Arm® Embedded Trace Router

Architecture Specification

arm

ARM IHI 0081A (ID071318)

Arm Embedded Trace Router
Architecture Specification

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
13 July 2018	A	Non-Confidential	First Release.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Product Status

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

Web Address

http://www.arm.com

Contents
Arm Embedded Trace Router Architecture Specification

Glossary

Preface

This preface introduces the *Embedded Trace Router (ETR) Architecture Specification*. It contains the following sections:

About this book

This book describes the architecture for the *Embedded Trace Router (ETR)*.

Intended audience

This document targets the following audiences:

- Designers of development tools supporting processor trace functionality.
- Advanced users of development tools supporting processor trace functionality.
- Designers of an Arm-based product that includes processor trace functionality.

Arm recommends that all users of this specification have experience of the Arm architecture.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to processor-originated trace, the Embedded Trace Router, and the overlap with the CoreSight Trace Memory Controller.

Chapter 2 *Specification*

Read this chapter for the Embedded Trace Router specification, register configurations for optional features, and operation modes.

Chapter 3 *Programmers' model*

Read this chapter for a description of the programmers' model for the Embedded Trace Router.

Appendix A *Pseudocode Definition*

Read this appendix for a description of the pseudocode that is used in this document.

Glossary Read this for definitions of some of the terms used in this book. The glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

Conventions

The following sections describe conventions that this document might use:

- *Typographic conventions*.
- *Signals*.
- *Numbers*.
- *Pseudocode descriptions*.

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

Colored text Indicates a link. This can be:

- A URL, for example `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Pseudocode descriptions*.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *JTAG Access Port (JTAG-AP)* or Debug Access Port (DAP).

Signals

The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

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`.

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, and is described in Appendix A *Pseudocode Definition*.

Additional reading

This section lists relevant publications from Arm and third parties.

See the Infocenter http://infocenter.arm.com, for access to Arm documentation.

Arm publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *CoreSight™ Trace Memory Controller* (ARM DDI 0461).
- *Arm® CoreSight™ Architecture Specification* (ARM DDI 0029).
- *Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual* (ARM 100806).
- *Arm® Embedded Trace Macrocell Architecture Specification ETMv4*.0to ETMv4.4 (ARM IHI 0064).

Other publications

The following books are referred to in this book, or provide more information:

- JEDEC, *Standard Manufacturers Identification Code,* JEP106 hhttp://www.jedec.org.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, ARM IHI 0081A.
- The page numbers to which your comments apply.
- 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

This chapter introduces the Embedded Trace Router. It contains the following sections:

1.1 About trace capture

In many CoreSight systems, trace is routed over a CoreSight trace fabric (AMBA ATB) through a series of trace funnels, and replicators, to one or more trace sinks.

One such trace sink is the CoreSight *Trace Memory Controller* (TMC). This can take various forms, one of which is an *Embedded Trace Router* (ETR). The function of the ETR is to write formatted trace to a buffer in memory. In effect, it is a bridge between the trace and memory fabrics.

This specification describes the ETR architecture, which is useful for both:

- Self-hosted trace, where trace is captured in system memory, for analysis by software running on the same system.
- External trace, where trace is captured in system memory, for subsequent export to an external debugger.

Table 1-1 Comparison of self-hosted and external trace

Term	Self-hosted trace	External trace
Definition	Located within the software stack. The agent controlling trace collection is part of the same software stack as the software being traced.	Located outside of the software stack. The agent controlling trace collection is not part of the same software stack as the software being traced.
Examples	**perf**: trace of applications and the operating system is collected by the operating system. **gdb**: the debugger is an application but uses operating system services to trace another application running under that operating system.	**Arm DS-5 Debugger**: Trace is collected on-chip and exported to a debugger on a different host for analysis. **Post-mortem**: Trace is collected by a system control processor to provide crash dump information.

1.1.1 Trace format

This specification does not define the trace format used. It is expected that it is used in conjunction with an *ETMv4 trace unit.* However, other options are permitted.

Note

If the trace protocol uses variable packet sizes, it must support an Alignment Synchronization mechanism to allow use in Circular Buffer mode.

