADDR is a Secure address.
1	ERR<n>ADDR.PADDR is a Non-secure address.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### SI, bit [62]

Secure Incorrect. Indicates whether ERR<n>ADDR.NS is valid. The possible values of this bit are:

0	ERR<n>ADDR.NS is correct. That is, it matches the programmers' view of the Non-secure attribute for this recorded location.
1	ERR<n>ADDR.NS might not be correct, and might not match the programmers' view of the Non-secure attribute for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### AI, bit [61]

Address Incorrect. Indicates whether ERR<n>ADDR.PADDR is a valid physical address that is known to match the programmers' view of the physical address for the recorded location. The possible values of this bit are:

0	ERR<n>ADDR.PADDR is a valid physical address. That is, it matches the programmers' view of the physical address for the recorded location.
1	ERR<n>ADDR.PADDR might not be a valid physical address, and might not match the programmers' view of the physical address for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### VA, bit [60]

Virtual Address. Indicates whether ERR<n>ADDR.PADDR field is a virtual address. The possible values of this bit are:

0	ERR<n>ADDR.PADDR is not a virtual address.
1	ERR<n>ADDR.PADDR is a virtual address.

No context information is provided for the virtual address. When ERR<n>ADDR.VA == 0b1, ERR<n>ADDR.{NS,SI,AI} read as {0,1,1}.

Support for this bit is optional. If this bit is not implemented and ERR<n>ADDR.PADDR field is a virtual address, then ERR<n>ADDR.{NS,SI,AI} read as {0,1,1}.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Bits [59:56]

Reserved. This field is RES0.

#### PADDR, bits [55:0]

Physical Address. Address of the recorded location. If the physical address size implemented by this component is smaller than the size of this field, then high-order bits are unimplemented and either RES0 or have a fixed read-only IMPLEMENTATION DEFINED value. Low-order address bits might also be unimplemented and RES0, for example, if the physical address is always aligned to the size of a protection granule.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

## Accessibility

ERR<n>ADDR ignores writes if all of the following are true:

- Any of the following are true:
  - The RAS Common Fault Injection Model Extension is implemented by the node that owns this error record and `ERR<q>PFGF.AV == 0b0`.
  - The RAS Common Fault Injection Model Extension is not implemented by the node that owns this error record.
- `ERR<n>STATUS.AV == 0b1`.

### 4.3.3 ERR<n>CTLR, Error Record Control Register

The ERR<n>CTLR characteristics are:

#### Purpose

The error control register contains enable bits for the node that writes to this record:

- Enabling error detection and correction.
- Enabling the critical error, error recovery, and fault handling interrupts.
- Enabling in-band error response for Uncorrected errors.

For each bit, if the node does not support the feature, then the bit is RES0. The definition of each record is IMPLEMENTATION DEFINED.

#### Configurations

ERR<n>CTLR is present only if all of the following are true:

- Error record <n> is implemented.
- Error record <n> is the first error record owned by a node.

ERR<n>CTLR is RES0 otherwise.

ERR<n>FR describes the features implemented by the node.

#### Attributes

When accessed using a System register, ERR<n>CTLR is a 64-bit read/write register accessed using:

- MRC and MCR of ERXCTLR for ERR<n>CTLR[31:0] when ERRSELR.SEL is set to *n*.
- MRC and MCR of ERXCTLR2 for ERR<n>CTLR[63:32] when ERRSELR.SEL is set to *n*.
- MRS and MSR of ERXCTLR\_EL1 when ERRSELR\_EL1.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>CTLR is a 64-bit read/write register located at offset  $0 \times 008 + 64 \times n$ .

### Field descriptions

The ERR<n>CTLR bit assignments are:

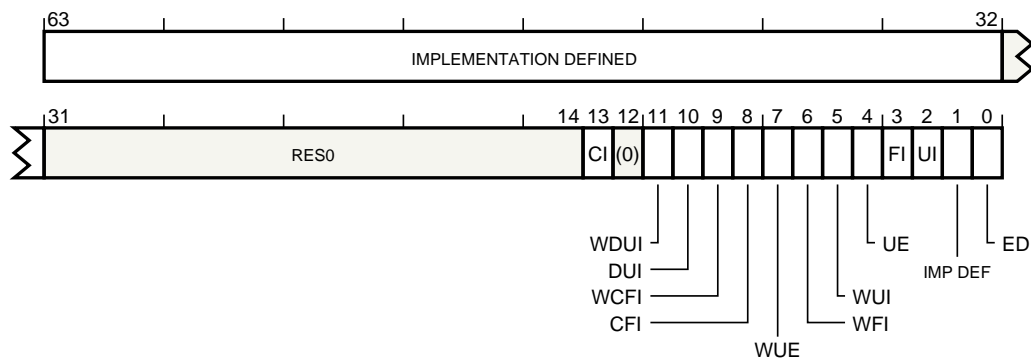


Figure 4.2: ERR<n>CTLR

#### Bits [63:32]

Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### Bits [31:14,12]

Reserved. This field is RES0.

#### CI, bit [13]

Critical error interrupt enable.

##### When $ERR<n>FR.CI == 0b10$

When enabled, the critical error interrupt is generated for a critical error condition. The possible values of this bit are:

0	Critical error interrupt not generated for critical errors. Critical errors are treated as Uncontained errors.
1	Critical error interrupt generated for critical errors.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

##### Otherwise

Reserved. This bit is RES0.

#### WDUI, bit [11]

Error recovery interrupt for deferred errors on writes enable.

##### When $ERR<n>FR.DUI == 0b11$

When enabled, the error recovery interrupt is generated for detected Deferred errors on writes.

The possible values of this bit are:

0	Error recovery interrupt not generated for deferred errors on writes.
1	Error recovery interrupt generated for deferred errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

##### Otherwise

Reserved. This bit is RES0.

#### DUI, bit [10]

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

##### When $ERR<n>FR.DUI == 0b10$

Error recovery interrupt for deferred errors enable.

When  $ERR<n>FR.DUI == 0b10$ , this control applies to errors arising from both reads and writes.

When enabled, the error recovery interrupt is generated for all detected Deferred errors.

The possible values of this bit are:

0	Error recovery interrupt not generated for deferred errors.
1	Error recovery interrupt generated for deferred errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

##### When $ERR<n>FR.DUI == 0b11$

Error recovery interrupt for deferred errors on reads enable.



When **ERR<n>FR.DUI == 0b11**, this bit is named RDUI.

When enabled, the error recovery interrupt is generated for detected Deferred errors on reads.

The possible values of this bit are:

0	Error recovery interrupt not generated for deferred errors on reads.
1	Error recovery interrupt generated for deferred errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

#### WCFI, bit [9]

Fault handling interrupt for Corrected errors on writes enable.

**When ERR<n>FR.CFI == 0b11**

When enabled:

- If the node implements Corrected error counters for writes, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 0b1. For more information, see **ERR<n>MISC0**.
- Otherwise, the fault handling interrupt is also generated for detected Corrected errors on writes.

The possible values of this bit are:

0	Fault handling interrupt not generated for Corrected errors on writes.
1	Fault handling interrupt generated for Corrected errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Otherwise

Reserved. This bit is RES0.

#### CFI, bit [8]

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**When ERR<n>FR.CFI == 0b10**

Fault handling interrupt for Corrected errors enable.

When **ERR<n>FR.CFI == 0b10**, this control applies to errors arising from both reads and writes.

When enabled:

- If the node implements Corrected error counters, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 0b1. For more information, see **ERR<n>MISC0**.
- Otherwise, the fault handling interrupt is also generated for all detected Corrected errors.

The possible values of this bit are:

0	Fault handling interrupt not generated for Corrected errors.
1	Fault handling interrupt generated for Corrected errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

**When  $ERR<n>FR.CFI == 0b11$**

Fault handling interrupt for Corrected errors on reads enable.

When  $ERR<n>FR.CFI == 0b11$ , this bit is named RCFI.

When enabled:

- If the node implements Corrected error counters for reads, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 0b1. For more information, see  $ERR<n>MISC0$ .
- Otherwise, the fault handling interrupt is also generated for detected Corrected errors onreads.

The possible values of this bit are:

0	Fault handling interrupt not generated for Corrected errors on reads.
1	Fault handling interrupt generated for Corrected errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

**WUE, bit [7]**

In-band Uncorrected error reporting on writes enable.

**When  $ERR<n>FR.UE == 0b11$**

When enabled, responses to writes that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

The possible values of this bit are:

0	External Abort response for Uncorrected errors on writes disabled.
1	External Abort response for Uncorrected errors on writes enabled.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**Otherwise**

Reserved. This bit is RES0.

**WFI, bit [6]**

Fault handling interrupt on writes enable.

**When  $ERR<n>FR.FI == 0b11$**

When enabled:

- The fault handling interrupt is generated for detected Deferred errors and Uncorrected errors.
- If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  - If the node implements Corrected error counters for writes, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 0b1.
  - Otherwise, the fault handling interrupt is also generated for detected Corrected errors on writes.

The possible values of this bit are:

0	Fault handling interrupt on writes disabled.
1	Fault handling interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**Otherwise**

Reserved. This bit is RES0.

**WUI, bit [5]**

Uncorrected error recovery interrupt on writes enable.

**When  $ERR<n>FR.UI == 0b11$**

When enabled, the error recovery interrupt is generated for detected Uncorrected errors on writes that are not deferred.

The possible values of this bit are:

0	Error recovery interrupt on writes disabled.
1	Error recovery interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**Otherwise**

Reserved. This bit is RES0.

**UE, bit [4]**

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**When  $ERR<n>FR.UE == 0b10$**

In-band Uncorrected error reporting enable.

When  $ERR<n>FR.UE == 0b10$ , this control applies to errors arising from both reads and writes.

When enabled, responses to transactions that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

The possible values of this bit are:

0	External Abort response for Uncorrected errors disabled.
1	External Abort response for Uncorrected errors enabled.

**When  $ERR<n>FR.UE == 0b11$**

In-band Uncorrected error reporting on reads enable.

When  $ERR<n>FR.UE == 0b11$ , this bit is named RUE.

When enabled, responses to reads that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

The possible values of this bit are:

0	External Abort response for Uncorrected errors on reads disabled.
1	External Abort response for Uncorrected errors on reads enabled.

#### FI, bit [3]

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

##### When $ERR<n>FR.FI == 0b10$

Fault handling interrupt enable.

When  $ERR<n>FR.FI == 0b10$ , this control applies to errors arising from both reads and writes.

When enabled:

- The fault handling interrupt is generated for all detected Deferred errors and Uncorrected errors.
- If the fault handling interrupt for Corrected errors control is not implemented:
  - If the node implements Corrected error counters, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 0b1.
  - Otherwise, the fault handling interrupt is also generated for all detected Corrected errors.

The possible values of this bit are:

0	Fault handling interrupt disabled.
1	Fault handling interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

##### When $ERR<n>FR.FI == 0b11$

Fault handling interrupt on reads enable.

When  $ERR<n>FR.FI == 0b11$ , this bit is named RFI.

When enabled:

- The fault handling interrupt is generated for detected Deferred errors and Uncorrected errors.
- If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  - If the node implements Corrected error counters for reads, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 0b1.
  - Otherwise, the fault handling interrupt is also generated for detected Corrected errors on reads.

The possible values of this bit are:

0	Fault handling interrupt on reads disabled.
1	Fault handling interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

#### UI, bit [2]

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

##### When $ERR<n>FR.UI == 0b10$

Uncorrected error recovery interrupt enable.

When  $ERR<n>FR.UI == 0b10$ , this control applies to errors arising from both reads and writes.

When enabled, the error recovery interrupt is generated for all detected Uncorrected errors that are not deferred.

The possible values of this bit are:

0	Error recovery interrupt disabled.
1	Error recovery interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

**When  $ERR<n>FR.UI == 0b11$**

Uncorrected error recovery interrupt on reads enable.

When  $ERR<n>FR.UI == 0b11$ , this bit is named RUI.

When enabled, the error recovery interrupt is generated for detected Uncorrected errors on reads that are not deferred.

The possible values of this bit are:

0	Error recovery interrupt on reads disabled.
1	Error recovery interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

**Bit [1]**

Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software.

This bit reads as an IMPLEMENTATION DEFINED value and writes to this bit have IMPLEMENTATION DEFINED behavior.

**ED, bit [0]**

Error reporting and logging enable.

**When  $ERR<n>FR.ED == 0b10$**

When disabled, the node behaves as if error detection and correction are disabled, and no errors are recorded or signaled by the node. Arm recommends that, when disabled, correct error detection and correction codes are written for writes, unless disabled by an IMPLEMENTATION DEFINED control for error injection. The possible values of this bit are:

0	Error reporting disabled.
1	Error reporting enabled.

It is IMPLEMENTATION DEFINED whether the node fully disables error detection and correction when reporting is disabled. That is, even with error reporting disabled, the node might continue to silently correct errors. Uncorrectable errors might result in corrupt data being silently propagated by the node.

This bit resets to an IMPLEMENTATION DEFINED value on a Cold reset. This bit is preserved on an Error Recovery reset.

**Note:**

If this node requires initialization after Cold reset to prevent signaling false errors, then Arm recommends this bit is set to 0b0 on Cold reset, meaning errors are not reported from Cold reset. This allows boot software to initialize a node without signaling errors. Software can enable error reporting after the node is initialized. Otherwise, the Cold reset value is IMPLEMENTATION DEFINED. If the Cold reset value is 0b1, the reset values of other controls in this register are also IMPLEMENTATION DEFINED and should not be UNKNOWN.

**Otherwise**

Reserved. This bit is RES0.

**Accessibility**

None.

#### 4.3.4 ERR<n>FR, Error Record Feature Register

The ERR<n>FR characteristics are:

##### Purpose

Defines whether <n> is the first record owned by a node:

- If <n> is the first error record owned by a node, then ERR<n>FR.ED != 0b00.
- If <n> is not the first error record owned by a node, then ERR<n>FR.ED == 0b00.

If <n> is the first record owned by the node, defines which of the common architecturally-defined features are implemented by the node and, of the implemented features, which are software programmable.

##### Configurations

ERR<n>FR is present only if error record <n> is implemented. ERR<n>FR is RES0 otherwise.

##### Attributes

When accessed using a System register, ERR<n>FR is a 64-bit read-only register accessed using:

- MRC of ERXFR for ERR<n>FR[31:0] when ERRSELR.SEL is set to *n*.
- MRC of ERXFR2 for ERR<n>FR[63:32] when ERRSELR.SEL is set to *n*.
- MRS of ERXFR\_EL1 when ERRSELR\_EL1.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>FR is a 64-bit read-only register located at offset  $0 \times 000 + 64 \times n$ .

#### ERR<n>FR (ERR<n>FR.ED != 0b00)

The ERR<n>FR (ERR<n>FR.ED != 0b00) bit assignments are:

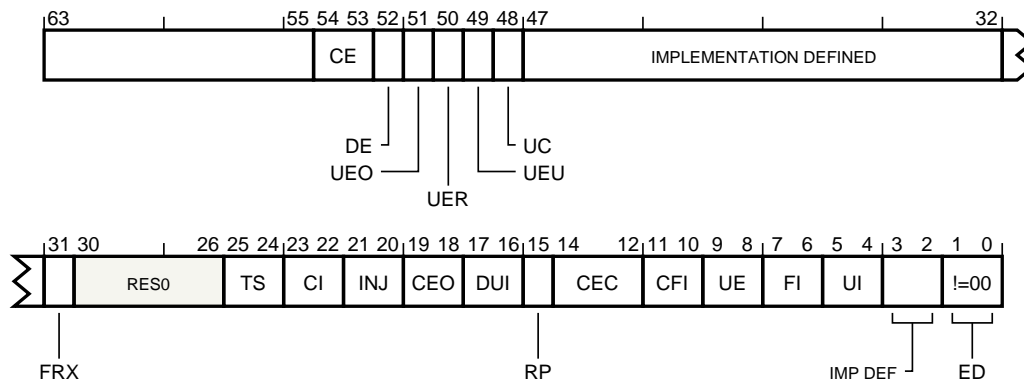


Figure 4.3: ERR<n>FR

##### Bits [63:55]

Reserved.

##### When ERR<n>FR.FRX == 0b1

Reserved. This field is RES0.

##### Otherwise

Reserved for identifying IMPLEMENTATION DEFINED controls. This field reads as an IMPLEMENTATION DEFINED value.

##### CE, bits [54:53]

Corrected Error recording.

**When  $ERR<n>FR.FR_X == 0b1$**

Describes the types of Corrected Error the node can record. The defined values of this field are:



0b00	The node does not record any type of Corrected Error.
0b01	The node can record transient or persistent Corrected Errors (Corrected Errors that are recorded as <code>ERR&lt;n&gt;STATUS.CE == 0b01</code> and <code>0b11</code> ).
0b10	The node can record of a non-specific Corrected Error (a Corrected Error that is recorded as <code>ERR&lt;n&gt;STATUS.CE == 0b10</code> ).
0b11	The node can record any type of Corrected Error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This field reads as an IMPLEMENTATION DEFINED value.

**DE, bit [52]**

Deferred Error recording.

**When `ERR<n>FR.FRX == 0b1`**

Describes whether the node can record this type of error. The defined values of this bit are:

0	The node does not record this type of error.
1	The node can record this type of error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This bit reads as an IMPLEMENTATION DEFINED value.

**UEO, bit [51]**

Latent or Restartable Error recording.

**When `ERR<n>FR.FRX == 0b1`**

Describes whether the node can record this type of error. The defined values of this bit are:

0	The node does not record this type of error.
1	The node can record this type of error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This bit reads as an IMPLEMENTATION DEFINED value.

**UER, bit [50]**

Signaled or Recoverable Error recording.

**When `ERR<n>FR.FRX == 0b1`**

Describes whether the node can record this type of error. The defined values of this bit are:

0	The node does not record this type of error.
1	The node can record this type of error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This bit reads as an IMPLEMENTATION DEFINED value.

**UEU, bit [49]**

Unrecoverable Error recording.

**When  $ERR\langle n \rangle FR.FRX == 0b1$**

Describes whether the node can record this type of error. The defined values of this bit are:

0	The node does not record this type of error.
1	The node can record this type of error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This bit reads as an IMPLEMENTATION DEFINED value.

**UC, bit [48]**

Uncontainable Error recording.

**When  $ERR\langle n \rangle FR.FRX == 0b1$**

Describes whether the node can record this type of error. The defined values of this bit are:

0	The node does not record this type of error.
1	The node can record this type of error.

**Otherwise**

Reserved for identifying IMPLEMENTATION DEFINED controls. This bit reads as an IMPLEMENTATION DEFINED value.

**Bits [47:32]**

Reserved for identifying IMPLEMENTATION DEFINED controls. This field reads as an IMPLEMENTATION DEFINED value.

**FRX, bit [31]**

Feature Register extension.

**When RAS System Architecture v1.1 is implemented**

Defines whether  $ERR\langle n \rangle FR[63:48]$  are architecturally defined. The defined values of this bit are:

0b0	$ERR\langle n \rangle FR[63:48]$ are IMPLEMENTATION DEFINED.
0b1	$ERR\langle n \rangle FR[63:48]$ are defined by the architecture.

**Otherwise**

Reserved. This bit is RES0.

**Bits [30:26]**

Reserved. This field is RES0.

**TS, bits [25:24]**

Timestamp Extension. Indicates whether, for each error record  $\langle m \rangle$  owned by this node,  $ERR\langle m \rangle MISC3$  is used as the timestamp register, and, if it is, the timebase used by the timestamp. The defined values of this field are:

0b00	The node does not support a timestamp register.
0b01	The node implements a timestamp register. The timestamp uses the same timebase as the system Generic Timer.

**Note:**

For an error record which has an affinity to a PE, this is the same timer that is visible through CNTPCT\_ELO at the highest Exception level on that PE.

0b10	The node implements a timestamp register. The timebase for the timestamp is IMPLEMENTATION DEFINED.
------	---

All other values are reserved.

#### CI, bits [23:22]

Critical error interrupt. Indicates whether the critical error interrupt and associated controls are implemented. The defined values of this field are:

0b00	Does not support the critical error interrupt. <a href="#">ERR&lt;n&gt;CTLR.CI</a> is RES0.
0b01	Critical error interrupt is supported and always enabled. <a href="#">ERR&lt;n&gt;CTLR.CI</a> is RES0.
0b10	Critical error interrupt is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.CI</a> .

All other values are reserved.

#### INJ, bits [21:20]

Fault Injection Extension. Indicates whether the RAS Common Fault Injection Model Extension is implemented. The defined values of this field are:

0b00	The node does not support the RAS Common Fault Injection Model Extension.
0b01	The node implements the RAS Common Fault Injection Model Extension. See <a href="#">ERR&lt;n&gt;PFGF</a> for more information.

All other values are reserved.

#### CEO, bits [19:18]

Corrected Error overwrite.

##### When [ERR<n>FR.CEC](#) != 0b000

Indicates the behavior when a second Corrected error is detected after a first Corrected error has been recorded by an error record <m> owned by the node. The defined values of this field are:

0b00	Counts Corrected errors if a counter is implemented. Keeps the previous error syndrome. If the counter overflows, or no counter is implemented, then <a href="#">ERR&lt;m&gt;STATUS.OF</a> is set to 0b1.
0b01	Counts Corrected errors. If <a href="#">ERR&lt;m&gt;STATUS.OF</a> == 0b1 before the Corrected error is counted, then keeps the previous syndrome. Otherwise the previous syndrome is overwritten. If the counter overflows, then <a href="#">ERR&lt;m&gt;STATUS.OF</a> is set to 0b1.

All other values are reserved.

See [Writing the error record](#).

##### Otherwise

Reserved. This field is RES0.

#### DUI, bits [17:16]

Error recovery interrupt for deferred errors control.

##### When [ERR<n>FR.UI](#) != 0b00

Indicates whether the control for enabling error recovery interrupts on deferred errors are implemented. The defined values of this field are:

0b00	Does not support the control for enabling error recovery interrupts on deferred errors. <a href="#">ERR&lt;n&gt;CTLR.DUI</a> is RES0.
0b10	Control for enabling error recovery interrupts on deferred errors is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.DUI</a> .
0b11	Control for enabling error recovery interrupts on deferred errors is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.WDUI</a> for writes and <a href="#">ERR&lt;n&gt;CTLR.RDUI</a> for reads.

All other values are reserved.

#### Otherwise

Reserved. This field is RES0.

#### RP, bit [15]

Repeat counter.

#### When [ERR<n>FR.CEC](#) != 0b000

Indicates whether the node implements the repeat Corrected error counter in [ERR<m>MISC0](#) for each error record <m> owned by the node that implements the standard Corrected error counter. The defined values of this bit are:

0	A single CE counter is implemented.
1	A first (repeat) counter and a second (other) counter are implemented. The repeat counter is the same size as the primary error counter.