The ETMv4 Ignore packet allows the ETM to pad trace in the trace buffer to an arbitrary alignment. For earlier versions of the ETMv4 protocol, a trace formatter must be implemented in the ETR.

1.1.2 Interrupts

An event is raised on a trigger or other buffer service event, such as an abort or the buffer filling.

For self-hosted trace, this is ideally a PPI interrupt. For external trace, this is either an SPI, LPI (MSI) or a cross-trigger event.

The ETR architecture also supports message-signaled interrupts, providing configuration registers to control these interrupts.

1.2 About the ETR architecture

The ETR architecture provides control mechanisms for storing trace data in a buffer in main system memory. Such a trace buffer can operate in one of the following modes:

Circular Buffer

In Circular Buffer mode, trace data is written to the buffer continuously. When the end of the buffer is reached, trace continues to be written from the start of the buffer, overwriting the oldest data in the buffer.

Circular Buffer mode is used to continuously capture trace. When the trace capture stops, the buffer contains the most recently generated trace, and provides a recent history of trace up to the stop point. Trace data can only be read out of the buffer when trace capture has stopped.

An ETR always implements the Circular Buffer mode.

Software Read FIFO Modes

In the Software Read FIFO modes, trace is written to the buffer continuously until the buffer is full. When the buffer is full, trace stops being written until space is freed in the buffer. Usually, a software agent concurrently reads trace data out of the buffer, and the memory buffer is used as an intermediate FIFO.

In Software Read FIFO mode 1, the trace data is read out of the buffer using the RRD in the ETR programmer's model.

In Software Read FIFO mode 2, the trace data is read directly out of the memory buffer, which is typically faster than Software Read FIFO mode 1. The software agent reading the buffer uses the RURP in the ETR programmers' model to indicate when data has been read out of the buffer.

The Software Read FIFO modes are optional.

1.2.1 Overlap with CoreSight Trace Memory Controller

The ETR architecture is based on the Embedded Trace Router configuration of the *CoreSight Trace Memory Controller (TMC)*, but makes the following major features optional:

- Scatter mode.
- Software Read FIFO mode 1.
- Buffer Watermark and Level registers.

An ETR without Scatter mode and without Software Read FIFO mode 1 means that the ETR does not need to read memory, which can simplify the ETR design.

An ETR without the buffer watermark feature can still generate interrupts, by initializing the RWP in a specific way.

Note

To generate an interrupt when the buffer is almost full (around 90%), software can:

- Set DBA to the base of the buffer and RSZ to the size of the buffer.
- Set RWP to DBA + RSZ × 10%.
- Set MODE.MODE to Circular Buffer.
- Configure ETR to generate an interrupt on FULL event.

Software written for the *CoreSight TMC* is not guaranteed to be compatible with an ETR compliant with this specification.

Software written for this specification is not guaranteed to be compatible with the *CoreSight TMC*.

Chapter 2
Specification

This chapter describes the Embedded Trace Router Specification. It contains the following sections:

2.1 Overview

I_{JDFA} The ETR architecture specifies the following features not defined in *CoreSight ETR*:

-
- Self-hosted support through interrupt generation.

R_{ASDF} The following ETR features are optional:

- Trace formatter.
- Software Read FIFO mode 1, including the buffer watermark and level registers.
- Software Read FIFO mode 2, including the buffer watermark and level registers.
- Self-hosted support through interrupt generation.
- Periodic synchronization counter (PSCR).
- Claim tags.
- CoreSight Software Lock.
- RAM Write Data register (RWD) and the Integration Test registers.

R_{IUYE} If the trace formatter is not implemented, the ETR does not support:

- Multiple trace sources.
- Embedded Flush.
- Embedded Triggers on flush.
- Embedded Triggers on Trigger Events.

2.2 Configurations

R_{AYIK} DEVID.MEMWIDTH specifies the IMPLEMENTATION DEFINED minimum alignment for the DBA, RRP, RURP, RWP, BUFWM and RSZ registers. Software must treat the corresponding low-order bits of these registers as SBZP:

- When software first writes to a pointer register, the value written must be aligned to a multiple of the size specified by DEVID.MEMWIDTH.
- When context switching the ETR context, software must preserve the low-order bits.

DBA, BUFWM and RSZ have a minimum architectural alignment of 32-bits. The BUFWM and RSZ registers always specify a number of 32-bit words.

R_{CXVZ} Circular Buffer mode must be implemented.

R_{ZXCA} It is IMPLEMENTATION DEFINED whether Software Read FIFO mode 1 is implemented.

R_{ZXCV} It is IMPLEMENTATION DEFINED whether Software Read FIFO mode 2 is implemented.

R_{IFHM} If both Software Read FIFO modes are not implemented, RRP, CBUFLEVEL, LBUFLEVEL and BUFWM are not implemented.

I_{NWUI} The mode is controlled by MODE.MODE. DEVID.MODES describes which modes are implemented.

I_{GHKL} A system including an ETR might support the input trace to the ETR being routed to trace sinks other than the ETR, for example via a AMBA ATB interface. Arm recommends that when CTL.TraceCaptEn is `0b1`, the routing of trace to other trace sinks does not limit the input trace bandwidth to the ETR.

2.3 Operation

I_{ASDF} The ETR state machine is defined by CTL.TraceCaptEn, STS.TMCReady and the state of the trace formatter (if implemented) and internal buffers:

Table 2-1 ETR state machine index

State	CTL.TraceCaptEn	STS.TMCReady	Formatter and buffer(s)
Disabled	0	1	Empty
Running	1	0	-
Stopping	1	0	Not empty
Stopped	1	1	Empty
Disabling	0	0	Not empty

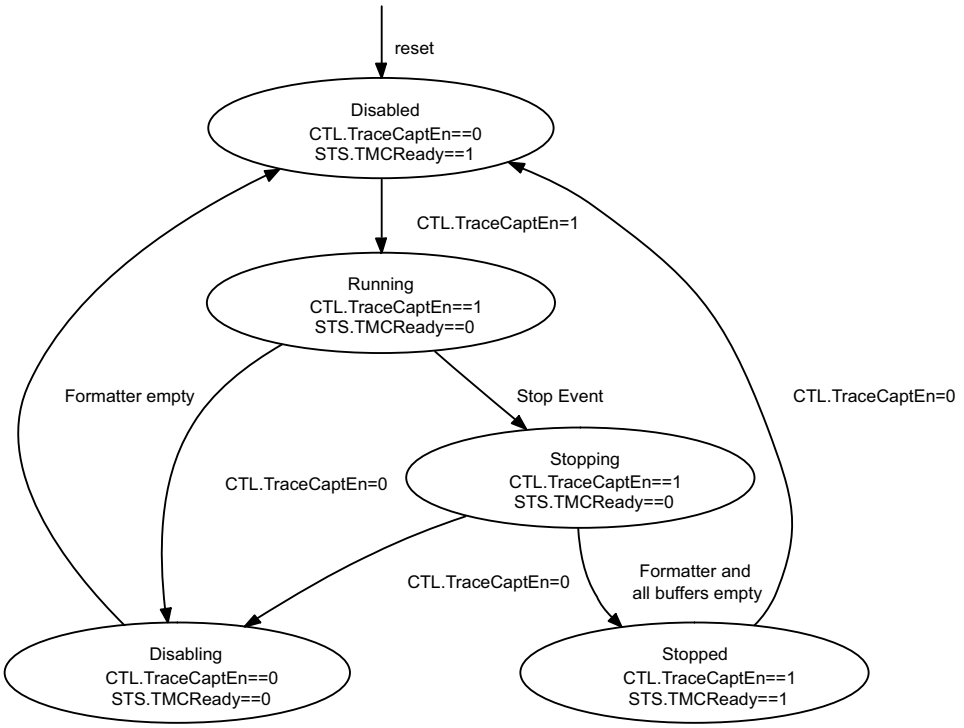


Figure 2-1 ETR state machine

I_{SDFV} When the ETR leaves the Disabled state, the ETR starts collecting trace to a buffer starting from the address in the RWP register.