#### Otherwise

Reserved. This bit is RES0.

#### CEC, bits [14:12]

Corrected Error Counter. Indicates whether the node implements the standard Corrected error counter (CE counter) mechanisms in [ERR<m>MISC0](#) for each error record <m> owned by the node that can record countable errors. The defined values of this field are:

0b000	Does not implement the standard Corrected error counter model.
0b010	Implements an 8-bit Corrected error counter in <a href="#">ERR&lt;m&gt;MISC0</a> [39:32].
0b100	Implements a 16-bit Corrected error counter in <a href="#">ERR&lt;m&gt;MISC0</a> [47:32].

All other values are reserved.

#### Note:

Implementations might include other error counter models, or might include the standard model and not indicate this in [ERR<n>FR](#).

#### CFI, bits [11:10]

Fault handling interrupt for corrected errors.

#### When [ERR<n>FR.FI](#) != 0b00

Indicates whether the control for enabling fault handling interrupts on corrected errors are implemented. The defined values of this field are:

0b00	Does not support the control for enabling fault handling interrupts on corrected errors. <a href="#">ERR&lt;n&gt;CTLR.CFI</a> is RES0.
0b10	Control for enabling fault handling interrupts on corrected errors is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.CFI</a> .

0b11	Control for enabling fault handling interrupts on corrected errors is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.WCFI</a> for writes and <a href="#">ERR&lt;n&gt;CTLR.RCFI</a> for reads.
------	--

All other values are reserved.

#### Otherwise

Reserved. This field is RES0.

#### UE, bits [9:8]

In-band uncorrected error reporting. Indicates whether the in-band uncorrected error reporting (External Aborts) and associated controls are implemented. The defined values of this field are:

0b00	Does not support the in-band uncorrected error reporting (External Aborts). <a href="#">ERR&lt;n&gt;CTLR.UE</a> is RES0.
0b01	In-band uncorrected error reporting (External Aborts) is supported and always enabled. <a href="#">ERR&lt;n&gt;CTLR.UE</a> is RES0.
0b10	In-band uncorrected error reporting (External Aborts) is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.UE</a> .
0b11	In-band uncorrected error reporting (External Aborts) is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.WUE</a> for writes and <a href="#">ERR&lt;n&gt;CTLR.RUE</a> for reads.

#### FI, bits [7:6]

Fault handling interrupt. Indicates whether the fault handling interrupt and associated controls are implemented. The defined values of this field are:

0b00	Does not support the fault handling interrupt. <a href="#">ERR&lt;n&gt;CTLR.FI</a> is RES0.
0b01	Fault handling interrupt is supported and always enabled. <a href="#">ERR&lt;n&gt;CTLR.FI</a> is RES0.
0b10	Fault handling interrupt is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.FI</a> .
0b11	Fault handling interrupt is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.WFI</a> for writes and <a href="#">ERR&lt;n&gt;CTLR.RFI</a> for reads.

#### UI, bits [5:4]

Error recovery interrupt for uncorrected errors. Indicates whether the error handling interrupt and associated controls are implemented. The defined values of this field are:

0b00	Does not support the error handling interrupt. <a href="#">ERR&lt;n&gt;CTLR.UI</a> is RES0.
0b01	Error handling interrupt is supported and always enabled. <a href="#">ERR&lt;n&gt;CTLR.UI</a> is RES0.
0b10	Error handling interrupt is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.UI</a> .
0b11	Error handling interrupt is supported and controllable using <a href="#">ERR&lt;n&gt;CTLR.WUI</a> for writes and <a href="#">ERR&lt;n&gt;CTLR.RUI</a> for reads.

#### Bits [3:2]

This field reads as an IMPLEMENTATION DEFINED value.

#### ED, bits [1:0]

Error reporting and logging. Indicates whether error record <n> is the first record owned the node, and, if so, whether it implements the controls for enabling and disabling error reporting and logging. The defined values of this field are:

0b01	Error reporting and logging always enabled. <a href="#">ERR&lt;n&gt;CTLR.ED</a> is RES0.
------	--

0b10	Error reporting and logging is controllable using <a href="#">ERR&lt;n&gt;CTLR.ED</a> .
------	---

All other values are reserved.

**ERR<n>FR (ERR<n>FR.ED == 0b00)**

The ERR<n>FR (ERR<n>FR.ED == 0b00) bit assignments are:

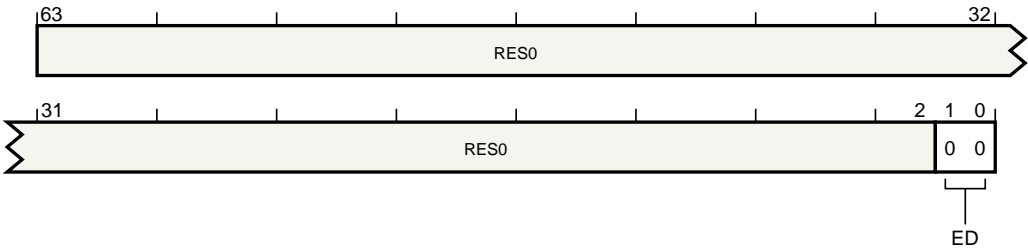


Figure 4.4: ERR<n>FR

**Bits [63:2]**

Reserved. This field is RES0.

**ED, bits [1:0]**

Error reporting and logging. Indicates error record <n> is not the first record owned the node. The defined values of this field are:

0b00	Error record <n> is not the first record owned by the node.
------	---

This field reads as 0b00.

**Accessibility**

None.

### 4.3.5 ERR<n>MISC0, Error Record Miscellaneous Register 0

The ERR<n>MISC0 characteristics are:

#### Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record <n> implements architecturally-defined error counters (ERR<q>FR.CEC != 0b000), and error record <n> can record countable errors, then ERR<n>MISC0 implements the architecturally-defined error counter or counters.

#### Configurations

ERR<n>MISC0 is present only if error record <n> is implemented. ERR<n>MISC0 is RES0 otherwise.

ERR<q>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then  $q = n$ .

For IMPLEMENTATION DEFINED fields in ERR<n>MISC0, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

#### Note:

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

#### Attributes

When accessed using a System register, ERR<n>MISC0 is a 64-bit read/write register accessed using:

- MRC and MCR of ERXMISC0 for ERR<n>MISC0[31:0] when ERRSELR.SEL is set to  $n$ .
- MRS and MSR of ERXMISC0\_EL1 when ERRSELR\_EL1.SEL is set to  $n$ .
- MRC and MCR of ERXMISC1 for ERR<n>MISC0[63:32] when ERRSELR.SEL is set to  $n$ .

When accessed as a memory-mapped register, ERR<n>MISC0 is a 64-bit read/write register located at offset  $0x020 + 64 \times n$ .

### ERR<n>MISC0 (ERR<q>FR.CEC == 0b000)

The ERR<n>MISC0 (ERR<q>FR.CEC == 0b000) bit assignments are:

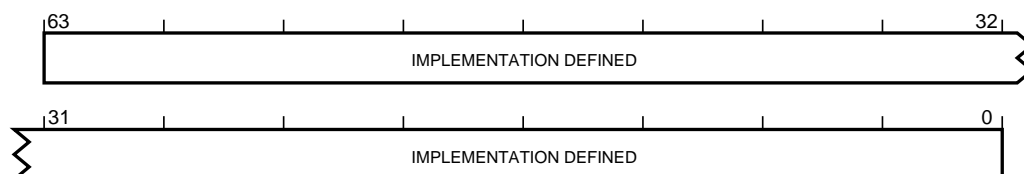


Figure 4.5: ERR<n>MISC0

#### Bits [63:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### ERR<n>MISC0 (ERR<q>FR.CEC == 0b100 && ERR<q>FR.RP == '0')

The ERR<n>MISC0 (ERR<q>FR.CEC == 0b100 && ERR<q>FR.RP == '0') bit assignments are:

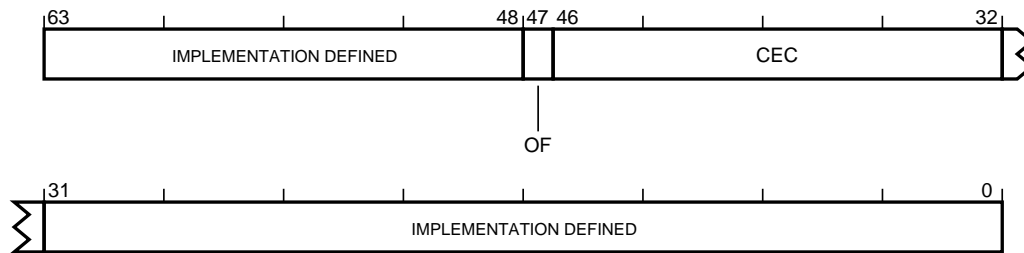


Figure 4.6: ERR<n>MISC0

#### Bits [63:48,31:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### OF, bit [47]

Sticky overflow bit. Set to 1 when ERR<n>MISC0.CEC is incremented and wraps through zero. The possible values of this bit are:

0	Counter has not overflowed.
1	Counter has overflowed.

A direct write that modifies this bit might indirectly set [ERR<n>STATUS.OF](#) to an UNKNOWN value and a direct write to [ERR<n>STATUS.OF](#) that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CEC, bits [46:32]

Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### ERR<n>MISC0 (ERR<q>FR.CEC == 0b010 && ERR<q>FR.RP == '0')

The ERR<n>MISC0 (ERR<q>FR.CEC == 0b010 && ERR<q>FR.RP == '0') bit assignments are:



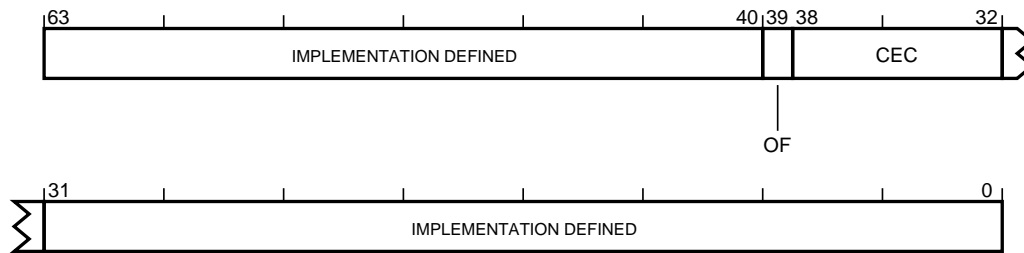


Figure 4.7: ERR<n>MISC0

#### Bits [63:40,31:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### OF, bit [39]

Sticky overflow bit. Set to 1 when ERR<n>MISC0.CEC is incremented and wraps through zero. The possible values of this bit are:

0	Counter has not overflowed.
1	Counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CEC, bits [38:32]

Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### ERR<n>MISC0 (ERR<q>FR.CEC == 0b100 && ERR<q>FR.RP == '1')

The ERR<n>MISC0 (ERR<q>FR.CEC == 0b100 && ERR<q>FR.RP == '1') bit assignments are:

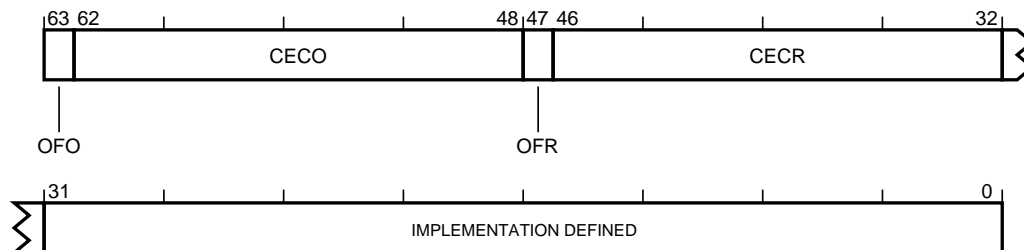


Figure 4.8: ERR<n>MISC0

#### OFO, bit [63]

Sticky overflow bit, other. Set to 1 when ERR<n>MISC0.CECO is incremented and wraps through zero. The possible values of this bit are:

0	Other counter has not overflowed.
---	-----------------------------------

1	Other counter has overflowed.
---	-------------------------------

A direct write that modifies this bit might indirectly set [ERR<n>STATUS.OF](#) to an UNKNOWN value and a direct write to [ERR<n>STATUS.OF](#) that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CECO, bits [62:48]

Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing [ERR<n>MISC0.CECR](#).

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### OFR, bit [47]

Sticky overflow bit, repeat. Set to 1 when [ERR<n>MISC0.CECR](#) is incremented and wraps through zero. The possible values of this bit are:

0	Repeat counter has not overflowed.
1	Repeat counter has overflowed.

A direct write that modifies this bit might indirectly set [ERR<n>STATUS.OF](#) to an UNKNOWN value and a direct write to [ERR<n>STATUS.OF](#) that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CECR, bits [46:32]

Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### Note:

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED [ERR<n>MISC<m>](#) fields on a first Corrected error. [ERR<n>MISC0.CECR](#) is then incremented for each subsequent Corrected Error in the same set and way.

#### Bits [31:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

### **ERR<n>MISC0 (ERR<q>FR.CEC == 0b010 && ERR<q>FR.RP == '1')**

The [ERR<n>MISC0 \(ERR<q>FR.CEC == 0b010 && ERR<q>FR.RP == '1'\)](#) bit assignments are:

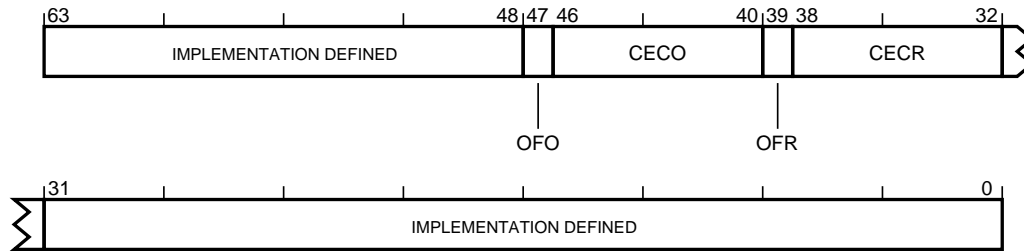


Figure 4.9: ERR<n>MISC0

#### Bits [63:48,31:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### OFO, bit [47]

Sticky overflow bit, other. Set to 1 when ERR<n>MISC0.CECO is incremented and wraps through zero. The possible values of this bit are:

0	Other counter has not overflowed.
1	Other counter has overflowed.

A direct write that modifies this bit might indirectly set [ERR<n>STATUS.OF](#) to an UNKNOWN value and a direct write to [ERR<n>STATUS.OF](#) that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CECO, bits [46:40]

Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing ERR<n>MISC0.CECR.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### OFR, bit [39]

Sticky overflow bit, repeat. Set to 1 when ERR<n>MISC0.CECR is incremented and wraps through zero. The possible values of this bit are:

0	Repeat counter has not overflowed.
1	Repeat counter has overflowed.

A direct write that modifies this bit might indirectly set [ERR<n>STATUS.OF](#) to an UNKNOWN value and a direct write to [ERR<n>STATUS.OF](#) that clears it to zero might indirectly set this bit to an UNKNOWN value.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CECR, bits [38:32]

Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error

Recovery reset.

**Note:**

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED ERR<n>MISC<m> fields on a first Corrected error. ERR<n>MISC0.CECR is then incremented for each subsequent Corrected Error in the same set and way.

## Accessibility

Reads from ERR<n>MISC0 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and ERR<q>PFGF.MV is 0b1, then some parts of this register are read/write when ERR<n>STATUS.MV == 0b1. See ERR<n>PFGF.MV for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When ERR<n>STATUS.MV == 0b1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

**Note:**

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

### 4.3.6 ERR<n>MISC1, Error Record Miscellaneous Register 1

The ERR<n>MISC1 characteristics are:

#### Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

#### Configurations

ERR<n>MISC1 is present only if error record <n> is implemented. ERR<n>MISC1 is RES0 otherwise.

ERR<q>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then  $q = n$ .

For IMPLEMENTATION DEFINED fields in ERR<n>MISC1, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

#### Note:

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

#### Attributes

When accessed using a System register, ERR<n>MISC1 is a 64-bit read/write register accessed using:

- MRS and MSR of ERXMISC1\_EL1 when ERRSELR\_EL1.SEL is set to  $n$ .
- MRC and MCR of ERXMISC2 for ERR<n>MISC1[31:0] when ERRSELR.SEL is set to  $n$ .
- MRC and MCR of ERXMISC3 for ERR<n>MISC1[63:32] when ERRSELR.SEL is set to  $n$ .

When accessed as a memory-mapped register, ERR<n>MISC1 is a 64-bit read/write register located at offset  $0x028 + 64 \times n$ .

### Field descriptions

The ERR<n>MISC1 bit assignments are:

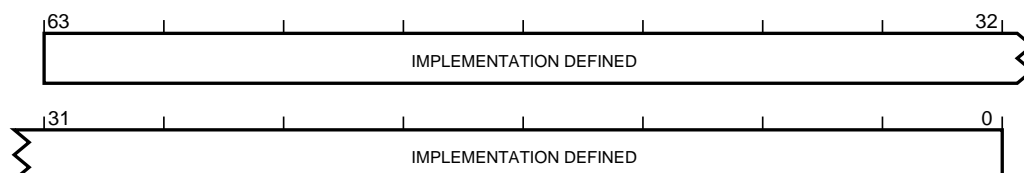


Figure 4.10: ERR<n>MISC1

#### Bits [63:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

## Accessibility

Reads from `ERR<n>MISC1` return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and `ERR<q>PFGF.MV` is `0b1`, then some parts of this register are read/write when `ERR<n>STATUS.MV == 0b1`. See `ERR<n>PFGF.MV` for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When `ERR<n>STATUS.MV == 0b1`, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

### Note:

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

### 4.3.7 ERR<n>MISC2, Error Record Miscellaneous Register 2

The ERR<n>MISC2 characteristics are:

#### Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

#### Configurations

ERR<n>MISC2 is present if error record <n> is implemented. It is IMPLEMENTATION DEFINED whether ERR<n>MISC2 is present if RAS System Architecture v1.1 is not implemented. ERR<n>MISC2 is RES0 if not present.

ERR<q>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then  $q = n$ .

For IMPLEMENTATION DEFINED fields in ERR<n>MISC2, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<n>MISC2 does not require zeroing to return the record to a quiescent state.

#### Note:

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

#### Attributes

When accessed using a System register, ERR<n>MISC2 is a 64-bit read/write register accessed using:

- MRS and MSR of ERXMISC2\_EL1 when ERRSELR\_EL1.SEL is set to  $n$ .
- MRC and MCR of ERXMISC4 for ERR<n>MISC2[31:0] when ERRSELR.SEL is set to  $n$ .
- MRC and MCR of ERXMISC5 for ERR<n>MISC2[63:32] when ERRSELR.SEL is set to  $n$ .

When accessed as a memory-mapped register, ERR<n>MISC2 is a 64-bit read/write register located at offset  $0 \times 030 + 64 \times n$ .

#### Field descriptions

The ERR<n>MISC2 bit assignments are:

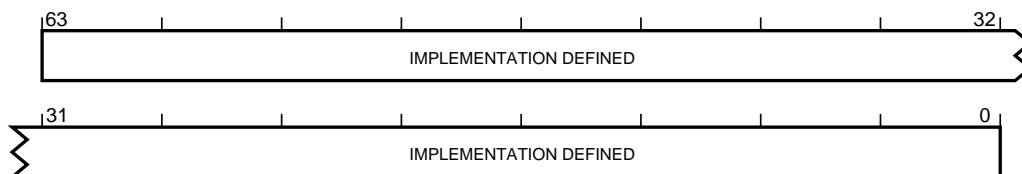


Figure 4.11: ERR<n>MISC2

**Bits [63:0]**

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessibility**

Reads from ERR<n>MISC2 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and ERR<q>PFGF.MV is 0b1, then some parts of this register are read/write when ERR<n>STATUS.MV == 0b1. See ERR<n>PFGF.MV for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When ERR<n>STATUS.MV == 0b1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

**Note:**

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.



### 4.3.8 ERR<n>MISC3, Error Record Miscellaneous Register 3

The ERR<n>MISC3 characteristics are:

#### Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record *n* supports the RAS Timestamp Extension (ERR<q>FR.TS != 0b00), then ERR<n>MISC3 contains the timestamp value for error record *n* when the error was detected. Otherwise the contents of ERR<n>MISC3 are IMPLEMENTATION DEFINED.

#### Configurations

ERR<n>MISC3 is present if error record <*n*> is implemented. It is IMPLEMENTATION DEFINED whether ERR<n>MISC3 is present if RAS System Architecture v1.1 is not implemented. ERR<n>MISC3 is RES0 if not present.

ERR<q>FR describes the features implemented by the node that owns error record <*n*>. <*q*> is the index of the first error record owned by the same node as error record <*n*>. If the node owns a single record, then *q* = *n*.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC3, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<n>MISC3 does not require zeroing to return the record to a quiescent state.

#### Note:

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<q>CTLR.

#### Attributes

When accessed using a System register, ERR<n>MISC3 is a 64-bit read/write register accessed using:

- MRS and MSR of ERXMISC3\_EL1 when ERRSELR\_EL1.SEL is set to *n*.
- MRC and MCR of ERXMISC6 for ERR<n>MISC3[31:0] when ERRSELR.SEL is set to *n*.
- MRC and MCR of ERXMISC7 for ERR<n>MISC3[63:32] when ERRSELR.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>MISC3 is a 64-bit read/write register located at offset 0x038 + 64×*n*.

### ERR<n>MISC3 (ERR<q>FR.TS != 0b00)

The ERR<n>MISC3 (ERR<q>FR.TS != 0b00) bit assignments are:

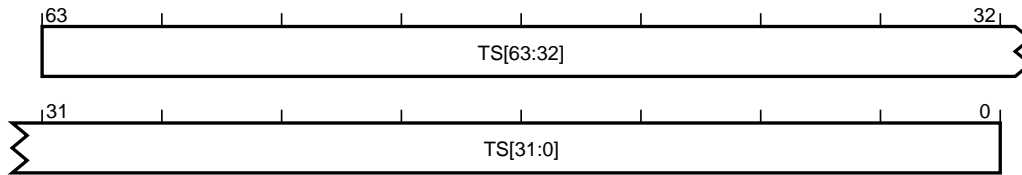


Figure 4.12: ERR<n>MISC3

#### TS, bits [63:0]

Timestamp. Timestamp value recorded when the error was detected. Valid only if [ERR<n>STATUS.V == 0b1](#).

It is IMPLEMENTATION DEFINED whether this field is read-only or read/write.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

See [ERR<q>FR.TS](#).

#### ERR<n>MISC3 (ERR<q>FR.TS == 0b00)

The ERR<n>MISC3 (ERR<q>FR.TS == 0b00) bit assignments are:

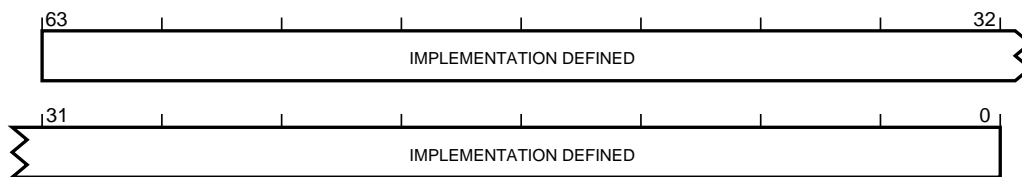


Figure 4.13: ERR<n>MISC3

#### Bits [63:0]

IMPLEMENTATION DEFINED syndrome. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### Accessibility

Reads from ERR<n>MISC3 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and [ERR<q>PFGF.MV](#) is 0b1, then some parts of this register are read/write when [ERR<n>STATUS.MV == 0b1](#). See [ERR<n>PFGF.MV](#) for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When [ERR<n>STATUS.MV == 0b1](#), the miscellaneous syndrome specific to the most recently recorded error ignores writes.

#### Note:

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

### 4.3.9 ERR<n>PFGCDN, Pseudo-fault Generation Countdown Register

The ERR<n>PFGCDN characteristics are:

#### Purpose

Generates one of the errors enabled in the corresponding ERR<n>PFGCTL register.

#### Configurations

ERR<n>PFGCDN is present only if all of the following are true:

- Error record <n> is implemented.
- The node implements the RAS Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00).
- Error record <n> is the first error record owned by a node.

ERR<n>PFGCDN is RES0 otherwise.

ERR<n>FR describes the features implemented by the node.

#### Attributes

When accessed using a System register, ERR<n>PFGCDN is a 64-bit read/write register accessed using MRS and MSR of ERXPFGCDN\_EL1 when ERRSELR\_EL1.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>PFGCDN is a 64-bit read/write register located at offset  $0 \times 810 + 64 \times n$ .

### Field descriptions

The ERR<n>PFGCDN bit assignments are:

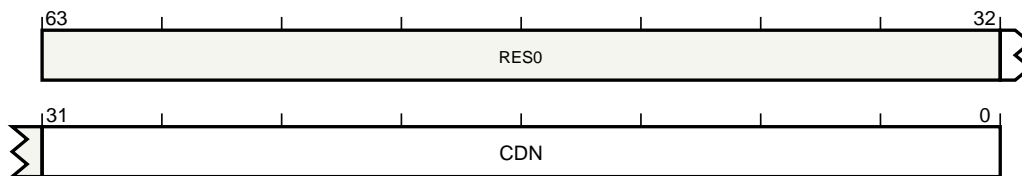


Figure 4.14: ERR<n>PFGCDN

#### Bits [63:32]

Reserved. This field is RES0.

#### CDN, bits [31:0]

Countdown value.

This field is copied to Error Generation Counter when either:

- Software writes ERR<n>PFGCTL.CDNEN with 1.
- The Error Generation Counter decrements to zero and ERR<n>PFGCTL.R == 0b1.

While ERR<n>PFGCTL.CDNEN == 0b1 and the Error Generation Counter is nonzero, the counter decrements by 1 for each cycle at an IMPLEMENTATION DEFINED clock rate. When the counter reaches 0, one of the errors enabled in the ERR<n>PFGCTL register is generated.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### Note:

The current Error Generation Counter value is not visible to software.

## Accessibility

None.

### 4.3.10 ERR<n>PFGCTL, Pseudo-fault Generation Control Register

The ERR<n>PFGCTL characteristics are:

#### Purpose

Enables controlled fault generation.

#### Configurations

ERR<n>PFGCTL is present only if all of the following are true:

- Error record <n> is implemented.
- The node implements the RAS Common Fault Injection Model Extension ([ERR<n>FR.INJ](#) != 0b00).
- Error record <n> is the first error record owned by a node.

ERR<n>PFGCTL is RES0 otherwise.

[ERR<n>FR](#) describes the features implemented by the node.

#### Attributes

When accessed using a System register, ERR<n>PFGCTL is a 64-bit read/write register accessed using MRS and MSR of ERXPFGCTL\_EL1 when ERRSELR\_EL1.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>PFGCTL is a 64-bit read/write register located at offset  $0 \times 808 + 64 \times n$ .

### Field descriptions

The ERR<n>PFGCTL bit assignments are:

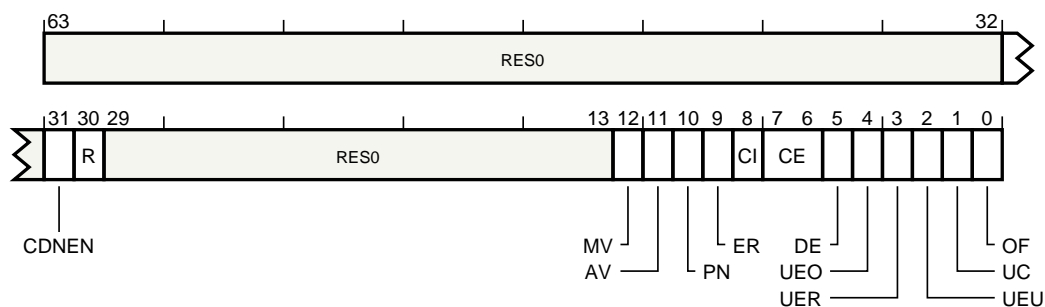


Figure 4.15: ERR<n>PFGCTL

#### Bits [63:32,29:13]

Reserved. This field is RES0.

#### CDNEN, bit [31]

Countdown Enable. Controls transfers from the value that is held in the [ERR<n>PFGCDN](#) into the Error Generation Counter and enables this counter. The possible values of this bit are:

0	The Error Generation Counter is disabled.
1	The Error Generation Counter is enabled. On a write of 0b1 to this bit, the Error Generation Counter is set to <a href="#">ERR&lt;n&gt;PFGCDN.CDN</a> .

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

#### R, bit [30]

Restart. Controls whether, upon reaching zero, the Error Generation Counter restarts from the [ERR<n>PFGCDN](#) value or stops. The possible values of this bit are:

0	On reaching 0, the Error Generation Counter will stop.
1	On reaching 0, the Error Generation Counter is set to <a href="#">ERR&lt;n&gt;PFGCDN.CDN</a> .

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### MV, bit [12]

Miscellaneous syndrome. The value that is written to [ERR<n>STATUS.MV](#) when an injected error is recorded. The possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.MV</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.MV</a> is set to 0b1 when an injected error is recorded.

This bit reads-as-one if the node always records some syndrome in [ERR<n>MISC<m>](#), setting [ERR<n>STATUS.MV](#) to 1, when an injected error is recorded. This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### AV, bit [11]

Address syndrome. The value that is written to [ERR<n>STATUS.AV](#) when an injected error is recorded. The possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.AV</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.AV</a> is set to 0b1 when an injected error is recorded.

This bit reads-as-one if the node always sets [ERR<n>STATUS.AV](#) to 0b1 when an injected error is recorded. This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### PN, bit [10]

Poison flag. The value that is written to [ERR<n>STATUS.PN](#) when an injected error is recorded. The possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.PN</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.PN</a> is set to 0b1 when an injected error is recorded.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### ER, bit [9]

Error Reported flag. The value that is written to [ERR<n>STATUS.ER](#) when an injected error is recorded. The possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.ER</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.ER</a> is set to 0b1 when an injected error is recorded.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CI, bit [8]

Critical Error flag. The value that is written to [ERR<n>STATUS.CI](#) when an injected error is recorded. The possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.CI</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.CI</a> is set to 0b1 when an injected error is recorded.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### CE, bits [7:6]

Corrected Error generation enable. Controls the type of Corrected Error condition that might be generated. The possible values of this field are:

0b00	No error of this type will be generated.
0b01	A non-specific Corrected Error, that is, a Corrected Error that is recorded as <a href="#">ERR&lt;n&gt;STATUS.CE</a> == 0b10, might be generated when the Error Generation Counter decrements to zero.
0b10	A transient Corrected Error, that is, a Corrected Error that is recorded as <a href="#">ERR&lt;n&gt;STATUS.CE</a> == 0b01, might be generated when the Error Generation Counter decrements to zero.
0b11	A persistent Corrected Error, that is, a Corrected Error that is recorded as <a href="#">ERR&lt;n&gt;STATUS.CE</a> == 0b11, might be generated when the Error Generation Counter decrements to zero.

The set of permitted values for this field is defined by [ERR<n>PFGF.CE](#).

This field is RES0 if the node does not support this control.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### DE, bit [5]

Deferred Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed. The possible values of this bit are:

0	No error of this type will be generated.
1	An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UEO, bit [4]

Latent or Restartable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

The possible values of this bit are:

0	No error of this type will be generated.
1	An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UER, bit [3]

Signaled or Recoverable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed. The possible values of this bit are:

0	No error of this type will be generated.
1	An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UEU, bit [2]

Unrecoverable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed. The possible values of this bit are:

0	No error of this type will be generated.
1	An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UC, bit [1]

Uncontainable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed. The possible values of this bit are:

0	No error of this type will be generated.
1	An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### OF, bit [0]

Overflow flag. The value that is written to [ERR<n>STATUS.OF](#) when an injected error is recorded. The



possible values of this bit are:

0	<a href="#">ERR&lt;n&gt;STATUS.OF</a> is set to 0b0 when an injected error is recorded.
1	<a href="#">ERR&lt;n&gt;STATUS.OF</a> is set to 0b1 when an injected error is recorded.

This bit is RES0 if the node does not support this control.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

## Accessibility

None.

### 4.3.11 ERR<n>PFGF, Pseudo-fault Generation Feature Register

The ERR<n>PFGF characteristics are:

#### Purpose

Defines which common architecturally-defined fault generation features are implemented.

#### Configurations

ERR<n>PFGF is present only if all of the following are true:

- Error record <n> is implemented.
- The node implements the RAS Common Fault Injection Model Extension ([ERR<n>FR.INJ](#) != 0b00).
- Error record <n> is the first error record owned by a node.

ERR<n>PFGF is RES0 otherwise.

[ERR<n>FR](#) describes the features implemented by the node.

#### Attributes

When accessed using a System register, ERR<n>PFGF is a 64-bit read-only register accessed using MRS of ERXPFGF\_EL1 when ERRSELR\_EL1.SEL is set to *n*.

When accessed as a memory-mapped register, ERR<n>PFGF is a 64-bit read-only register located at offset  $0 \times 800 + 64 \times n$ .

### Field descriptions

The ERR<n>PFGF bit assignments are:

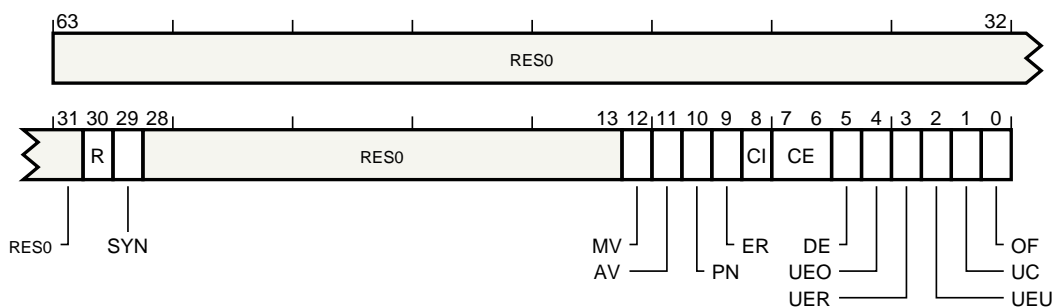


Figure 4.16: ERR<n>PFGF

#### Bits [63:31,28:13]

Reserved. This field is RES0.

#### R, bit [30]

Restartable. Support for Error Generation Counter restart mode. The defined values of this bit are:

0	The node does not support this feature.
1	Feature controllable.

#### SYN, bit [29]

Syndrome. Fault syndrome injection. The defined values of this bit are:

0	When an injected error is recorded, the node sets <a href="#">ERR&lt;n&gt;STATUS</a> .{IERR, SERR} to IMPLEMENTATION DEFINED values. <a href="#">ERR&lt;n&gt;STATUS</a> .{IERR, SERR} are UNKNOWN when <a href="#">ERR&lt;n&gt;STATUS.V</a> == 0b0.
---	---

1	When an injected error is recorded, the node does not update the <a href="#">ERR&lt;n&gt;STATUS.{IERR, SERR}</a> fields. <a href="#">ERR&lt;n&gt;STATUS.{IERR, SERR}</a> are writable when <a href="#">ERR&lt;n&gt;STATUS.V == 0b0</a> .
---	--

**Note:**

If [ERR<n>PFGF.SYN == 0b1](#), software can write specific values into the [ERR<n>STATUS.{IERR, SERR}](#) fields when setting up a fault injection event. The sets of values that can be written to these fields is IMPLEMENTATION DEFINED.

**MV, bit [12]**

Miscellaneous syndrome.

Additional syndrome injection. Defines whether software can control all or part of the syndrome recorded in the [ERR<n>MISC<m>](#) registers when an injected error is recorded.

It is IMPLEMENTATION DEFINED which syndrome fields in [ERR<n>MISC<m>](#) this refers to, as some fields might always be recorded by an error. For example, a Corrected Error counter.

The defined values of this bit are:

0	When an injected error is recorded, the node might record IMPLEMENTATION DEFINED additional syndrome in <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> . If any syndrome is recorded in <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> , then <a href="#">ERR&lt;n&gt;STATUS.MV</a> is set to 0b1.
1	When an injected error is recorded, the node does not update all the syndrome fields in the <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> and does one of: <ul style="list-style-type: none"> <li>The node does not update any fields in <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> and sets <a href="#">ERR&lt;n&gt;STATUS.MV</a> to <a href="#">ERR&lt;n&gt;PFGCTL.MV</a>.</li> <li>The node records some syndrome in <a href="#">ERR&lt;n&gt;MISC&lt;m&gt;</a> and sets <a href="#">ERR&lt;n&gt;STATUS.MV</a> to 0b1. <a href="#">ERR&lt;n&gt;PFGCTL.MV</a> is RAO.</li> </ul> The syndrome fields that the node does not update are unchanged and are writable when <a href="#">ERR&lt;n&gt;STATUS.MV == 0b0</a> .

**Note:**

If [ERR<n>PFGF.MV == 0b1](#), software can write specific values into the [ERR<n>MISC<m>](#) registers when setting up a fault injection event. The values that can be written to these registers are IMPLEMENTATION DEFINED.

**AV, bit [11]**

Address syndrome. Address syndrome injection. The defined values of this bit are:

0	When an injected error is recorded, the node either sets <a href="#">ERR&lt;n&gt;ADDR</a> and <a href="#">ERR&lt;n&gt;STATUS.AV</a> for the access, or leaves these unchanged.
1	When an injected error is recorded, the node does not update <a href="#">ERR&lt;n&gt;ADDR</a> and does one of: <ul style="list-style-type: none"> <li>Sets <a href="#">ERR&lt;n&gt;STATUS.AV</a> to <a href="#">ERR&lt;n&gt;PFGCTL.AV</a>.</li> <li>Sets <a href="#">ERR&lt;n&gt;STATUS.AV</a> to 0b1. <a href="#">ERR&lt;n&gt;PFGCTL.AV</a> is RAO.</li> </ul> <a href="#">ERR&lt;n&gt;ADDR</a> is writable when <a href="#">ERR&lt;n&gt;STATUS.AV == 0b0</a> .

**Note:**

If [ERR<n>PFGF.AV == 0b1](#), software can write a specific value into [ERR<n>ADDR](#) when setting up a fault injection event.

**PN, bit [10]**

Poison flag. Describes how the fault generation feature of the node sets the [ERR<n>STATUS.PN](#) status flag. The defined values of this bit are:

0	When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets <a href="#">ERR&lt;n&gt;STATUS.PN</a> to 0b1.
1	When an injected error is recorded, <a href="#">ERR&lt;n&gt;STATUS.PN</a> is set to <a href="#">ERR&lt;n&gt;PFGCTL.PN</a> .

This behavior replaces the architecture-defined rules for setting the PN bit.

This bit reads-as-zero if the node does not support this flag.

#### ER, bit [9]

Error Reported flag. Describes how the fault generation feature of the node sets the [ERR<n>STATUS.ER](#) status flag. The defined values of this bit are:

0	When an injected error is recorded, the node sets <a href="#">ERR&lt;n&gt;STATUS.ER</a> according to the architecture-defined rules for setting the ER bit.
1	When an injected error is recorded, <a href="#">ERR&lt;n&gt;STATUS.ER</a> is set to <a href="#">ERR&lt;n&gt;PFGCTL.ER</a> . This behavior replaces the architecture-defined rules for setting the ER bit.

This bit reads-as-zero if the node does not support this flag.

#### CI, bit [8]

Critical Error flag. Describes how the fault generation feature of the node sets the [ERR<n>STATUS.CI](#) status flag. The defined values of this bit are:

0	When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets <a href="#">ERR&lt;n&gt;STATUS.CI</a> to 0b1.
1	When an injected error is recorded, <a href="#">ERR&lt;n&gt;STATUS.CI</a> is set to <a href="#">ERR&lt;n&gt;PFGCTL.CI</a> .

This behavior replaces the architecture-defined rules for setting the CI bit.

This bit reads-as-zero if the node does not support this flag.

#### CE, bits [7:6]

Corrected Error generation. Describes the types of Corrected Error that the fault generation feature of the node can generate. The defined values of this field are:

0b00	The fault generation feature of the node cannot generate this type of error.
0b01	The fault generation feature of the node allows generation of a non-specific Corrected Error, that is, a Corrected Error that is recorded as <a href="#">ERR&lt;n&gt;STATUS.CE</a> == 0b10.
0b11	The fault generation feature of the node allows generation of transient or persistent Corrected Errors, that is, Corrected Errors that are recorded as <a href="#">ERR&lt;n&gt;STATUS.CE</a> == 0b01 and 0b11.

All other values are reserved.

If [ERR<n>FR.FRX](#) is 0b1 then [ERR<n>FR.CE](#) indicates whether the node supports this type of error.

This field reads-as-zeros if the node does not support this type of error.

#### DE, bit [5]

Deferred Error generation. Describes whether the fault generation feature of the node can generate this type of error. The defined values of this bit are:

0	The fault generation feature of the node cannot generate this type of error.
1	The fault generation feature of the node allows generation of this type of error.

If **ERR<n>FR.FRX** is 0b1 then **ERR<n>FR.DE** indicates whether the node supports this type of error.

This bit reads-as-zero if the node does not support this type of error.

#### UEO, bit [4]

Latent or Restartable Error generation. Describes whether the fault generation feature of the node can generate this type of error. The defined values of this bit are:

0	The fault generation feature of the node cannot generate this type of error.
1	The fault generation feature of the node allows generation of this type of error.

If **ERR<n>FR.FRX** is 0b1 then **ERR<n>FR.UEO** indicates whether the node supports this type of error.

This bit reads-as-zero if the node does not support this type of error.

#### UER, bit [3]

Signaled or Recoverable Error generation. Describes whether the fault generation feature of the node can generate this type of error. The defined values of this bit are:

0	The fault generation feature of the node cannot generate this type of error.
1	The fault generation feature of the node allows generation of this type of error.

If **ERR<n>FR.FRX** is 0b1 then **ERR<n>FR.UER** indicates whether the node supports this type of error.

This bit reads-as-zero if the node does not support this type of error.

#### UEU, bit [2]

Unrecoverable Error generation. Describes whether the fault generation feature of the node can generate this type of error. The defined values of this bit are:

0	The fault generation feature of the node cannot generate this type of error.
1	The fault generation feature of the node allows generation of this type of error.

If **ERR<n>FR.FRX** is 0b1 then **ERR<n>FR.UEU** indicates whether the node supports this type of error.

This bit reads-as-zero if the node does not support this type of error.

#### UC, bit [1]

Uncontainable Error generation. Describes whether the fault generation feature of the node can generate this type of error. The defined values of this bit are:

0	The fault generation feature of the node cannot generate this type of error.
1	The fault generation feature of the node allows generation of this type of error.

If **ERR<n>FR.FRX** is 0b1 then **ERR<n>FR.UC** indicates whether the node supports this type of error.

This bit reads-as-zero if the node does not support this type of error.

#### OF, bit [0]

Overflow flag. Describes how the fault generation feature of the node sets the **ERR<n>STATUS.OF** status flag. The defined values of this bit are:

---

0	When an injected error is recorded, the node sets <a href="#">ERR&lt;n&gt;STATUS.OF</a> according to the architecture-defined rules for setting the OF bit.
1	When an injected error is recorded, <a href="#">ERR&lt;n&gt;STATUS.OF</a> is set to <a href="#">ERR&lt;n&gt;PFGCTL.OF</a> . This behavior replaces the architecture-defined rules for setting the OF bit.

---

This bit reads-as-zero if the node does not support this flag.

### Accessibility

None.

### 4.3.12 ERR<n>STATUS, Error Record Primary Status Register

The ERR<n>STATUS characteristics are:

#### Purpose

Contains status information for error record <n>, including:

- Whether any error has been detected (valid).
- Whether any detected error was not corrected, and returned to a Requester.
- Whether any detected error was not corrected and deferred.
- Whether an error record has been discarded because additional errors have been detected before the first error was handled by software (overflow).
- Whether any error has been reported.
- Whether the other error record registers contain valid information.
- Whether the error was reported because poison data was detected or because a corrupt value was detected by an error detection code.
- A primary error code.
- An IMPLEMENTATION DEFINED extended error code.

Within this register:

- The {AV, V, MV} bits are valid bits that define whether error record <n> registers are valid.
- The {UE, OF, CE, DE, UET} bits encode the types of error or errors recorded.
- The {CI, ER, PN, IERR, SERR} fields are syndrome fields.

#### Configurations

ERR<n>STATUS is present only if error record <n> is implemented. ERR<n>STATUS is RES0 otherwise.

ERR<q>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then  $q = n$ .

For IMPLEMENTATION DEFINED fields in ERR<n>STATUS, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

#### Note:

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<q>CTLR.

#### Attributes

When accessed using a System register, ERR<n>STATUS is a 64-bit read/write register accessed using:

- MRC and MCR of ERXSTATUS for ERR<n>STATUS[31:0] when ERRSELR.SEL is set to  $n$ .
- MRS and MSR of ERXSTATUS\_EL1 when ERRSELR\_EL1.SEL is set to  $n$ .

When accessed as a memory-mapped register, ERR<n>STATUS is a 64-bit read/write register located at offset  $0x010 + 64 \times n$ .

### ERR<n>STATUS (RAS System Architecture v1.1 is implemented)

The ERR<n>STATUS (RAS System Architecture v1.1 is implemented) bit assignments are:

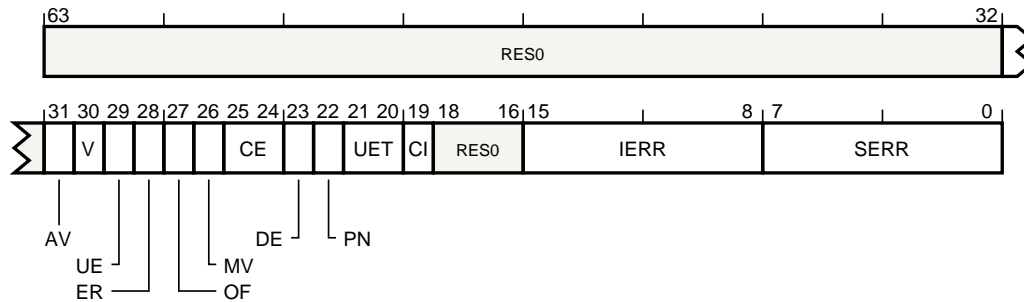


Figure 4.17: ERR<n>STATUS

**Bits [63:32,18:16]**

Reserved. This field is RES0.

**AV, bit [31]**

Address Valid.

**When error record <n> includes an address associated with an error**

The possible values of this bit are:

0	ERR<n>ADDR not valid.
1	ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

**Otherwise**

Reserved. This bit is RES0.

**V, bit [30]**

Status Register Valid. The possible values of this bit are:

0	ERR<n>STATUS not valid.
1	ERR<n>STATUS valid. At least one error has been recorded.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

**UE, bit [29]**

Uncorrected Error. The possible values of this bit are:

0	No errors have been detected, or all detected errors have been either corrected or deferred.
1	At least one detected error was not corrected and not deferred.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0.

This bit is read/write-one-to-clear.



This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### ER, bit [28]

Error Reported. The possible values of this bit are:

0	No in-band error (External Abort) reported.
1	An External Abort was signaled by the component to the Requester making the access or other transaction. This can be because any of the following are true: <ul style="list-style-type: none"> <li>The applicable one of the <a href="#">ERR&lt;q&gt;CTLR</a>.{WUE,RUE,UE} bits is implemented and was set to 0b1 when an Uncorrected error was detected.</li> <li>The applicable one of the <a href="#">ERR&lt;q&gt;CTLR</a>.{WUE,RUE,UE} bits is not implemented and the component always reports errors.</li> </ul>

It is IMPLEMENTATION DEFINED whether this bit can be set to 0b1 by a Deferred error.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == 0b0.
- ERR<n>STATUS.UE == 0b0 and this bit is never set to 0b1 by a Deferred error.
- ERR<n>STATUS.{UE,DE} == {0,0} and this bit can be set to 0b1 by a Deferred error.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Note:

An External Abort signaled by the component might be masked and not generate any exception.

#### OF, bit [27]

Overflow.

Indicates that multiple errors have been detected. This bit is set to 0b1 when one of the following occurs:

- A Corrected error counter is implemented, an error is counted, and the counter overflows.
- ERR<n>STATUS.V was previously set to 0b1, a Corrected error counter is not implemented, and a Corrected error is recorded.
- ERR<n>STATUS.V was previously set to 0b1, and a type of error other than a Corrected error is recorded.

Otherwise, this bit is unchanged when an error is recorded.

If a Corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this bit to an UNKNOWN value.
- A direct write to this bit that clears this bit to zero might indirectly set the counter overflow flag to an UNKNOWN value.

The possible values of this bit are:

0	Since this bit was last cleared to zero, no error syndrome has been discarded and, if a Corrected error counter is implemented, it has not overflowed.
1	Since this bit was last cleared to zero, at least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### MV, bit [26]

Miscellaneous Registers Valid.

#### When error record <n> includes an additional information for an error

The possible values of this bit are:

0	ERR<n>MISC<m> not valid.
1	The IMPLEMENTATION DEFINED contents of the ERR<n>MISC<m> registers contains additional information for an error recorded by this record.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Note:

If the ERR<n>MISC<m> registers can contain additional information for a previously recorded error, then the contents must be self-describing to software or a user. For example, certain fields might relate only to Corrected errors, and other fields only to the most recent error that was not discarded.

#### Otherwise

Reserved. This bit is RES0.

#### CE, bits [25:24]

Corrected Error. The possible values of this field are:

0b00	No errors were corrected.
0b01	At least one transient error was corrected.
0b10	At least one error was corrected.
0b11	At least one persistent error was corrected.

The mechanism by which a component or node detects whether a correctable error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to 0b10 when a corrected error is recorded.

When clearing ERR<n>STATUS.V to 0b0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### DE, bit [23]

Deferred Error. The possible values of this bit are:

0	No errors were deferred.
1	At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### PN, bit [22]

Poison. The possible values of this bit are:

0	Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC), or Corrected error recorded.
1	Uncorrected error or Deferred error recorded because a poison value was detected.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == 0b0.
- ERR<n>STATUS.{DE,UE} == {0,0}.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error. The possible values of this field are:

0b00	Uncorrected error, Uncontainable error (UC).
0b01	Uncorrected error, Unrecoverable error (UEU).
0b10	Uncorrected error, Latent or Restartable error (UEO).
0b11	Uncorrected error, Signaled or Recoverable error (UER).

When clearing ERR<n>STATUS.V to 0b0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == 0b0.
- ERR<n>STATUS.UE == 0b0.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

**Note:**

Software might use the information in the error record registers to determine what recovery is necessary.

**CI, bit [19]**

Critical Error. Indicates whether a critical error condition has been recorded. The possible values of this bit are:

0	No critical error condition.
1	Critical error condition.

When clearing `ERR<n>STATUS.V` to `0b0`, if this bit is nonzero, then Arm recommends that software write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if `ERR<n>STATUS.V == 0b0`.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

**IERR, bits [15:8]**

IMPLEMENTATION DEFINED error code. Used with any primary error code `ERR<n>STATUS.SERR` value. Further IMPLEMENTATION DEFINED information can be placed in the `ERR<n>MISC<m>` registers.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

**Note:**

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

This field is not valid and reads UNKNOWN if all of the following are true:

- Any of the following are true:
  - The RAS Common Fault Injection Model Extension is implemented by the node that owns this error record and `ERR<q>PFGF.SYN == 0b0`.
  - The RAS Common Fault Injection Model Extension is not implemented by the node that owns this error record.
- `ERR<n>STATUS.V == 0b0`.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

**SERR, bits [7:0]**

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry. The possible values of this field are:

0	No error.
1	IMPLEMENTATION DEFINED error.
2	Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.
3	IMPLEMENTATION DEFINED pin. For example, <b>nSEI</b> pin.
4	Assertion failure. For example, consistency failure.
5	Error detected on internal data path. For example, parity on ALU result.
6	Data value from associative memory. For example, ECC error on cache data.
7	Address/control value from associative memory. For example, ECC error on cache tag.

8	Data value from a TLB. For example, ECC error on TLB data.
9	Address/control value from a TLB. For example, ECC error on TLB tag.
10	Data value from producer. For example, parity error on write data bus.
11	Address/control value from producer. For example, parity error on address bus.
12	Data value from (non-associative) external memory. For example, ECC error in SDRAM.
13	Illegal address (software fault). For example, access to unpopulated memory.
14	Illegal access (software fault). For example, byte write to word register.
15	Illegal state (software fault). For example, device not ready.
16	Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.
17	Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.
18	Error response from Completer of access. For example, error response from cache write-back.
19	External timeout. For example, timeout on interaction with another component.
20	Internal timeout. For example, timeout on interface within the component.
21	Deferred error from Completer not supported at Requester. For example, poisoned data received from the Completer of an access by a Requester that cannot defer the error further.
22	Deferred error from Requester not supported at Completer. For example, poisoned data received from the Requester of an access by a Completer that cannot defer the error further.
23	Deferred error from Completer passed through. For example, poisoned data received from the Completer of an access and returned to the Requester.
24	Deferred error from Requester passed through. For example, poisoned data received from the Requester of an access and deferred to the Completer.
25	Error recorded by PCIe error logs. Indicates that the component has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

All other values are reserved.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

**Note:**

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

This field is not valid and reads UNKNOWN if all of the following are true:

- Any of the following are true:
  - The RAS Common Fault Injection Model Extension is implemented by the node that owns this error record and `ERR<q>PFGF.SYN == 0b0`.
  - The RAS Common Fault Injection Model Extension is not implemented by the node that owns this error record.
- `ERR<n>STATUS.V == 0b0`.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

## ERR<n>STATUS (RAS System Architecture v1.0 is implemented)

The ERR<n>STATUS (RAS System Architecture v1.0 is implemented) bit assignments are:

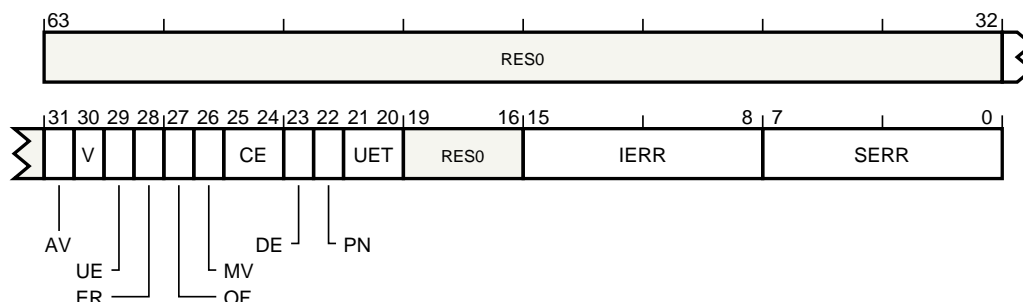


Figure 4.18: ERR<n>STATUS

### Bits [63:32,19:16]

Reserved. This field is RES0.

### AV, bit [31]

Address Valid.

#### When error record <n> includes an address associated with an error

The possible values of this bit are:

0	ERR<n>ADDR not valid.
1	ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Otherwise

Reserved. This bit is RES0.

### V, bit [30]

Status Register Valid. The possible values of this bit are:

0	ERR<n>STATUS not valid.
1	ERR<n>STATUS valid. At least one error has been recorded.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and is not being cleared to 0b0 in the same write.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

### UE, bit [29]

Uncorrected Error. The possible values of this bit are:

0	No errors have been detected, or all detected errors have been either corrected or deferred.
1	At least one detected error was not corrected and not deferred.

When clearing `ERR<n>STATUS.V` to `0b0`, if this bit is nonzero, then Arm recommends that software write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads `UNKNOWN` if `ERR<n>STATUS.V == 0b0`. This bit ignores writes if `ERR<n>STATUS.OF == 0b1` and is not being cleared to `0b0` in the same write.

This bit is read/write-one-to-clear.

This bit resets to an architecturally `UNKNOWN` value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### ER, bit [28]

Error Reported. The possible values of this bit are:

0	No in-band error (External Abort) reported.
1	An External Abort was signaled by the component to the Requester making the access or other transaction. This can be because any of the following are true: <ul style="list-style-type: none"> <li>The applicable one of the <code>ERR&lt;q&gt;CTLR.{WUE,RUE,UE}</code> bits is implemented and was set to <code>0b1</code> when an Uncorrected error was detected.</li> <li>The applicable one of the <code>ERR&lt;q&gt;CTLR.{WUE,RUE,UE}</code> bits is not implemented and the component always reports errors.</li> </ul>

It is IMPLEMENTATION DEFINED whether this bit can be set to `0b1` by a Deferred error.

If this bit is nonzero, then Arm recommends that software write `0b1` to this bit to clear this bit to zero, when any of:

- Clearing `ERR<n>STATUS.V` to `0b0`.
- Clearing `ERR<n>STATUS.UE` to `0b0`, if this bit is never set to `0b1` by a Deferred error.
- Clearing `ERR<n>STATUS.{UE,DE}` to `{0,0}`, if this bit can be set to `0b1` by a Deferred error.

This bit is not valid and reads `UNKNOWN` if any of the following are true:

- `ERR<n>STATUS.V == 0b0`.
- `ERR<n>STATUS.UE == 0b0` and this bit is never set to `0b1` by a Deferred error.
- `ERR<n>STATUS.{UE,DE} == {0,0}` and this bit can be set to `0b1` by a Deferred error.

This bit ignores writes if `ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}`, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

This bit resets to an architecturally `UNKNOWN` value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Note:

An External Abort signaled by the component might be masked and not generate any exception.

#### OF, bit [27]

Overflow.

Indicates that multiple errors have been detected. This bit is set to `0b1` when one of the following occurs:

- An Uncorrected error is detected and `ERR<n>STATUS.UE == 0b1`.

- A Deferred error is detected,  $ERR\langle n \rangle STATUS.UE == 0b0$  and  $ERR\langle n \rangle STATUS.DE == 0b1$ .
- A Corrected error is detected, no Corrected error counter is implemented,  $ERR\langle n \rangle STATUS.UE == 0b0$ ,  $ERR\langle n \rangle STATUS.DE == 0b0$ , and  $ERR\langle n \rangle STATUS.CE \neq 0b00$ .  $ERR\langle n \rangle STATUS.CE$  might be updated for the new Corrected error.
- A Corrected error counter is implemented,  $ERR\langle n \rangle STATUS.UE == 0b0$ ,  $ERR\langle n \rangle STATUS.DE == 0b0$ , and the counter overflows.

It is IMPLEMENTATION DEFINED whether this bit is set to 0b1 when one of the following occurs:

- A Deferred error is detected and  $ERR\langle n \rangle STATUS.UE == 0b1$ .
- A Corrected error is detected, no Corrected error counter is implemented, and either or both the  $ERR\langle n \rangle STATUS.UE$  or  $ERR\langle n \rangle STATUS.DE$  bits are set to 0b1.
- A Corrected error counter is implemented, either or both the  $ERR\langle n \rangle STATUS.UE$  or  $ERR\langle n \rangle STATUS.DE$  bits are set to 0b1, and the counter overflows.

It is IMPLEMENTATION DEFINED whether this bit is cleared to 0b0 when one of the following occurs:

- An Uncorrected error is detected and  $ERR\langle n \rangle STATUS.UE == 0b0$ .
- A Deferred error is detected,  $ERR\langle n \rangle STATUS.UE == 0b0$  and  $ERR\langle n \rangle STATUS.DE == 0b0$ .
- A Corrected error is detected,  $ERR\langle n \rangle STATUS.UE == 0b0$ ,  $ERR\langle n \rangle STATUS.DE == 0b0$  and  $ERR\langle n \rangle STATUS.CE == 0b00$ .

The IMPLEMENTATION DEFINED clearing of this bit might also depend on the value of the other error status bits.

If a Corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this bit to an UNKNOWN value.
- A direct write to this bit that clears this bit to 0b0 might indirectly set the counter overflow flag to an UNKNOWN value.

The possible values of this bit are:

0	<p>If <math>ERR\langle n \rangle STATUS.UE == 0b1</math>, then no error syndrome for an Uncorrected error has been discarded.</p> <p>If <math>ERR\langle n \rangle STATUS.UE == 0b0</math> and <math>ERR\langle n \rangle STATUS.DE == 0b1</math>, then no error syndrome for a Deferred error has been discarded.</p> <p>If <math>ERR\langle n \rangle STATUS.UE == 0b0</math>, <math>ERR\langle n \rangle STATUS.DE == 0b0</math>, and a Corrected error counter is implemented, then the counter has not overflowed.</p> <p>If <math>ERR\langle n \rangle STATUS.UE == 0b0</math>, <math>ERR\langle n \rangle STATUS.DE == 0b0</math>, <math>ERR\langle n \rangle STATUS.CE \neq 0b00</math>, and no Corrected error counter is implemented, then no error syndrome for a Corrected error has been discarded.</p> <p><b>Note:</b></p> <p>This bit might have been set to 0b1 when an error syndrome was discarded and later cleared to 0b0 when a higher priority syndrome was recorded.</p>
1	<p>At least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed.</p>

When clearing  $ERR\langle n \rangle STATUS.V$  to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if  $ERR\langle n \rangle STATUS.V == 0b0$ .

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.



#### MV, bit [26]

Miscellaneous Registers Valid.

#### When error record <n> includes an additional information for an error

The possible values of this bit are:

0	ERR<n>MISC<m> not valid.
1	The IMPLEMENTATION DEFINED contents of the ERR<n>MISC<m> registers contains additional information for an error recorded by this record.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset. This bit is preserved on an Error Recovery reset.

#### Note:

If the ERR<n>MISC<m> registers can contain additional information for a previously recorded error, then the contents must be self-describing to software or a user. For example, certain fields might relate only to Corrected errors, and other fields only to the most recent error that was not discarded.

#### Otherwise

Reserved. This bit is RES0.

#### CE, bits [25:24]

Corrected Error. The possible values of this field are:

0b00	No errors were corrected.
0b01	At least one transient error was corrected.
0b10	At least one error was corrected.
0b11	At least one persistent error was corrected.

The mechanism by which a component or node detects whether a correctable error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to 0b10 when a corrected error is recorded.

When clearing ERR<n>STATUS.V to 0b0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0. This field ignores writes if ERR<n>STATUS.OF == 0b1 and is not being cleared to 0b0 in the same write.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

#### DE, bit [23]

Deferred Error. The possible values of this bit are:

0	No errors were deferred.
1	At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<n>STATUS.V to 0b0, if this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == 0b0. This bit ignores writes if ERR<n>STATUS.OF == 0b1 and is not being cleared to 0b0 in the same write.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### PN, bit [22]

Poison. The possible values of this bit are:

0	Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC), or Corrected error recorded.
1	Uncorrected error or Deferred error recorded because a poison value was detected.

If this bit is nonzero, then Arm recommends that software write 0b1 to this bit to clear this bit to zero, when any of:

- Clearing ERR<n>STATUS.V to 0b0.
- Clearing both ERR<n>STATUS.{DE, UE} to 0b0.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == 0b0.
- ERR<n>STATUS.{DE,UE} == {0,0}.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a Cold reset. This bit is preserved on an Error Recovery reset.

#### UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error. The possible values of this field are:

0b00	Uncorrected error, Uncontainable error (UC).
0b01	Uncorrected error, Unrecoverable error (UEU).
0b10	Uncorrected error, Latent or Restartable error (UEO).
0b11	Uncorrected error, Signaled or Recoverable error (UER).

If this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero, when any of:

- Clearing ERR<n>STATUS.V to 0b0.
- Clearing ERR<n>STATUS.UE to 0b0.

This field is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == 0b0.
- ERR<n>STATUS.UE == 0b0.

This field ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

**Note:**

Software might use the information in the error record registers to determine what recovery is necessary.

**IERR, bits [15:8]**

IMPLEMENTATION DEFINED error code. Used with any primary error code ERR<n>STATUS.SERR value. Further IMPLEMENTATION DEFINED information can be placed in the ERR<n>MISC<m> registers.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

**Note:**

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

This field is not valid and reads UNKNOWN if all of the following are true:

- Any of the following are true:
  - The RAS Common Fault Injection Model Extension is implemented by the node that owns this error record and `ERR<q>PFGF.SYN == 0b0`.
  - The RAS Common Fault Injection Model Extension is not implemented by the node that owns this error record.
- `ERR<n>STATUS.V == 0b0`.

This field ignores writes if `ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}`, and the highest priority of these is not being cleared to zero in the same write.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

**SERR, bits [7:0]**

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry. The possible values of this field are:

0	No error.
1	IMPLEMENTATION DEFINED error.
2	Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.
3	IMPLEMENTATION DEFINED pin. For example, <b>nSEI</b> pin.
4	Assertion failure. For example, consistency failure.
5	Error detected on internal data path. For example, parity on ALU result.
6	Data value from associative memory. For example, ECC error on cache data.
7	Address/control value from associative memory. For example, ECC error on cache tag.
8	Data value from a TLB. For example, ECC error on TLB data.
9	Address/control value from a TLB. For example, ECC error on TLB tag.
10	Data value from producer. For example, parity error on write data bus.
11	Address/control value from producer. For example, parity error on address bus.
12	Data value from (non-associative) external memory. For example, ECC error in SDRAM.
13	Illegal address (software fault). For example, access to unpopulated memory.
14	Illegal access (software fault). For example, byte write to word register.
15	Illegal state (software fault). For example, device not ready.

16	Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.
17	Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.
18	Error response from Completer of access. For example, error response from cache write-back.
19	External timeout. For example, timeout on interaction with another component.
20	Internal timeout. For example, timeout on interface within the component.
21	Deferred error from Completer not supported at Requester. For example, poisoned data received from the Completer of an access by a Requester that cannot defer the error further.
22	Deferred error from Requester not supported at Completer. For example, poisoned data received from the Requester of an access by a Completer that cannot defer the error further.
23	Deferred error from Completer passed through. For example, poisoned data received from the Completer of an access and returned to the Requester.
24	Deferred error from Requester passed through. For example, poisoned data received from the Requester of an access and deferred to the Completer.
25	Error recorded by PCIe error logs. Indicates that the component has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

All other values are reserved.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

**Note:**

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

This field is not valid and reads UNKNOWN if all of the following are true:

- Any of the following are true:
  - The RAS Common Fault Injection Model Extension is implemented by the node that owns this error record and `ERR<q>PFGF.SYN == 0b0`.
  - The RAS Common Fault Injection Model Extension is not implemented by the node that owns this error record.
- `ERR<n>STATUS.V == 0b0`.

This field ignores writes if `ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}`, and the highest priority of these is not being cleared to zero in the same write.

This field resets to an architecturally UNKNOWN value on a Cold reset. This field is preserved on an Error Recovery reset.

## Accessibility

The {AV, V, UE, ER, OF, MV, CE, DE, PN, UET, CI} fields are write-one-to-clear, meaning writes of zero are ignored, and a write of one or all-ones to the field clears the field to zero. The {IERR, SERR} fields are read/write fields, although the set of implemented valid values is IMPLEMENTATION DEFINED. See also [ERR<n>PFGF.SYN](#).

After reading `ERR<n>STATUS`, software must clear the valid bits in the register to allow new errors to be recorded. However, between reading the register and clearing the valid bits, a new error might have overwritten

the register. To prevent this error being lost by software, the register prevents updates to fields that might have been updated by a new error.

When RAS System Architecture v1.0 is implemented:

- Writes to the {UE, DE, CE} fields are ignored if the OF bit is set and is not being cleared.
- Writes to the V bit are ignored if any of the {UE, DE, CE} fields are nonzero and are not being cleared.
- Writes to the {AV, MV} bits and {ER, PN, UET, IERR, SERR} syndrome fields are ignored if the highest priority error status field is nonzero and not being cleared. The error status fields in priority order from highest to lowest, are UE, DE, and CE.

When RAS System Architecture v1.1 is implemented, a write to the register is ignored if all of:

- Any of {V, UE, OF, CE, DE} fields are nonzero before the write.
- The write does not clear the nonzero {V, UE, OF, CE, DE} fields to zero by writing ones to the applicable field or fields.

Some of the fields in ERR<n>STATUS are also defined as UNKNOWN where certain combinations of the {V, DE, UE} status fields are zero. The rules for writes to ERR<n>STATUS allow a node to implement such a field as a fixed read-only value.

For example, when RAS System Architecture v1.1 is implemented, a write to ERR<n>STATUS when ERR<n>STATUS.V is 1 results in either ERR<n>STATUS.V field being cleared to zero, or ERR<n>STATUS.V not changing. Since all fields in ERR<n>STATUS, other than {AV, V, MV}, usually read as UNKNOWN values when ERR<n>STATUS.V is zero, this means those fields can be implemented as read-only if applicable.

To ensure correct and portable operation, when software is clearing the valid bits in the register to allow new errors to be recorded, Arm recommends that software:

- Determine which fields to clear to zero by reading ERR<n>STATUS.
- Write ones to all the write-one-to-clear fields that are nonzero.
- Write zero to all the read/write fields.
- Write zero to all the write-one-to-clear fields that are zero.

Otherwise, these fields might not have the correct value when a new fault is recorded.

An exception is when the node supports writing to these fields as part of fault injection. See also [ERR<n>PFGF.SYN](#).

## Pseudocode operation

---

```
// ERRSTATUS[] (assignment form)
// =====
// For a system register, n = UInt(ERRSELR_EL1.SEL)

ERRSTATUS[integer n] = bits(64) w

// Generate candidate value from the written value and the previous
// (physical register) value
c = w<63:32>:(_ERRSTATUS[n]<31:19> AND NOT(w<31:19>)):Zeros(3):w<15:0>;

if HaveRASSysArchv1p1() then
    // RAS System Architecture v1.1
    // - ignore write if any of V/UE/DE/CE/OF is set
    if !IsZero(c.<V,UE,OF,CE,DE>) then
        c = _ERRSTATUS[n];
else
    // RAS System Architecture v1.0
    // - do not clear UE/DE/CE if OF is set
    if c.OF == '1' then c.<UE,DE,CE> = _ERRSTATUS[n].<UE,DE,CE>;
    // - do not clear V if any of UE/DE/CE is set
```

```

        if !IsZero(c.<UE,DE,CE>) then c.V = _ERRSTATUS[n].V;
        // - do not clear syndrome if not clearing highest priority error
        if (c.UE != '0' ||
            (_ERRSTATUS[n].UE == '0' && c.DE != '0') ||
            (_ERRSTATUS[n].<UE,DE> == '00' && c.CE != '00')) then
            c.<AV,ER,MV,PN,CI,UET,IERR,SERR> = _ERRSTATUS.<AV,ER,MV,PN,CI,UET,IERR,SERR>;

        _ERRSTATUS[n] = c;

    return;

```

---

#### 4.3.13 ERRCIDR0, Component Identification Register 0

The ERRCIDR0 characteristics are:

## Purpose

Provides discovery information about the component.

## Configurations

It is IMPLEMENTATION DEFINED whether ERRCIDR0 is present. ERRCIDR0 is RES0 if not present.

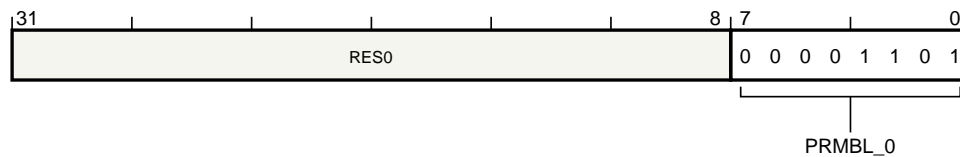
ERRCIDR0 is implemented only as part of a memory-mapped group of error records.

## Attributes

ERRCIDR0 is a 32-bit read-only memory-mapped register located at offset 0xFF0.

## Field descriptions

The ERRCIDR0 bit assignments are:



### Figure 4.19: ERRCIDR0

**Bits [31:8]**

Reserved. This field is RES0.

**PRMBL\_0, bits [7:0]**

Component identification preamble, segment 0. This field reads as 0x0D.

## Accessibility

None.

4.3.14 ERRCIDR1, Component Identification Register 1

The ERRCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

Configurations

It is IMPLEMENTATION DEFINED whether ERRCIDR1 is present. ERRCIDR1 is RES0 if not present.

ERRCIDR1 is implemented only as part of a memory-mapped group of error records.

Attributes

ERRCIDR1 is a 32-bit read-only memory-mapped register located at offset 0xFF4.

Field descriptions

The ERRCIDR1 bit assignments are:

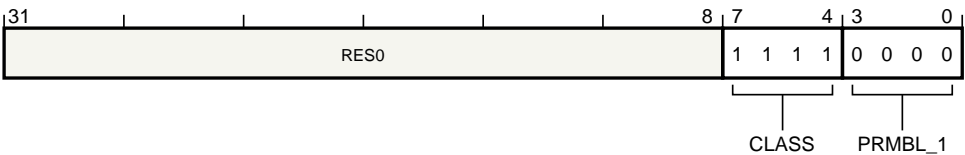


Figure 4.20: ERRCIDR1

Bits [31:8]

Reserved. This field is RES0.

CLASS, bits [7:4]

Component class. The defined values of this field are:

0xF	Generic peripheral with IMPLEMENTATION DEFINED register layout.
-----	---

Other values are defined by the CoreSight Architecture.

This field reads as 0xF.

PRMBL\_1, bits [3:0]

Component identification preamble, segment 1. This field reads as 0x0.

Accessibility

None.



#### 4.3.15 ERRCIDR2, Component Identification Register 2

The ERRCIDR2 characteristics are:

## Purpose

Provides discovery information about the component.

## Configurations

It is IMPLEMENTATION DEFINED whether ERRCIDR2 is present. ERRCIDR2 is RES0 if not present.

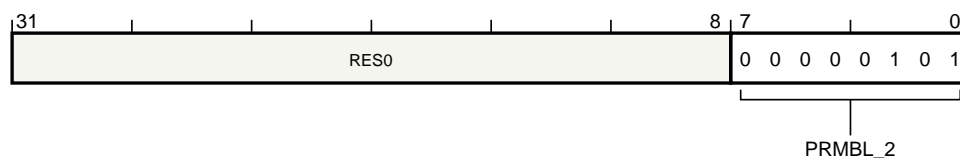
ERRCIDR2 is implemented only as part of a memory-mapped group of error records.

## Attributes

ERRCIDR2 is a 32-bit read-only memory-mapped register located at offset 0xFF8.

## Field descriptions

The ERRCIDR2 bit assignments are:



**Figure 4.21: ERRCIDR2**

**Bits [31:8]**

Reserved. This field is RES0.

**PRMBL\_2, bits [7:0]**

Component identification preamble, segment 2. This field reads as 0x05.

## Accessibility

None.

### 4.3.16 ERRCIDR3, Component Identification Register 3

The ERRCIDR3 characteristics are:

#### Purpose

Provides discovery information about the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRCIDR3 is present. ERRCIDR3 is RES0 if not present.

ERRCIDR3 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRCIDR3 is a 32-bit read-only memory-mapped register located at offset 0xFFC.

### Field descriptions

The ERRCIDR3 bit assignments are:

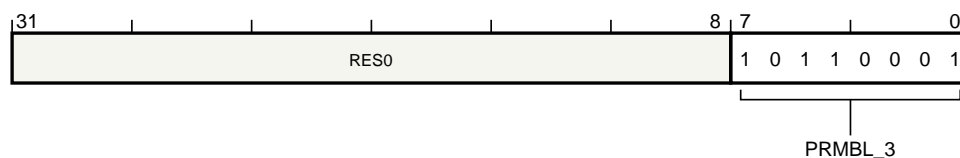


Figure 4.22: ERRCIDR3

#### Bits [31:8]

Reserved. This field is RES0.

#### PRMBL\_3, bits [7:0]

Component identification preamble, segment 3. This field reads as 0xB1.

### Accessibility

None.

### 4.3.17 ERRCRICR0, Critical Error Interrupt Configuration Register 0

The ERRCRICR0 characteristics are:

#### Purpose

Critical Error Interrupt configuration register.

#### Configurations

ERRCRICR0 is present only if all of the following are true:

- Any of the following are true:
  - The Critical Error Interrupt is implemented.
  - The implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.
- Interrupt configuration registers are implemented.

ERRCRICR0 is RES0 otherwise.

ERRCRICR0 is architecturally mapped to memory-mapped register ERRIRQCR4[63:0].

ERRCRICR0 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRCRICR0 is a 64-bit read/write memory-mapped register located at offset 0xEA0.

### Critical Error Interrupt is implemented, recommended layout

#### Configurations

Defined only if all of the following are true:

- The Critical Error Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Critical Error Interrupt is implemented, recommended layout bit assignments are:

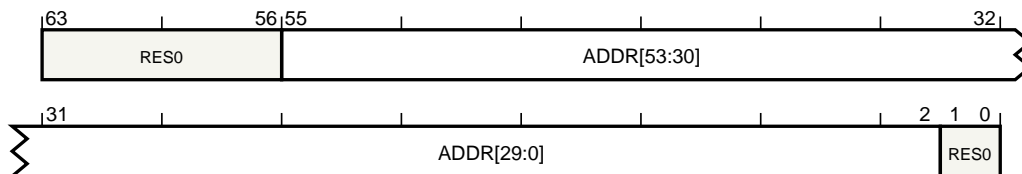


Figure 4.23: ERRCRICR0 Critical Error Interrupt is implemented, recommended layout

#### Bits [63:56,1:0]

Reserved. This field is RES0.

#### ADDR, bits [55:2]

Message Signaled Interrupt address. (ERRCRICR0.ADDR << 2) is the address that the component writes to when signaling the Critical Error Interrupt. Bits [1:0] of the address are always zero.

The physical address size supported by the component is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

This field resets to an architecturally UNKNOWN value on a reset.

### IMPLEMENTATION DEFINED layout

#### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:

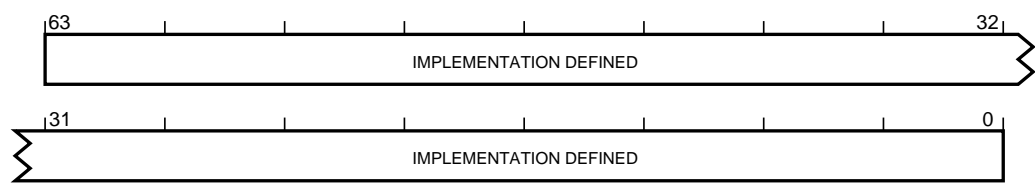


Figure 4.24: ERRCRICR0 IMPLEMENTATION DEFINED layout

**Bits [63:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessibility**

None.

#### 4.3.18 ERRCRICR1, Critical Error Interrupt Configuration Register 1

The ERRCRICR1 characteristics are:

## Purpose

Critical Error Interrupt configuration register.

## Configurations

ERRCRICR1 is present only if all of the following are true:

- Any of the following are true:
  - The Critical Error Interrupt is implemented.
  - The implementation does not use the recommended layout for the `ERRIRQCR<n>` registers.
- Interrupt configuration registers are implemented.

ERRCRICR1 is RES0 otherwise.

ERRCRICR1 is architecturally mapped to memory-mapped register ERRIRQCR5[31:0].

ERRCRICR1 is implemented only as part of a memory-mapped group of error records.

## Attributes

ERRCRICR1 is a 32-bit read/write memory-mapped register located at offset 0xEA8.

**Critical Error Interrupt is implemented, recommended layout**

## Configurations

Defined only if all of the following are true:

- The Critical Error Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Critical Error Interrupt is implemented, recommended layout bit assignments are:



**Figure 4.25: ERRCRICR1 Critical Error Interrupt is implemented, recommended layout**

**DATA, bits [31:0]**

Payload for the message signaled interrupt. This field resets to an architecturally UNKNOWN value on a reset.

## IMPLEMENTATION DEFINED layout

## Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



**Figure 4.26: ERRCRICR1 IMPLEMENTATION DEFINED layout**

**Bits [31:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

## **Accessibility**

None.

### 4.3.19 ERRCRICR2, Critical Error Interrupt Configuration Register 2

The ERRCRICR2 characteristics are:

#### Purpose

Critical Error Interrupt control and configuration register.

#### Configurations

ERRCRICR2 is present only if all of the following are true:

- Any of the following are true:
  - The Critical Error Interrupt is implemented.
  - The implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.
- Interrupt configuration registers are implemented.

ERRCRICR2 is RES0 otherwise.

ERRCRICR2 is architecturally mapped to memory-mapped register ERRIRQCR5[63:32].

ERRCRICR2 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRCRICR2 is a 32-bit read/write memory-mapped register located at offset 0xEAC.

### Critical Error Interrupt is implemented, recommended layout

#### Configurations

Defined only if all of the following are true:

- The Critical Error Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Critical Error Interrupt is implemented, recommended layout bit assignments are:

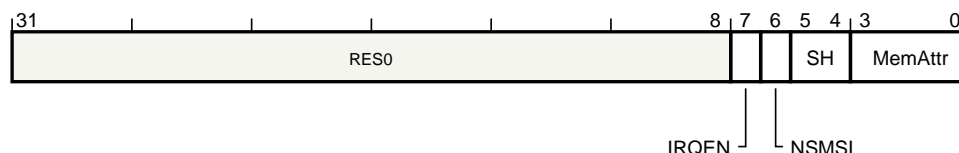


Figure 4.27: ERRCRICR2 Critical Error Interrupt is implemented, recommended layout

#### Bits [31:8]

Reserved. This field is RES0.

#### IRQEN, bit [7]

Message signaled interrupt enable.

#### When the component does not support disabling message signaled interrupts

Message signaled interrupts are always enabled.

This bit is RES0.

#### Otherwise

Enables generation of message signaled interrupts. The possible values of this bit are:

0b0	Disabled.
0b1	Enabled.

This bit resets to 0b0 on a reset.

#### NSMSI, bit [6]

Security attribute. Defines the physical address space for message signaled interrupts.

##### When the component allows Non-secure writes to ERRCRICR2

The Security attribute used for message signaled interrupts is Non-secure.

This bit is RES0.

##### When the component does not support configuring the Security attribute for message signaled interrupts

The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

This bit is RES0.

##### Otherwise

The possible values of this bit are:

0b0	Secure.
0b1	Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a reset.

#### SH, bits [5:4]

Shareability.

##### When the component does not support configuring the Shareability domain

The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the Shareability domain for message signaled interrupts. The possible values of this field are:

0b00	Not shared.
0b10	Outer Shareable.
0b11	Inner Shareable.

All other values are reserved.

This field is ignored when ERRCRICR2.MemAttr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

This field resets to an architecturally UNKNOWN value on a reset.

#### MemAttr, bits [3:0]

Memory type.

##### When the component does not support configuring the memory type

The component does not support configuring the memory type, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the memory type and attributes for message signaled interrupts. The possible values of this field are:



0b0000	Device-nGnRnE memory.
0b0001	Device-nGnRE memory.
0b0010	Device-nGRE memory.
0b0011	Device-GRE memory.
0b0101	Normal memory, Inner Non-cacheable, Outer Non-cacheable.
0b0110	Normal memory, Inner Write-Through, Outer Non-cacheable.
0b0111	Normal memory, Inner Write-Back, Outer Non-cacheable.
0b1001	Normal memory, Inner Non-cacheable, Outer Write-Through.
0b1010	Normal memory, Inner Write-Through, Outer Write-Through.
0b1011	Normal memory, Inner Write-Back, Outer Write-Through.
0b1101	Normal memory, Inner Non-cacheable, Outer Write-Back.
0b1110	Normal memory, Inner Write-Through, Outer Write-Back.
0b1111	Normal memory, Inner Write-Back, Outer Write-Back.

All other values are reserved.

This field resets to an architecturally UNKNOWN value on a reset.

**Note:**

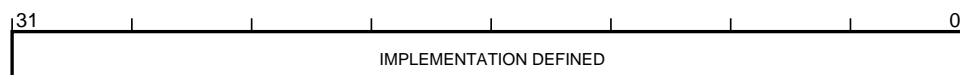
This is the same format as the VMSAv8-64 stage 2 memory region attributes.

## IMPLEMENTATION DEFINED layout

### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



**Figure 4.28: ERRCRICR2 IMPLEMENTATION DEFINED layout**

### Bits [31:0]

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

### Accessibility

None.

### 4.3.20 ERRDEVAFF, Device Affinity Register

The ERRDEVAFF characteristics are:

#### Purpose

For a group of error records that has affinity with a single PE or a group of PEs, ERRDEVAFF is a copy of MPIDR\_EL1 or part of MPIDR\_EL1:

- If the group of error records has affinity with a single PE, the affinity level is 0, ERRDEVAFF reads the same value as MPIDR\_EL1, and ERRDEVAFF.F0V reads-as-one to indicate affinity level 0.
- If the group of error records has affinity with a group of PEs, the affinity level is 1, 2, or 3, parts of ERRDEVAFF reads the same value as parts of MPIDR\_EL1, and the rest of ERRDEVAFF indicates the level.

For example, if the group of PEs is a subset of the PEs at affinity level 1 then all of the following are true:

- All the PEs in the group have the same values in MPIDR\_EL1.{Aff3,Aff2}, and these values are equal to ERRDEVAFF.{Aff3,Aff2}.
- ERRDEVAFF.Aff1 is nonzero and not 0x80, and ERRDEVAFF.{Aff0,F0V} read-as-zero, to indicate at least affinity level 1. The subset of PEs at level 1 that the group of error records has affinity with is indicated by the least-significant set bit in ERRDEVAFF.Aff1. In this example, if ERRDEVAFF.Aff1[2:0] is 0b100, then the group of error records has affinity with the up-to 8 PEs that have MPIDR\_EL1.Aff1[7:3] == ERRDEVAFF.Aff1[7:3].

If RAS System Architecture v1.1 is not implemented, ERRDEVAFF can only describe a group of error records that is affine with a single PE or all the PEs at an affinity level.

#### Configurations

ERRDEVAFF is present only if the group of error records has affinity with a PE or cluster of PEs. ERRDEVAFF is RES0 otherwise.

ERRDEVAFF is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRDEVAFF is a 64-bit read-only memory-mapped register located at offset 0xFA8.

### Field descriptions

The ERRDEVAFF bit assignments are:

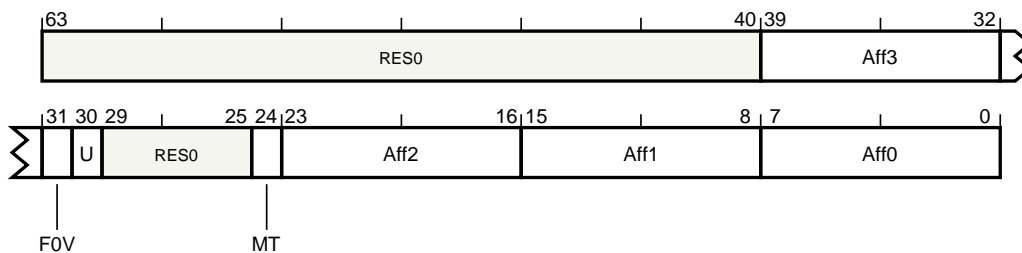


Figure 4.29: ERRDEVAFF

#### Bits [63:40,29:25]

Reserved. This field is RES0.

#### Aff3, bits [39:32]

PE affinity level 3. The MPIDR\_EL1.Aff3 field, viewed from the highest Exception level of the associated PE or PEs.

#### F0V, bit [31]

Indicates that the ERRDEVAFF.Aff0 field is valid. The defined values of this bit are:

0b0	ERRDEVAFF.Aff0 is not valid, and the PE affinity is above level 0 or a subset of level 0.
0b1	ERRDEVAFF.Aff0 is valid, and the PE affinity is at level 0.

#### U, bit [30]

Uniprocessor.

##### When ERRDEVAFF.F0V == 0b1

The MPIDR\_EL1.U bit, viewed from the highest Exception level of the associated PE.

##### Otherwise

Reserved. This bit is UNKNOWN.

#### MT, bit [24]

Multithreaded.

##### When ERRDEVAFF.F0V == 0b1

The MPIDR\_EL1.MT bit, viewed from the highest Exception level of the associated PE.

##### Otherwise

Reserved. This bit is UNKNOWN.

#### Aff2, bits [23:16]

PE affinity level 2.

##### When affine with a PE or PEs at affinity level 2 or below

The MPIDR\_EL1.Aff2 field, viewed from the highest Exception level of the associated PE or PEs.

##### When affine with a sub-set of PEs at affinity level 2

Defines part of the MPIDR\_EL1.Aff2 field, viewed from the highest Exception level of the associated PEs. The defined values of this field are:

0bxxxxxxxx1	ERRDEVAFF.Aff2[7:1] is the value of MPIDR_EL1.Aff2[7:1], viewed from the highest Exception level of the associated PEs.
0bxxxxxxxx10	ERRDEVAFF.Aff2[7:2] is the value of MPIDR_EL1.Aff2[7:2], viewed from the highest Exception level of the associated PEs.
0bxxxxxx100	ERRDEVAFF.Aff2[7:3] is the value of MPIDR_EL1.Aff2[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxxx1000	ERRDEVAFF.Aff2[7:4] is the value of MPIDR_EL1.Aff2[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	ERRDEVAFF.Aff2[7:5] is the value of MPIDR_EL1.Aff2[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	ERRDEVAFF.Aff2[7:6] is the value of MPIDR_EL1.Aff2[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	ERRDEVAFF.Aff2[7] is the value of MPIDR_EL1.Aff2[7], viewed from the highest Exception level of the associated PEs.

##### Otherwise

Indicates whether the PE affinity is at level 3. The defined values of this field are:

0x80	PE affinity is at level 3.
------	----------------------------

All other values are reserved.

#### Aff1, bits [15:8]

PE affinity level 1.

##### When affine with a PE or PEs at affinity level 1 or below

The MPIDR\_EL1.Aff1 field, viewed from the highest Exception level of the associated PE or PEs.

##### When affine with a sub-set of PEs at affinity level 1

Defines part of the MPIDR\_EL1.Aff1 field, viewed from the highest Exception level of the associated PEs. The defined values of this field are:

---

0bxxxxxxx1	ERRDEVAFF.Aff1[7:1] is the value of MPIDR_EL1.Aff1[7:1], viewed from the highest Exception level of the associated PEs.
0bxxxxxx10	ERRDEVAFF.Aff1[7:2] is the value of MPIDR_EL1.Aff1[7:2], viewed from the highest Exception level of the associated PEs.
0bxxxxx100	ERRDEVAFF.Aff1[7:3] is the value of MPIDR_EL1.Aff1[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxx1000	ERRDEVAFF.Aff1[7:4] is the value of MPIDR_EL1.Aff1[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	ERRDEVAFF.Aff1[7:5] is the value of MPIDR_EL1.Aff1[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	ERRDEVAFF.Aff1[7:6] is the value of MPIDR_EL1.Aff1[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	ERRDEVAFF.Aff1[7] is the value of MPIDR_EL1.Aff1[7], viewed from the highest Exception level of the associated PEs.

---

##### Otherwise

Indicates whether the PE affinity is at level 2. The defined values of this field are:

---

0x00	PE affinity is above level 2 or a subset of level 2.
0x80	PE affinity is at level 2.

---

#### Aff0, bits [7:0]

PE affinity level 0.

##### When affine with a PE at affinity level 0

The MPIDR\_EL1.Aff0 field, viewed from the highest Exception level of the associated PE.

##### When affine with a sub-set of PEs at affinity level 0

Defines part of the MPIDR\_EL1.Aff0 field, viewed from the highest Exception level of the associated PEs. The defined values of this field are:

---

0bxxxxxxx1	ERRDEVAFF.Aff0[7:1] is the value of MPIDR_EL1.Aff0[7:1], viewed from the highest Exception level of the associated PEs.
0bxxxxxx10	ERRDEVAFF.Aff0[7:2] is the value of MPIDR_EL1.Aff0[7:2], viewed from the highest Exception level of the associated PEs.
0bxxxxx100	ERRDEVAFF.Aff0[7:3] is the value of MPIDR_EL1.Aff0[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxx1000	ERRDEVAFF.Aff0[7:4] is the value of MPIDR_EL1.Aff0[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	ERRDEVAFF.Aff0[7:5] is the value of MPIDR_EL1.Aff0[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	ERRDEVAFF.Aff0[7:6] is the value of MPIDR_EL1.Aff0[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	ERRDEVAFF.Aff0[7] is the value of MPIDR_EL1.Aff0[7], viewed from the highest Exception level of the associated PEs.

---

**Otherwise**

Indicates whether the PE affinity is at level 1. The defined values of this field are:

0x00	PE affinity is above level 1 or a subset of level 1.
0x80	PE affinity is at level 1.

**Accessibility**

None.

4.3.21 ERRDEVARCH, Device Architecture Register

The ERRDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

Configurations

ERRDEVARCH is implemented only as part of a memory-mapped group of error records.

Attributes

ERRDEVARCH is a 32-bit read-only memory-mapped register located at offset 0xFBC.

Field descriptions

The ERRDEVARCH bit assignments are:

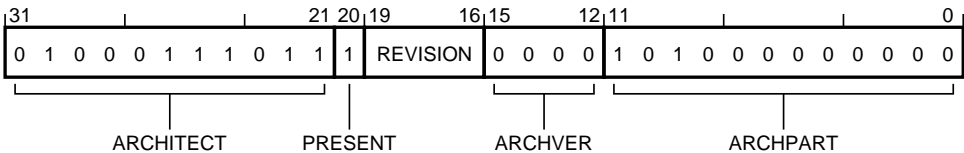


Figure 4.30: ERRDEVARCH

ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code. The defined values of this field are:

0x23B	JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.
-------	--

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present. The defined values of this bit are:

0b0	Device Architecture information not present.
0b1	Device Architecture information present.

This bit reads as 0b1.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component. The defined values of this field are:

0b0000	RAS System Architecture v1.0.
0b0001	RAS System Architecture v1.1. As 0b0000 and also: <ul style="list-style-type: none"><li>• Simplifies <a href="#">ERR&lt;n&gt;STATUS</a>.</li><li>• Adds support for additional ERR&lt;n&gt;MISC&lt;m&gt; registers.</li><li>• Adds support for the optional RAS Timestamp Extension.</li><li>• Adds support for the optional RAS Common Fault Injection Model Extension.</li></ul>

All other values are reserved.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component. The defined values of this field are:

0b0000	RAS System Architecture v1.
--------	-----------------------------

All other values are reserved.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0000.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component. The defined values of this field are:

0xA00	RAS System Architecture.
-------	--------------------------

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA00.

**Accessibility**

None.

### 4.3.22 ERRDEVID, Device Configuration Register

The ERRDEVID characteristics are:

**Purpose**

Provides discovery information for the component.

**Configurations**

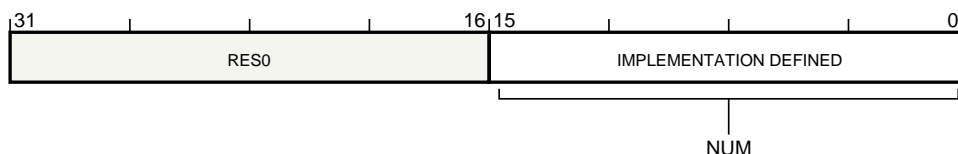
ERRDEVID is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRDEVID is a 32-bit read-only memory-mapped register located at offset 0xFC8.

### Field descriptions

The ERRDEVID bit assignments are:



**Figure 4.31: ERRDEVID**

**Bits [31:16]**

Reserved. This field is RES0.

**NUM, bits [15:0]**

Highest numbered index of the error records in this group, plus one. Each implemented record is owned by a node. A node might own multiple records.

This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise.

This field reads as an IMPLEMENTATION DEFINED value.

### Accessibility

None.



### 4.3.23 ERRERICR0, Error Recovery Interrupt Configuration Register 0

The ERRERICR0 characteristics are:

#### Purpose

Error Recovery Interrupt configuration register.

#### Configurations

ERRERICR0 is present only if all of the following are true:

- Any of the following are true:
  - The Error Recovery Interrupt is implemented.
  - The implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.
- Interrupt configuration registers are implemented.

ERRERICR0 is RES0 otherwise.

ERRERICR0 is architecturally mapped to memory-mapped register ERRIRQCR2[63:0].

ERRERICR0 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRERICR0 is a 64-bit read/write memory-mapped register located at offset 0xE90.

### Error Recovery Interrupt is implemented, recommended layout

#### Configurations

Defined only if all of the following are true:

- The Error Recovery Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Error Recovery Interrupt is implemented, recommended layout bit assignments are:

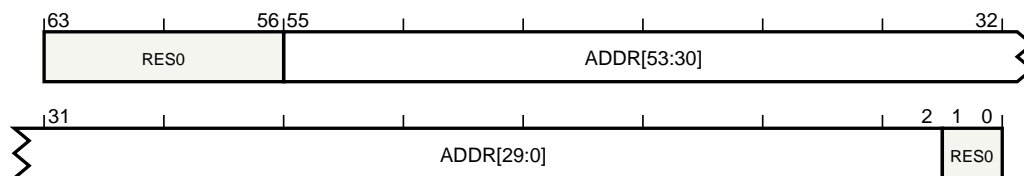


Figure 4.32: ERRERICR0 Error Recovery Interrupt is implemented, recommended layout

#### Bits [63:56,1:0]

Reserved. This field is RES0.

#### ADDR, bits [55:2]

Message Signaled Interrupt address. (ERRERICR0.ADDR << 2) is the address that the component writes to when signaling the Error Recovery Interrupt. Bits [1:0] of the address are always zero.

The physical address size supported by the component is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

This field resets to an architecturally UNKNOWN value on a reset.

### IMPLEMENTATION DEFINED layout

#### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:

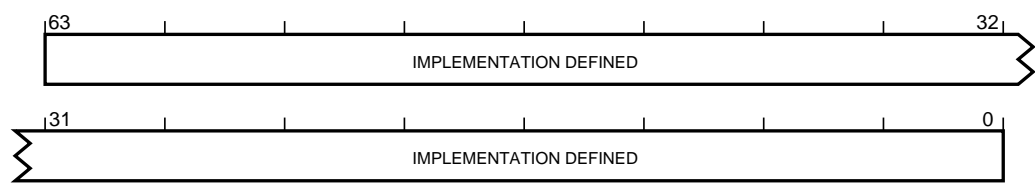


Figure 4.33: ERRERICR0 IMPLEMENTATION DEFINED layout

**Bits [63:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessibility**

None.

#### 4.3.24 ERRERICR1, Error Recovery Interrupt Configuration Register 1

The ERRERICR1 characteristics are:

##### Purpose

Error Recovery Interrupt configuration register.

##### Configurations

ERRERICR1 is present only if all of the following are true:

- Any of the following are true:
  - The Error Recovery Interrupt is implemented.
  - The implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.
- Interrupt configuration registers are implemented.

ERRERICR1 is RES0 otherwise.

ERRERICR1 is architecturally mapped to memory-mapped register ERRIRQCR3[31:0].

ERRERICR1 is implemented only as part of a memory-mapped group of error records.

##### Attributes

ERRERICR1 is a 32-bit read/write memory-mapped register located at offset 0xE98.

#### Error Recovery Interrupt is implemented, recommended layout

##### Configurations

Defined only if all of the following are true:

- The Error Recovery Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Error Recovery Interrupt is implemented, recommended layout bit assignments are:

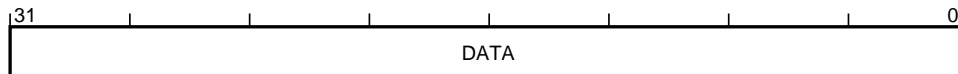


Figure 4.34: ERRERICR1 Error Recovery Interrupt is implemented, recommended layout

##### DATA, bits [31:0]

Payload for the message signaled interrupt. This field resets to an architecturally UNKNOWN value on a reset.

#### IMPLEMENTATION DEFINED layout

##### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



Figure 4.35: ERRERICR1 IMPLEMENTATION DEFINED layout

##### Bits [31:0]

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

## **Accessibility**

None.

4.3.25 ERRERICR2, Error Recovery Interrupt Configuration Register 2

The ERRERICR2 characteristics are:

Purpose

Error Recovery Interrupt control and configuration register.

Configurations

ERRERICR2 is present only if all of the following are true:

- Any of the following are true:
  - The Error Recovery Interrupt is implemented.
  - The implementation does not use the recommended layout for the ERRIRQCR<n> registers.
- Interrupt configuration registers are implemented.

ERRERICR2 is RES0 otherwise.

ERRERICR2 is architecturally mapped to memory-mapped register ERRIRQCR3[63:32].

ERRERICR2 is implemented only as part of a memory-mapped group of error records.

Attributes

ERRERICR2 is a 32-bit read/write memory-mapped register located at offset 0xE9C.

Error Recovery Interrupt is implemented, recommended layout

Configurations

Defined only if all of the following are true:

- The Error Recovery Interrupt is implemented.
- The implementation uses the recommended layout for the ERRIRQCR<n> registers.

The Error Recovery Interrupt is implemented, recommended layout bit assignments are:

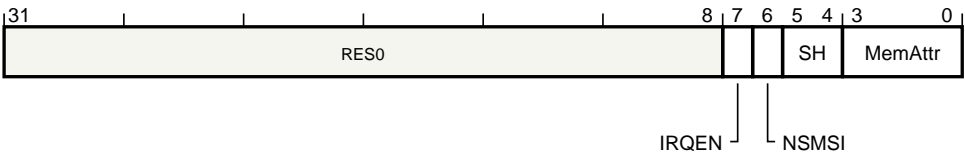


Figure 4.36: ERRERICR2 Error Recovery Interrupt is implemented, recommended layout

Bits [31:8]

Reserved. This field is RES0.

IRQEN, bit [7]

Message signaled interrupt enable.

When the component does not support disabling message signaled interrupts

Message signaled interrupts are always enabled.

This bit is RES0.

Otherwise

Enables generation of message signaled interrupts. The possible values of this bit are:

0b0	Disabled.
0b1	Enabled.

This bit resets to 0b0 on a reset.

#### NSMSI, bit [6]

Security attribute. Defines the physical address space for message signaled interrupts.

##### When the component allows Non-secure writes to ERRERICR2

The Security attribute used for message signaled interrupts is Non-secure.

This bit is RES0.

##### When the component does not support configuring the Security attribute for message signaled interrupts

The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

This bit is RES0.

##### Otherwise

The possible values of this bit are:

0b0	Secure.
0b1	Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a reset.

#### SH, bits [5:4]

Shareability.

##### When the component does not support configuring the Shareability domain

The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the Shareability domain for message signaled interrupts. The possible values of this field are:

0b00	Not shared.
0b10	Outer Shareable.
0b11	Inner Shareable.

All other values are reserved.

This field is ignored when ERRERICR2.MemAttr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

This field resets to an architecturally UNKNOWN value on a reset.

#### MemAttr, bits [3:0]

Memory type.

##### When the component does not support configuring the memory type

The component does not support configuring the memory type, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the memory type and attributes for message signaled interrupts. The possible values of this field are:

0b0000	Device-nGnRnE memory.
0b0001	Device-nGnRE memory.
0b0010	Device-nGRE memory.
0b0011	Device-GRE memory.
0b0101	Normal memory, Inner Non-cacheable, Outer Non-cacheable.
0b0110	Normal memory, Inner Write-Through, Outer Non-cacheable.
0b0111	Normal memory, Inner Write-Back, Outer Non-cacheable.
0b1001	Normal memory, Inner Non-cacheable, Outer Write-Through.
0b1010	Normal memory, Inner Write-Through, Outer Write-Through.
0b1011	Normal memory, Inner Write-Back, Outer Write-Through.
0b1101	Normal memory, Inner Non-cacheable, Outer Write-Back.
0b1110	Normal memory, Inner Write-Through, Outer Write-Back.
0b1111	Normal memory, Inner Write-Back, Outer Write-Back.

All other values are reserved.

This field resets to an architecturally UNKNOWN value on a reset.

**Note:**

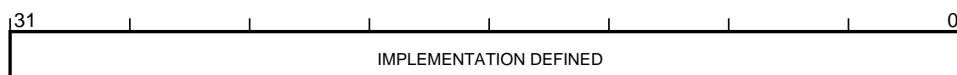
This is the same format as the VMSAv8-64 stage 2 memory region attributes.

## IMPLEMENTATION DEFINED layout

### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



**Figure 4.37: ERRERICR2 IMPLEMENTATION DEFINED layout**

### Bits [31:0]

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

### Accessibility

None.

### 4.3.26 ERRFHICR0, Fault Handling Interrupt Configuration Register 0

The ERRFHICR0 characteristics are:

#### Purpose

Fault Handling Interrupt configuration register.

#### Configurations

ERRFHICR0 is present only if all of the following are true:

- Any of the following are true:
  - The Fault Handling Interrupt is implemented.
  - The implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.
- Interrupt configuration registers are implemented.

ERRFHICR0 is RES0 otherwise.

ERRFHICR0 is architecturally mapped to memory-mapped register ERRIRQCR0[63:0].

ERRFHICR0 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRFHICR0 is a 64-bit read/write memory-mapped register located at offset 0xE80.

### Fault Handling Interrupt is implemented, recommended layout

#### Configurations

Defined only if all of the following are true:

- The Fault Handling Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Fault Handling Interrupt is implemented, recommended layout bit assignments are:

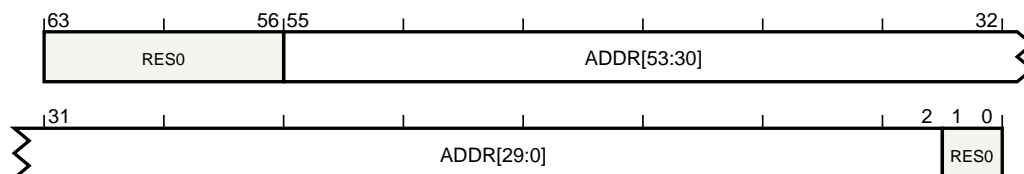


Figure 4.38: ERRFHICR0 Fault Handling Interrupt is implemented, recommended layout

#### Bits [63:56,1:0]

Reserved. This field is RES0.

#### ADDR, bits [55:2]

Message Signaled Interrupt address. (ERRFHICR0.ADDR << 2) is the address that the component writes to when signaling the Fault Handling Interrupt. Bits [1:0] of the address are always zero.

The physical address size supported by the component is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

This field resets to an architecturally UNKNOWN value on a reset.

### IMPLEMENTATION DEFINED layout

#### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.



The IMPLEMENTATION DEFINED layout bit assignments are:

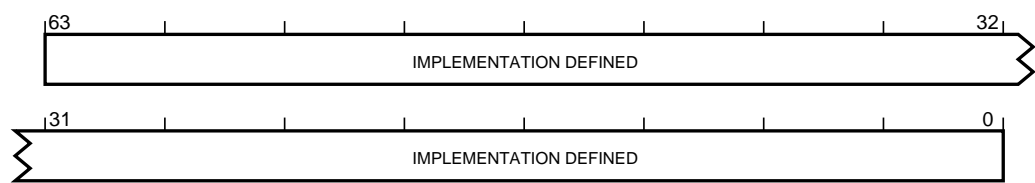


Figure 4.39: ERRFHICR0 IMPLEMENTATION DEFINED layout

**Bits [63:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessibility**

None.

#### 4.3.27 ERRFHC1R1, Fault Handling Interrupt Configuration Register 1

The ERRFHICR1 characteristics are:

## Purpose

Fault Handling Interrupt configuration register.

## Configurations

ERRFHICR1 is present only if all of the following are true:

- Any of the following are true:
  - The Fault Handling Interrupt is implemented.
  - The implementation does not use the recommended layout for the `ERRIRQCR<n>` registers.
- Interrupt configuration registers are implemented.

ERRFHICR1 is RES0 otherwise.

ERRFICR1 is architecturally mapped to memory-mapped register ERRIRQCR1[31:0].

ERRFHICR1 is implemented only as part of a memory-mapped group of error records.

## Attributes

ERRFHICR1 is a 32-bit read/write memory-mapped register located at offset 0xE88.

### Fault Handling Interrupt is implemented, recommended layout

## Configurations

Defined only if all of the following are true:

- The Fault Handling Interrupt is implemented.
- The implementation uses the recommended layout for the [ERRIRQCR<n>](#) registers.

The Fault Handling Interrupt is implemented, recommended layout bit assignments are:



**Figure 4.40: ERRFHICR1 Fault Handling Interrupt is implemented, recommended layout**

**DATA, bits [31:0]**

Payload for the message signaled interrupt. This field resets to an architecturally UNKNOWN value on a reset.

## IMPLEMENTATION DEFINED layout

## Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



**Figure 4.41: ERRFHICR1 IMPLEMENTATION DEFINED layout**

**Bits [31:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

## **Accessibility**

None.

4.3.28 ERRFHICR2, Fault Handling Interrupt Configuration Register 2

The ERRFHICR2 characteristics are:

Purpose

Fault Handling Interrupt control and configuration register.

Configurations

ERRFHICR2 is present only if all of the following are true:

- Any of the following are true:
  - The Fault Handling Interrupt is implemented.
  - The implementation does not use the recommended layout for the ERRIRQCR<n> registers.
- Interrupt configuration registers are implemented.

ERRFHICR2 is RES0 otherwise.

ERRFHICR2 is architecturally mapped to memory-mapped register ERRIRQCR1[63:32].

ERRFHICR2 is implemented only as part of a memory-mapped group of error records.

Attributes

ERRFHICR2 is a 32-bit read/write memory-mapped register located at offset 0xE8C.

Fault Handling Interrupt is implemented, recommended layout

Configurations

Defined only if all of the following are true:

- The Fault Handling Interrupt is implemented.
- The implementation uses the recommended layout for the ERRIRQCR<n> registers.

The Fault Handling Interrupt is implemented, recommended layout bit assignments are:

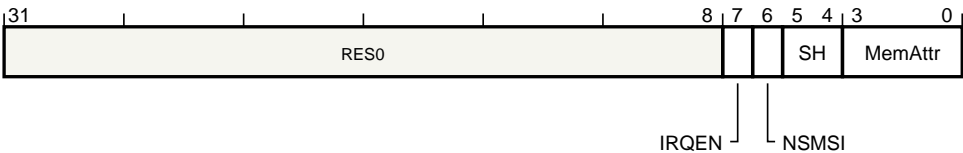


Figure 4.42: ERRFHICR2 Fault Handling Interrupt is implemented, recommended layout

Bits [31:8]

Reserved. This field is RES0.

IRQEN, bit [7]

Message signaled interrupt enable.

When the component does not support disabling message signaled interrupts

Message signaled interrupts are always enabled.

This bit is RES0.

Otherwise

Enables generation of message signaled interrupts. The possible values of this bit are:

0b0	Disabled.
0b1	Enabled.

This bit resets to 0b0 on a reset.

#### NSMSI, bit [6]

Security attribute. Defines the physical address space for message signaled interrupts.

##### When the component allows Non-secure writes to ERRFHICR2

The Security attribute used for message signaled interrupts is Non-secure.

This bit is RES0.

##### When the component does not support configuring the Security attribute for message signaled interrupts

The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

This bit is RES0.

##### Otherwise

The possible values of this bit are:

0b0	Secure.
0b1	Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a reset.

#### SH, bits [5:4]

Shareability.

##### When the component does not support configuring the Shareability domain

The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the Shareability domain for message signaled interrupts. The possible values of this field are:

0b00	Not shared.
0b10	Outer Shareable.
0b11	Inner Shareable.

All other values are reserved.

This field is ignored when ERRFHICR2.MemAttr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

This field resets to an architecturally UNKNOWN value on a reset.

#### MemAttr, bits [3:0]

Memory type.

##### When the component does not support configuring the memory type

The component does not support configuring the memory type, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

This field is RES0.

##### Otherwise

Defines the memory type and attributes for message signaled interrupts. The possible values of this field are:

0b0000	Device-nGnRnE memory.
0b0001	Device-nGnRE memory.
0b0010	Device-nGRE memory.
0b0011	Device-GRE memory.
0b0101	Normal memory, Inner Non-cacheable, Outer Non-cacheable.
0b0110	Normal memory, Inner Write-Through, Outer Non-cacheable.
0b0111	Normal memory, Inner Write-Back, Outer Non-cacheable.
0b1001	Normal memory, Inner Non-cacheable, Outer Write-Through.
0b1010	Normal memory, Inner Write-Through, Outer Write-Through.
0b1011	Normal memory, Inner Write-Back, Outer Write-Through.
0b1101	Normal memory, Inner Non-cacheable, Outer Write-Back.
0b1110	Normal memory, Inner Write-Through, Outer Write-Back.
0b1111	Normal memory, Inner Write-Back, Outer Write-Back.

All other values are reserved.

This field resets to an architecturally UNKNOWN value on a reset.

**Note:**

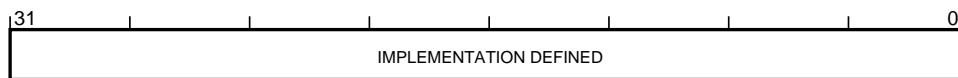
This is the same format as the VMSAv8-64 stage 2 memory region attributes.

## IMPLEMENTATION DEFINED layout

### Configurations

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:



**Figure 4.43: ERRFHICR2 IMPLEMENTATION DEFINED layout**

### Bits [31:0]

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

### Accessibility

None.

4.3.29 ERRGSR, Error Group Status Register

The ERRGSR characteristics are:

Purpose

Shows the status for the records in the group.

Configurations

ERRGSR is implemented only as part of a memory-mapped group of error records.

This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise.

Attributes

ERRGSR is a 64-bit read-only memory-mapped register located at offset 0xE00.

Field descriptions

The ERRGSR bit assignments are:

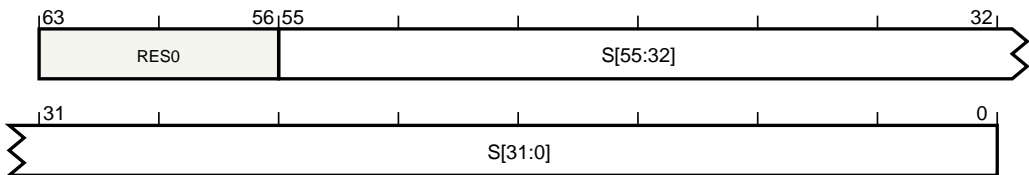


Figure 4.44: ERRGSR

Bits [63:56]

Reserved. This field is RES0.

S[m], bit [m], for m = 0 to 55

The status for error record <m>. A read-only copy of [ERR<m>STATUS.V](#).

When error record <m> is implemented, and error record <m> supports this type of reporting

The defined values of this bit are:

0	No error.
1	One or more errors.

If the Common Fault Injection Model is implemented, up-to 24 records can be implemented meaning bits [55:24] are RES0.

Otherwise

Reserved. This bit is RES0.

Accessibility

None.

### 4.3.30 ERRIIDR, Implementation Identification Register

The ERRIIDR characteristics are:

#### Purpose

Defines the implementer of the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRIIDR is present. ERRIIDR is RES0 if not present.

ERRIIDR is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRIIDR is a 32-bit read-only memory-mapped register located at offset 0xE10.

### Field descriptions

The ERRIIDR bit assignments are:

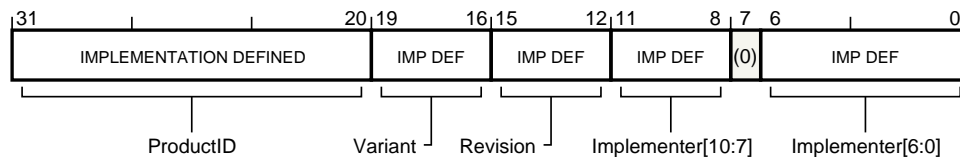


Figure 4.45: ERRIIDR

#### ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the {ERRPIDR1.PART\_1,ERRPIDR0.PART\_0} fields, if ERRPIDR0 and ERRPIDR1 are also present.

This field reads as an IMPLEMENTATION DEFINED value.

#### Variant, bits [19:16]

Component major revision.

ERRIIDR.Variant defines either a variant of the component defined by ERRIIDR.ProductID, or the major revision of the component.

When defining a major revision, ERRIIDR.Variant and ERRIIDR.Revision together form the revision number of the component, with ERRIIDR.Variant being the most significant part and ERRIIDR.Revision the least significant part. When a component is changed, ERRIIDR.Variant or ERRIIDR.Revision is increased to ensure that software can differentiate the different revisions of the component. If ERRIIDR.Variant is increased then ERRIIDR.Revision should be set to 0b0000.

Matches the ERRPIDR2.REVISION field, if ERRPIDR2 is also present.

This field reads as an IMPLEMENTATION DEFINED value.

#### Revision, bits [15:12]

Component minor revision.

When a component is changed:

- If ERRIIDR.Variant and ERRIIDR.Revision together form the revision number of the component then:
  - ERRIIDR.Variant or ERRIIDR.Revision is increased to ensure that software can differentiate the different revisions of the component.
  - If Variant is increased then Revision should be set to 0b0000.



- Otherwise, `ERRIHDR.Revision` is increased to ensure that software can differentiate the different revisions of the component.

Matches the `ERRPIDR3.REVAND` field, if `ERRPIDR3` is also present.

This field reads as an IMPLEMENTATION DEFINED value.

#### Implementer, bits [11:8,6:0]

JEDEC-assigned JEP106 identification code. `ERRIHDR[11:8]` is the JEP106 bank identifier minus 1 and `ERRIHDR[6:0]` is the JEP106 identification code for the designer of the component. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

`ERRIHDR[11:8]` matches `ERRPIDR4.DES_2` and `ERRIHDR[6:0]` match the `{ERRPIDR2.DES_1,ERRPIDR1.DES_0}` fields, if `ERRPIDR{1,2,4}` are also present.

This field reads as an IMPLEMENTATION DEFINED value.

#### Note:

For a component designed by Arm Limited, the JEP106 bank is 5, and the JEP106 identification code is `0x3B`, meaning `ERRIHDR[11:0]` has the value `0x43B`.

Zero is not a valid JEP106 identification code, meaning a value of zero for `ERRIHDR` indicates this register is not implemented.

#### Bit [7]

Reserved. This bit is RES0.

#### Accessibility

None.

### 4.3.31 ERRIMPDEF<n>, IMPLEMENTATION DEFINED Register <0-191>

The ERRIMPDEF<0-191> characteristics are:

#### Purpose

IMPLEMENTATION DEFINED RAS extensions.

#### Configurations

ERRIMPDEF<n> is present if all of the following are true:

- The RAS Common Fault Injection Model Extension is not implemented.
- [ERRDEVID.NUM](#) <= 32.

It is IMPLEMENTATION DEFINED whether ERRIMPDEF<n> is present.

ERRIMPDEF<n> is RES0 if not present.

#### Attributes

ERRIMPDEF<n> is a 64-bit read/write memory-mapped register located at offset  $0 \times 800 + 8 \times n$ .

#### Field descriptions

The ERRIMPDEF<0-191> bit assignments are:

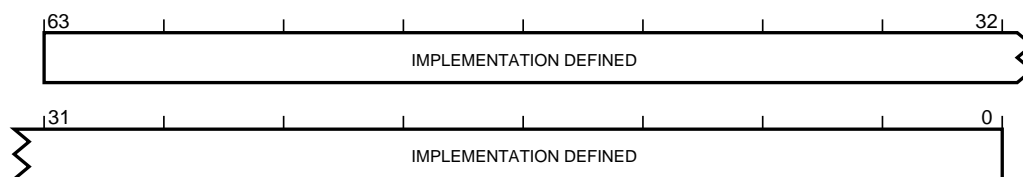


Figure 4.46: ERRIMPDEF<n>

#### Bits [63:0]

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### Accessibility

None.

### 4.3.32 ERRIRQCR<n>, Generic Error Interrupt Configuration Register

The ERRIRQCR<0-15> characteristics are:

#### Purpose

The ERRIRQCR<n> registers are reserved for IMPLEMENTATION DEFINED interrupt configuration registers.

The architecture provides a recommended layout for the ERRIRQCR<n> registers. These registers are named:

- [ERRFHICR0](#), [ERRFHICR1](#), and [ERRFHICR2](#) for the fault handling interrupt controls.
- [ERRERICR0](#), [ERRERICR1](#), and [ERRERICR2](#) for the error recovery interrupt controls.
- [ERRCRICR0](#), [ERRCRICR1](#), and [ERRCRICR2](#) for the critical error interrupt controls.
- [ERRIRQSR](#) for the status register.

This section describes the generic, IMPLEMENTATION DEFINED, format.

#### Configurations

ERRIRQCR<n> is present only if the interrupt configuration registers are implemented. ERRIRQCR<n> is RES0 otherwise.

ERRIRQCR<n> is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRIRQCR<n> is a 64-bit read/write memory-mapped register located at offset  $0 \times E80 + 8 \times n$ .

#### Field descriptions

The ERRIRQCR<0-15> bit assignments are:

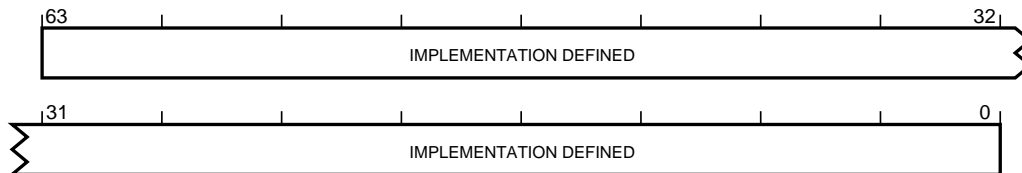


Figure 4.47: ERRIRQCR<n>

#### Bits [63:0]

IMPLEMENTATION DEFINED controls. The content of these registers is IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

#### Accessibility

None.

4.3.33 ERRIRQSR, Error Interrupt Status Register

The ERRIRQSR characteristics are:

Purpose

Interrupt status register.

Configurations

ERRIRQSR is present only if interrupt configuration registers are implemented. ERRIRQSR is RES0 otherwise.

ERRIRQSR is architecturally mapped to memory-mapped register ERRIRQCR15.

ERRIRQSR is implemented only as part of a memory-mapped group of error records.

Attributes

ERRIRQSR is a 64-bit read/write memory-mapped register located at offset 0xEF8.

Recommended layout

Configurations

Defined only if the implementation uses the recommended layout for the ERRIRQCR<n> registers.

The recommended layout bit assignments are:

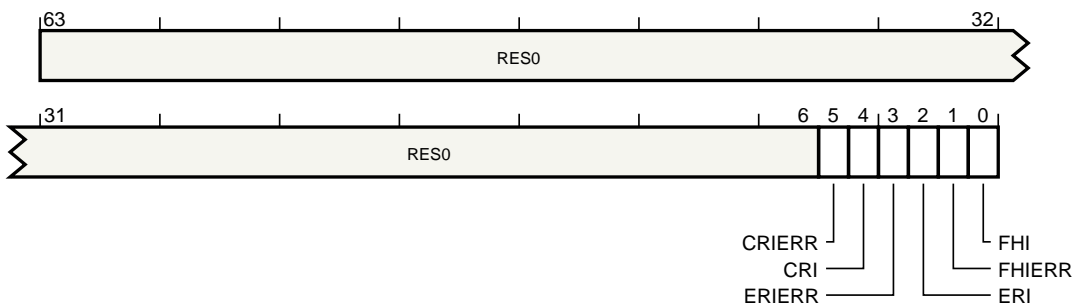


Figure 4.48: ERRIRQSR recommended layout

Bits [63:6]

Reserved. This field is RES0.

CRIERR, bit [5]

Critical Error Interrupt error.

When the Critical Error Interrupt is implemented

The possible values of this bit are:

0b0	Critical Error Interrupt write has not returned an error since this bit was last cleared to zero.
0b1	Critical Error Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a reset.

Otherwise

Reserved. This bit is RES0.

#### CRI, bit [4]

Critical Error Interrupt write in progress.

#### When the Critical Error Interrupt is implemented

The defined values of this bit are:

0b0	Critical Error Interrupt write not in progress.
0b1	Critical Error Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

This bit is read-only.

#### Note:

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual [ERR<n>STATUS](#) registers.

#### Otherwise

Reserved. This bit is RES0.

#### ERIERR, bit [3]

Error Recovery Interrupt error.

#### When the Error Recovery Interrupt is implemented

The possible values of this bit are:

0b0	Error Recovery Interrupt write has not returned an error since this bit was last cleared to zero.
0b1	Error Recovery Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a reset.

#### Otherwise

Reserved. This bit is RES0.

#### ERI, bit [2]

Error Recovery Interrupt write in progress.

#### When the Error Recovery Interrupt is implemented

The defined values of this bit are:

0b0	Error Recovery Interrupt write not in progress.
0b1	Error Recovery Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

This bit is read-only.

#### Note:

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual [ERR<n>STATUS](#) registers.

**Otherwise**

Reserved. This bit is RES0.

**FHIERR, bit [1]**

Fault Handling Interrupt error.

**When the Fault Handling Interrupt is implemented**

The possible values of this bit are:

0b0	Fault Handling Interrupt write has not returned an error since this bit was last cleared to zero.
0b1	Fault Handling Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

This bit resets to an architecturally UNKNOWN value on a reset.

**Otherwise**

Reserved. This bit is RES0.

**FHI, bit [0]**

Fault Handling Interrupt write in progress.

**When the Fault Handling Interrupt is implemented**

The defined values of this bit are:

0b0	Fault Handling Interrupt write not in progress.
0b1	Fault Handling Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

This bit is read-only.

**Note:**

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual [ERR<n>STATUS](#) registers.

**Otherwise**

Reserved. This bit is RES0.

**IMPLEMENTATION DEFINED layout**

**Configurations**

Defined only if the implementation does not use the recommended layout for the [ERRIRQCR<n>](#) registers.

The IMPLEMENTATION DEFINED layout bit assignments are:

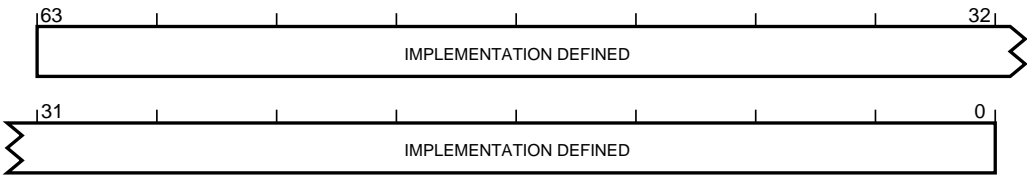


Figure 4.49: ERRIRQSR IMPLEMENTATION DEFINED layout

**Bits [63:0]**

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessibility**

None.

#### 4.3.34 ERRPIDR0, Peripheral Identification Register 0

The ERRPIDR0 characteristics are:

## Purpose

Provides discovery information about the component.

## Configurations

It is IMPLEMENTATION DEFINED whether ERRPIDR0 is present. ERRPIDR0 is RES0 if not present.

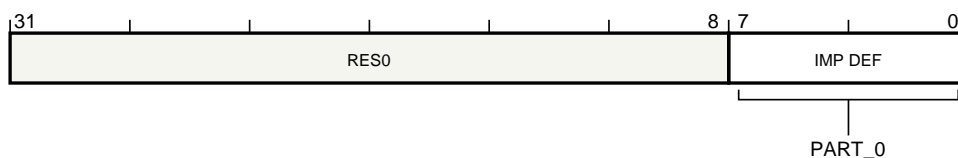
ERRPIDR0 is implemented only as part of a memory-mapped group of error records.

## Attributes

ERRPIDR0 is a 32-bit read-only memory-mapped register located at offset 0xFE0.

## Field descriptions

The ERRPIDR0 bit assignments are:



**Figure 4.50: ERRPIDR0**

**Bits [31:8]**

Reserved. This field is RES0.

**PART\_0, bits [7:0]**

Part number, bits [7:0].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in `ERRPIDR1.PART_1` and `ERRPIDR0.PART_0`. There are 8 bits, `ERRPIDR2.REVISION` and `ERRPIDR3.REVAND`, available to define the revision of the component.
- If a 16-bit part number is used, it is stored in `ERRPIDR2.PART_2`, `ERRPIDR1.PART_1` and `ERRPIDR0.PART_0`. There are 4 bits, `ERRPIDR3.REVISION`, available to define the revision of the component.

This field reads as an IMPLEMENTATION DEFINED value.

## Accessibility

None.



### 4.3.35 ERRPIDR1, Peripheral Identification Register 1

The ERRPIDR1 characteristics are:

#### Purpose

Provides discovery information about the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRPIDR1 is present. ERRPIDR1 is RES0 if not present.

ERRPIDR1 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRPIDR1 is a 32-bit read-only memory-mapped register located at offset 0xFE4.

### Field descriptions

The ERRPIDR1 bit assignments are:

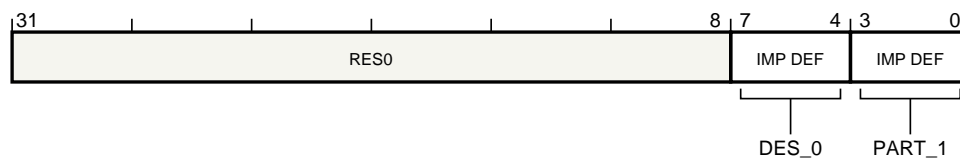


Figure 4.51: ERRPIDR1

#### Bits [31:8]

Reserved. This field is RES0.

#### DES\_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0]. ERRPIDR1.DES\_0 and [ERRPIDR2.DES\\_1](#) together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

This field reads as an IMPLEMENTATION DEFINED value.

#### Note:

For a component designed by Arm Limited, the JEP106 identification code is 0x3B.

#### PART\_1, bits [3:0]

Part number, bits [11:8].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in ERRPIDR1.PART\_1 and [ERRPIDR0.PART\\_0](#). There are 8 bits, [ERRPIDR2.REVISION](#) and [ERRPIDR3.REVAND](#), available to define the revision of the component.
- If a 16-bit part number is used, it is stored in [ERRPIDR2.PART\\_2](#), ERRPIDR1.PART\_1 and [ERRPIDR0.PART\\_0](#). There are 4 bits, [ERRPIDR3.REVISION](#), available to define the revision of the component.

This field reads as an IMPLEMENTATION DEFINED value.

### Accessibility

None.

### 4.3.36 ERRPIDR2, Peripheral Identification Register 2

The ERRPIDR2 characteristics are:

#### Purpose

Provides discovery information about the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRPIDR2 is present. ERRPIDR2 is RES0 if not present.

ERRPIDR2 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRPIDR2 is a 32-bit read-only memory-mapped register located at offset 0xFE8.

### The component uses a 12-bit part number

#### Configurations

Defined only if the component uses a 12-bit part number.

The the component uses a 12-bit part number bit assignments are:

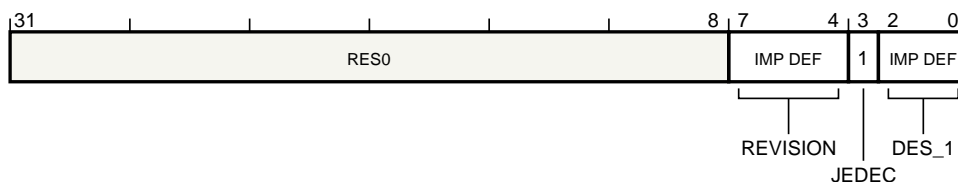


Figure 4.52: ERRPIDR2 the component uses a 12-bit part number

#### Bits [31:8]

Reserved. This field is RES0.

#### REVISION, bits [7:4]

Component major revision. ERRPIDR2.REVISION and [ERRPIDR3.REVAND](#) together form the revision number of the component, with ERRPIDR2.REVISION being the most significant part and [ERRPIDR3.REVAND](#) the least significant part. When a component is changed, ERRPIDR2.REVISION or [ERRPIDR3.REVAND](#) are increased to ensure that software can differentiate the different revisions of the component. If ERRPIDR2.REVISION is increased then [ERRPIDR3.REVAND](#) should be set to 0b0000.

This field reads as an IMPLEMENTATION DEFINED value.

#### JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used. This bit reads as 0b1.

#### DES\_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4]. [ERRPIDR1.DES\\_0](#) and ERRPIDR2.DES\_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

This field reads as an IMPLEMENTATION DEFINED value.

#### Note:

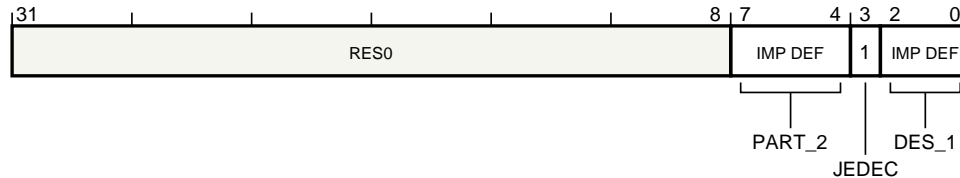
For a component designed by Arm Limited, the JEP106 identification code is 0x3B.

## The component uses a 16-bit part number

### Configurations

Defined only if the component uses a 16-bit part number.

The the component uses a 16-bit part number bit assignments are:



**Figure 4.53: ERRPIDR2 the component uses a 16-bit part number**

### Bits [31:8]

Reserved. This field is RES0.

### PART\_2, bits [7:4]

Part number, bits [15:12].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in [ERRPIDR1.PART\\_1](#) and [ERRPIDR0.PART\\_0](#). There are 8 bits, [ERRPIDR2.REVISION](#) and [ERRPIDR3.REVAND](#), available to define the revision of the component.
- If a 16-bit part number is used, it is stored in [ERRPIDR2.PART\\_2](#), [ERRPIDR1.PART\\_1](#) and [ERRPIDR0.PART\\_0](#). There are 4 bits, [ERRPIDR3.REVISION](#), available to define the revision of the component.

This field reads as an IMPLEMENTATION DEFINED value.

### JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used. This bit reads as 0b1.

### DES\_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4]. [ERRPIDR1.DES\\_0](#) and [ERRPIDR2.DES\\_1](#) together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

This field reads as an IMPLEMENTATION DEFINED value.

### Note:

For a component designed by Arm Limited, the JEP106 identification code is 0x3B.

## Accessibility

None.

### 4.3.37 ERRPIDR3, Peripheral Identification Register 3

The ERRPIDR3 characteristics are:

#### Purpose

Provides discovery information about the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRPIDR3 is present. ERRPIDR3 is RES0 if not present.

ERRPIDR3 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRPIDR3 is a 32-bit read-only memory-mapped register located at offset 0xFEC.

### The component uses a 12-bit part number

#### Configurations

Defined only if the component uses a 12-bit part number.

The the component uses a 12-bit part number bit assignments are:

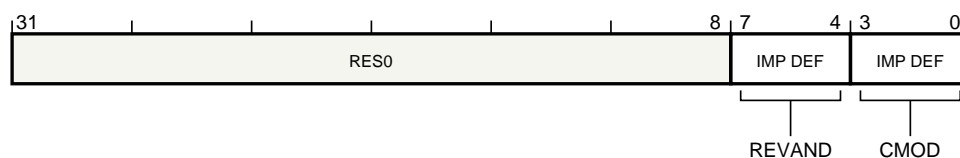


Figure 4.54: ERRPIDR3 the component uses a 12-bit part number

#### Bits [31:8]

Reserved. This field is RES0.

#### REVAND, bits [7:4]

Component minor revision. [ERRPIDR2.REVISION](#) and ERRPIDR3.REVAND together form the revision number of the component, with [ERRPIDR2.REVISION](#) being the most significant part and ERRPIDR3.REVAND the least significant part. When a component is changed, [ERRPIDR2.REVISION](#) or ERRPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. If [ERRPIDR2.REVISION](#) is increased then ERRPIDR3.REVAND should be set to 0b0000.

This field reads as an IMPLEMENTATION DEFINED value.

#### CMOD, bits [3:0]

Customer Modified.

Indicates the component has been modified.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

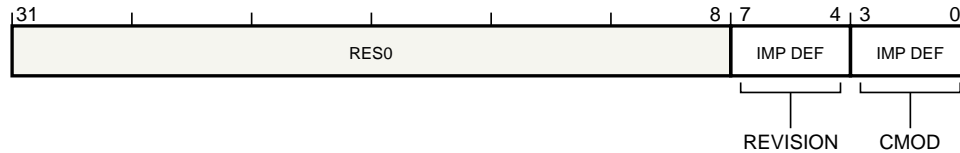
This field reads as an IMPLEMENTATION DEFINED value.

## The component uses a 16-bit part number

### Configurations

Defined only if the component uses a 16-bit part number.

The the component uses a 16-bit part number bit assignments are:



**Figure 4.55: ERRPIDR3 the component uses a 16-bit part number**

### Bits [31:8]

Reserved. This field is RES0.

### REVISION, bits [7:4]

Component revision. When a component is changed, ERRPIDR3.REVISION is increased to ensure that software can differentiate the different revisions of the component.

This field reads as an IMPLEMENTATION DEFINED value.

### CMOD, bits [3:0]

Customer Modified.

Indicates the component has been modified.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

This field reads as an IMPLEMENTATION DEFINED value.

## Accessibility

None.

### 4.3.38 ERRPIDR4, Peripheral Identification Register 4

The ERRPIDR4 characteristics are:

#### Purpose

Provides discovery information about the component.

#### Configurations

It is IMPLEMENTATION DEFINED whether ERRPIDR4 is present. ERRPIDR4 is RES0 if not present.

ERRPIDR4 is implemented only as part of a memory-mapped group of error records.

#### Attributes

ERRPIDR4 is a 32-bit read-only memory-mapped register located at offset 0xFD0.

### Field descriptions

The ERRPIDR4 bit assignments are:

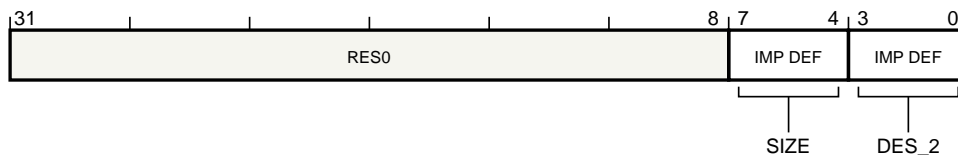


Figure 4.56: ERRPIDR4

#### Bits [31:8]

Reserved. This field is RES0.

#### SIZE, bits [7:4]

Size of the component.

The distance from the start of the address space used by this component to the end of the component identification registers.

A value of 0b0000 means one of the following is true:

- The component uses a single 4KB block.
- The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.

Any other value means the component occupies  $2^{\text{ERRPIDR4.SIZE}}$  4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

This field reads as an IMPLEMENTATION DEFINED value.

#### DES\_2, bits [3:0]

Designer, JEP106 continuation code. This is the JEDEC-assigned JEP106 bank identifier for the designer of the component, minus 1. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

This field reads as an IMPLEMENTATION DEFINED value.

#### Note:

For a component designed by Arm Limited, the JEP106 bank is 5, meaning this field has the value 0x4.

## **Accessibility**

None.

## Chapter 5

### Appendix

#### 5.1 Release notes for issue D.a

This release of the Arm® RAS Supplement represents a major rewrite of the manual. The most significant change is to use the rules-based writing approach. There are also some changes to the order and flow of the manual, compared to previous releases, as well as errata fixes, clarifications, relaxations, and new features.

##### Section 1.2.2 *Error propagation*

This was previously section 1.3.2. If a consumer of a *partial* write to a [protection granule](#) removes [poison](#) from the location, but the unchanged portion of the location is still [corrupt](#), then the previously [deferred error](#) is now an [undetected error](#), meaning the error must be considered to be [silently propagated](#) by the consumer.

##### Section 1.3.2 *Error handling and recovery*

This was previously section 1.2.2. Provide a definition for [deferred error](#), and how this relates to [poisoning](#). The previous description had *deferred* defined as *not taking action immediately*. However, deferring an error is taking action, so the definition is updated and extended with examples. Additionally, deferring an error means that a software error recovery agent does not need to be invoked at that time.

##### Sections 2.1 *PE error handling*, 2.2 *Generating error exceptions*, 2.3 *Taking error exceptions*

Use of *External abort* is changed to either [in-band error response](#) or [error exception](#) as appropriate, for consistency across this specification.

##### Section 2.1.2 *PE error propagation*

The exceptional case where an error [propagated by the Processing element \(PE\)](#) is *uncontained by the PE* (previously in section 2.1.4) is merged into the definition of [silently propagated by the PE](#). This removes a source of confusion with the [Uncontainable \(UC\)](#) error state described later.



The exceptional case is also changed such that:

- Any action when a PE takes an [error exception](#) is not silent propagation of the error.
- The case relating to an [Error synchronization event](#) applies only for the `ESB` case. That is, an implicit [Error synchronization event](#) when `SError` interrupts are masked does not contain the error.
- The propagation by an instruction is restricted to the consumption of the corrupt data, and not any other form of propagation. For example, propagating the error by causing an imprecise exception is not permitted by the relaxation. An example of a store of the corrupt data is given.

### Section 2.3 [Taking error exceptions](#)

This is a new section, previously part of section 2.2. The requirement to take error exceptions on instruction fetches as a synchronous abort is from [FEAT\\_DoubleFault](#), not [FEAT\\_RASv1p1](#) as previously stated. This also applies to error exceptions from translation table walks on instruction fetches.

Definitions for error exceptions on Allocation Tag accesses, when `FEAT_MTE` is implemented, and on cache maintenance operations, are added.

### Section 2.3 [Taking error exceptions](#)

The terminology for the [PE error state](#) on taking an error exception is updated, as follows:

- *Uncontainable error* becomes [Uncontainable \(UC\)](#).
- *Unrecoverable error* becomes [Unrecoverable state \(UEU\)](#).
- *Recoverable error* becomes [Recoverable state \(UER\)](#).
- *Restartable error* becomes [Restartable state \(UEO\)](#).

[Corrected \(CE\)](#), [Uncategorized error](#), and [IMPLEMENTATION DEFINED syndrome](#) are also brought into this taxonomy.

An issue with the previous definitions is corrected, where an error that meets most of the criteria for [Unrecoverable state \(UEU\)](#) but is not [synchronized by Error synchronization events](#) was not covered by any definition. Such errors are reported as [Uncontainable \(UC\)](#).

### Section 2.3 [Taking error exceptions](#)

The definition of [recover execution](#), used to define [Recoverable state \(UER\)](#), is relaxed to allow *executing the instruction at the return address will overwrite any corrupt state with the correct values*.

### Section 2.4 [Error synchronization event](#)

The definition of the [Error synchronization event](#) is corrected. In particular, the circular definition with [Unrecoverable state \(UEU\)](#) is removed.

The definition of the event is also extended to cover the state of the PE and memory. That is, on completing the Error synchronization event, if no error is pending by it, the PE and memory state are consistent with the PE having executed instructions in program order before it, allowing that software to be recovered.

### Section 2.4.2 [Extension for synchronization at exception entry and return](#)

An `ERET` instruction is not required to generate an [Error synchronization event](#) if it generates an exception. This also applies to `ERETAA` and `ERETAB`.

## Section 3 [RAS System Architecture](#)

Throughout this section, the distinction between the component that detects or consumes the error and the [node](#) that records the error is made more consistent, as are the descriptions for components that might *consume* an error, similar to how a PE does. For instance, in many places behaviors were defined only for a node *detecting* an error. These are changed to either *detecting or consuming* or, where more applicable, when the node *records* an error in its [error record](#).

### Section 3.2.1 [Component error states](#)

This was previously section 3.1.4.

The *types of error* is redefined as the [component error state](#), and modified from the previous release. The changes are similar to those in section 2.3 (see above). However, the terminology is unchanged.

The previous deprecation of use of [Restartable error](#), [Signaled error](#), [Recoverable error](#), and [Latent error](#) is removed. For many components, [Uncontainable error](#) and [Recoverable error](#), or [Uncontainable error](#) and [Restartable error](#) are the most applicable error types.

The linking of [component error state](#) to whether the error has been [silently propagated](#) is clarified as only applying to propagation by the component that reported the error. For example, the component might signal the error to the consumer (which propagates the error but not silently) only for the error to then be silently propagated by the consumer. The producer is not responsible for recording this case as silently propagated.

The definitions are also updated to change *the node has halted operation* to *the component has entered a service failure mode*, as this term is used elsewhere. It is clarified that the node records an error as [Uncorrected error](#) if the component enters a [service failure mode](#).

### Section 3.2.2 [Writing the error record](#)

Clarifications for the values written to [ERR<n>STATUS.{UE,DE,CE}](#) in [Overwriting the error syndrome](#) and to [ERR<n>STATUS.ER](#) in [Keeping the previous error syndrome](#) are added.

### Section 3.2.2 [Writing the error record](#)

In the section [Detecting multiple errors](#), the IMPLEMENTATION DEFINED behavior for counting the errors is relaxed to IMPLEMENTATION DEFINED *and might be UNPREDICTABLE*. For example, an implementation might choose to document this case as UNPREDICTABLE because of complex timing paths.

### Section 3.2.2 [Writing the error record](#)

In [Overwriting the error syndrome](#) when clearing [ERR<n>STATUS.AV](#) to 0b0, [ERR<n>ADDR](#) also becomes UNKNOWN. This is a relaxation from the previous behavior.

### Section 3.2.4 [Error syndrome](#)

This is a new section. This is added to provide more information on setting [ERR<n>STATUS.PN](#).

### Section 3.2.5 [Synchronization and error record accesses](#)

There is a finite time guarantee on accesses to error records.

### Section 3.2.6 [Bridges to other architectures](#)

This is a new section. This is added to cover cases where, for example, PCIe-AER defines how the errors are recorded, not this specification.

### Section 3.2.7 [Software faults](#)

This was previously section 3.1.5. The recommendation to generate a software fault interrupt is kept as an option, but, based on user feedback, no longer recommended. Other options are described.

### Section 3.7 [Standard format Corrected error counter](#)

The rules for which counter counts an error when a pair of counters is implemented are described in more detail.

### Section 3.11 [Common Fault Injection Model Extension](#)

Relaxed the requirement for a node to be able to inject all possible combinations of error type and syndrome bits. There is also a relationship between the capabilities described in [ERR<n>PFGF](#) and those described by [ERR<n>FR](#).

### Sections 3.11 [Common Fault Injection Model Extension](#), 4.1 [Memory-mapped view](#), 4.3.25 [ERRIMPDEF<n>](#)

If the [Common Fault Injection Model Extension](#) is not implemented and 24 or fewer error records are implemented, then a space for IMPLEMENTATION DEFINED registers is defined as [ERRIMPDEF<n>](#). The [ERRIMPDEF<n>](#) registers might be used for IMPLEMENTATION DEFINED fault injection.

### Section 4.1 [Memory-mapped view](#)

In the [Memory-mapped single error record view](#), [ERR<n>FR](#) is at a 64-byte aligned address.

### Sections 4.1 [Memory-mapped view](#), 4.3.22 [ERRDEVID](#), 4.3.23 [ERRGSR](#)

The number of error records that can be in a 4KB view is 56 if the [Common Fault Injection Model Extension](#) is not implemented and 24 if the [Common Fault Injection Model Extension](#) is implemented.

## Section 4.2 *Reset values*

The statements about reset values are removed, as each register field description now contains the correct information.

A further statement about the requirement to reset fields is added. Much of the behavior in particular of the [ERR<n>STATUS](#) register fields depends on inductive steps from an initial value of the register. For this to hold there has to be either a defined reset value or other IMPLEMENTATION DEFINED choices of behavior.

## Sections 4.3.5 [ERR<n>ADDR](#), 4.3.8 [ERR<n>MISC0](#), 4.3.9 [ERR<n>MISC1](#), 4.3.10 [ERR<n>MISC2](#), 4.3.11 [ERR<n>MISC3](#), 4.3.15 [ERR<n>STATUS](#)

Writes to [ERR<n>ADDR](#) and [ERR<n>MISC<m>](#) are permitted when allowed by the [Common Fault Injection Model Extension](#). Similarly for some fields in [ERR<n>STATUS](#).

## Section 4.3.5 [ERR<n>ADDR](#)

[ERR<n>ADDR](#) is optional if the error record does not include an address record. This was previously stated in the description of the error record.

## Section 4.3.7 [ERR<n>FR](#)

The [ERR<n>FR](#).{CE,DE,UEO,UER,UEU,UC,FRX} fields are added. These optionally allow a node to declare which error types it supports recording.

## Section 4.3.15 [ERR<n>STATUS](#)

[ERR<n>STATUS.SERR](#) codes for *Peripheral Component Interconnect Express* (PCIe) and bridges are added.

## Section 4.3.15 [ERR<n>STATUS](#)

The long name for [ERR<n>STATUS.SERR](#) is changed to *error code*. This was previously *error type* which can cause confusion with other uses of this term in the manual.

## Section 4.3.15 [ERR<n>STATUS](#)

The implemented set of valid values that the [ERR<n>STATUS](#).{IERR,SERR} fields can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

## Sections 4.3.17 [ERRCRICR0](#), 4.3.18 [ERRCRICR1](#), 4.3.19 [ERRCRICR2](#), 4.3.23 [ERRERICR0](#), 4.3.24 [ERRERICR1](#), 4.3.25 [ERRERICR2](#), 4.3.26 [ERRFHICR0](#), 4.3.27 [ERRFHICR1](#), 4.3.28 [ERRFHICR2](#)

When the recommended layout is not used, the registers might be IMPLEMENTATION DEFINED.

[ERRDEVAFF](#) is extended with a format for specifying affinity with a subset of a group of PEs at an affinity level.

# Glossary

## Asynchronous exception

Asynchronous exceptions are also known as interrupts. In the Armv8 architecture, an asynchronous exception is one for which any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

## Availability

Readiness for correct service.

## Baseboard Management Controller

A PE dedicated to system control and monitoring.

## BIST

Built-in self-test

## Built-in self-test

A mechanism that permits a machine to test itself.

## Catastrophic failure

A failure with harmful consequences that are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

## CE

Correctable or Corrected Error

## Completer

An agent in a computing system that responds to and completes a memory transaction initiated by a Requester.

## Contained or containable error

An error that is not uncontained or uncontainable.

## Containment

Limiting or preventing the silent propagation of an error. Arm recommends that the scope to which an error is contained is specified.

## Correctable or Corrected Error

An error that is detected by hardware and that hardware can correct / has corrected.

## DECTED

Double error correct, triple error detect EDAC. This can detect a single, double or triple bit error and correct a single or double bit error in a protection granule.

## Deferred error

An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to a consumer.

**Detected error**

An error that has been detected and signaled to a consumer.

**Detected Uncorrected Error**

A detected error that has not been corrected and causes failure.

**Device memory**

Memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

**Double fault**

A second error that is detected when the PE is in the process of handling a first error condition.

**DUE**

Detected Uncorrected Error

**DUE FIT rate**

The FIT rate for failures from a DUE.

**ECC**

Error Correction Code

**EDAC**

Error Detection and Correction Code

**EDC**

Error Detection Code

**Error**

Deviation from correct service or a correct value.

**Error Correction Code or Error Detection and Correction Code**

A code capable of detecting and correcting a number of errors.

**Error Detection Code**

A code capable of detecting, but not correcting, errors.

**Error log**

Historical data recorded about errors, usually by software.

**Error propagation**

Passing an error from a producer to a consumer.

**Error record**

Data recorded about an error, usually by hardware.

**Error synchronization event**

One of:

- Executing an `ESB` instruction.
- Taking an exception to an Exception level using AArch64, [FEAT\\_IESB](#) is implemented, and either:
  - The appropriate `SCTLR_ELx.IESB` bit is `0b1`.
  - [FEAT\\_DoubleFault](#) is implemented, the Exception level is `EL3`, and `SCTLR_EL3.NMEA` is `0b1`.

- Executing an Exception Return instruction at an Exception level using AArch64, [FEAT\\_IESB](#) is implemented, and either:
  - The appropriate SCTLR\_ELx.IESB bit is 0b1.
  - [FEAT\\_DoubleFault](#) is implemented, the Exception level is EL3, and SCR\_EL3.NMEA is 0b1.

## Exception

An exception handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

## External abort

Either:

- An in-band error that is generated as a response to a transaction. The name derives from the specific case of an abort generated by a memory system that is external to a PE, but the concept can apply to other interfaces.
- A type of exception in the Arm architecture, generated when consuming an in-band error response.

## Fail-safe

A failure mode in which the PE and other system components switch to backup mechanisms that keep processing instructions and data to allow either a safe shutdown or restart of the system, or to continue processing critical functions, or both.

## Fail-secure

A failure mode in which the PE and other system components fail but the system is secured to allow either a safe shutdown or restart of the system, or to continue processing critical functions without exposing secret data, or both.

## Fail-signaled

A failure mode in which the PE signals to the system that it has failed. It might continue to process instructions, but the system must ignore its output, or treat all outputs as detected errors.

## Fail-silent

Failure mode in which the PE and all other system components (such as DMAs) stop processing instructions. A watchdog process will detect the failure and restart the system with an Error Recovery reset.

## Failure

The event of deviation from correct service.

## Failure-in-Time

The number of expected failures per billion hours of operation.

## Fault

The cause of an error.

## Fault injection

The deliberate injection of faults into a system for testing.

## Fault prevention

Designing a system to avoid faults.

## Fault removal

Logic or other mechanisms for detecting faults and correcting or bypassing their effect.

## Field Replaceable Unit

A component or unit in a system that can be replaced without return to base.

**FIT**

Failure-in-Time

**FRU**

Field Replaceable Unit

**General-purpose registers**

The registers that the base instructions use for processing:

- In AArch32 state the general-purpose registers are R0-R14.
- In AArch64 state the general-purpose registers are R0-R30.

**Generic Interrupt Controller**

Arm system architecture interrupt controller for IRQ and FIQ interrupt exceptions.

**GIC**

Generic Interrupt Controller

**Hardware fault**

A fault that originates in, or affects, hardware.

**Imprecise exception**

An exception that is not precise.

**Infected**

Being in error.

**Interrupt**

In a PE context, an asynchronous exception. There are three interrupt exceptions: IRQ, FIQ and SError. IRQ and FIQ are always precise. In a system architecture context, an asynchronous event sent to a PE or GIC for processing as an interrupt exception.

**Isolation**

Limiting the impact of an error only to components that actually try to use corrupted data.

**Latent error or latent fault**

An error that is present in a system but not yet detected.

**MBIST**

Memory BIST

**Minor failure**

A failure with harmful consequences that are of a similar cost to the benefits that are provided by correct service delivery.

**MSI**

Message Signaled Interrupt

**Normal memory**

Used for bulk memory operations. Hardware might speculatively read these locations.

**PCIe**

Peripheral Component Interconnect Express

**PE**

Processing element

**Peripheral Component Interconnect Express (PCI Express or PCIe)**

A high-speed serial computer expansion bus standard maintained and developed by the PCI Special Interest Group.

**Persistent fault**

A fault that is not transient.

**PFA**

Predictive Failure Analysis

**Poisoned**

State that has been marked as being in error so that subsequent consumption of the state will be treated as a detected error.

**PPI**

Private Peripheral Interrupt

**Precise exception**

An exception where the exception handler receives the state of the PE and the state of the memory system 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. The state of the PE and the state of the memory do not include instructions that occurred after this point.

**Predictive Failure Analysis**

Mechanisms to analyze errors and predict future failures.

**Processing element (PE)**

The abstract machine defined in the Armv8 architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Armv8 architecture conforms with the behaviors described in the corresponding Arm Architecture Reference Manual.

**Propagated**

See Error propagation.

**Protection granule**

A quantum of memory for which an EDC or ECC provides detection or correction. For example, a 72/64 SECDED ECC scheme has a 64-bit protection granule.

**RAS**

Reliability, Availability, Serviceability

**Recoverable error**

A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

**Reliability**

Continuity of correct service.

**Requester**

An agent in a computing system that initiates memory transactions.

**Restartable error**



A contained error that does not immediately impact correct operation. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part.

**SDC**

Silent Data Corruption

**SDC FIT rate**

The FIT rate for failures because of SDC.

**SDEC**

Single device error correction EDAC. This can detect and correct multiple clustered errors in a protection granule, such as the types of errors that might be seen if a protection granule is striped across multiple devices and multiple errors come from a single device.

**SECDED**

Single error correct, double error detect EDAC. This can detect a single or double bit error and correct a single bit error in a protection granule.

**SED**

Single error detect EDC. This can detect a single bit error in a protection granule.

**SError Interrupt**

An asynchronous interrupt in the Armv8 architecture.

**Service failure mode**

A mode entered to reduce the severity of an error.

**Serviceability**

The ability to undergo modifications and repairs.

**Silent Data Corruption**

An error that is not detected by hardware or software.

**Silently propagated**

An error that is passed from place to place without being signaled as a detected error.

**Software fault**

A fault that originates in and affects software.

**Synchronous exception**

In the Armv8 architecture, an exception for which all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

**Synchronous External Abort**

A synchronous exception in the Armv8 architecture.

**System Control Processor**

A PE dedicated to system control and monitoring.

**Transient fault**

A fault that is not persistent.

**Uncontained or uncontainable error**

An error that has been, or might have been, silently propagated.

**Undetected error or undetected fault**

See Latent error or latent fault.

**Unrecoverable error**

A contained error that is not recoverable. Continued correct operation is generally not possible. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part. Systems might use high-level recovery techniques to work around an unrecoverable yet contained error in a component so that the system recovers from the error.