Note

In *CoreSight Trace Memory Controller (TMC)*, when leaving the Disabled state, RWP is set to the value in DBA. The ETR architecture does not automatically set RWP when leaving the Disabled state. To ensure compatibility, software should not rely on the ETR resetting RWP to DBA

R_{SDFG} If the trace formatter is not implemented or disabled, when the ETR is in the Running state, each trace data byte received by the ETR is written to the address specified by RWP.

R_{XOVD} If the trace formatter is implemented and enabled, when the ETR is in the Running state, each trace byte received by the ETR is formatted. Each formatted byte is written to the address specified by RWP.

I_{DFAV} The trace formatter protocol is described by the *CoreSight Architecture Specification*.

R_{KPDF} For each byte written by the ETR, RWP is incremented by 1.

I_{MGHJ} The ETR might buffer multiple bytes into a single write, and increment RWP by the number of bytes written, so long as it does not write past the end of the buffer.

R_{BWUA} If Scatter mode is not implemented or is disabled, and the ETR increments RWP to (DBA + RSZ×4), then all of:

- RWP is set to DBA.
- If MODE.MODE == `0b00` (Circular Buffer) then STS.Full is set to 1.

R_{CVBN} If Scatter mode is not implemented or is disabled, and the ETR increments RRP to (DBA + RSZ × 4), then RRP is set to DBA.

R_{NLSF} If MODE.MODE != `0b00`, the buffer fill level is defined as:

- ((RWP - RRP) modulo (RSZ × 4)) when the trace buffer is not full.
- RSZ × 4 when the trace buffer is full.

R_{ALRU} If MODE.MODE != `0b00`, STS.Full is:

- Set to 1 when the buffer fill level is greater than or equal to (RSZ - BUFWM)×4.
- Set to 0 when the buffer fill level is less than (RSZ - BUFWM)×4.

I_{FGHB} In Software Read FIFO modes:

- The trace buffer is empty when RRP increments to RWP.
- The trace buffer is full when RWP increments to RRP.

R_{BYUU} In Circular Buffer mode, the ETR continues to write trace data to the trace buffer until a Stop Event occurs.

R_{FGJH} In Software FIFO modes, the ETR continues to write trace data to the trace buffer until a Stop Event occurs or the trace buffer becomes full. While the trace buffer is full, the ETR stalls the acceptance of new trace data until space becomes available in the trace buffer.

I_{VNWN} If the self-hosted interrupts are implemented, software configures which interrupts are generated using MODE.IRQOnReady and MODE.IRQOnFull.

R_{NZKO} If Software Read FIFO mode 2 is implemented, the buffer interrupt for Software Read FIFO mode 2 is enabled whenever Software Read FIFO mode 2 is selected and the ETR is enabled.

R_{YDHF} In Circular Buffer mode and if RRP is implemented, if RWP increments to RRP then any subsequent increment to RWP also increments RRP to the new value of RWP. This tracking continues until the ETR stops or is disabled.

R_{LPZF} If the trace formatter is implemented and enabled when trace capture is stopped, then the traces are padded in the formatted frames with additional bytes of data with a value of `0x00` and an ID of `0x00`, until the following conditions are met:

- A whole number of frames have been generated.
- RWP is aligned to a multiple of DEVID.MEMWIDTH.

R_{NQWE} If the trace formatter is not implemented or is disabled when trace capture is stopped, then it is IMPLEMENTATION DEFINED whether the ETR writes the stop sequence to the end of the trace buffer. The stop sequence consists of:

- A single byte of value `0x01`, to indicate the position of the last byte before the stop sequence.
- Zero or more bytes of `0x00`, to align RWP to a multiple of DEVID.MEMWIDTH.

I_{MLIU} If the trace formatter is not implemented and the ETR is tightly-coupled to a trace source, Arm recommends that the trace source generates sufficient padding bytes in its own trace protocol to align RWP before trace capture is stopped, and the ETR does not generate the stop sequence.

R_{OUYG} When the ETR is in the Running state, on a Stop Event the ETR enters the Stopping state.

R_{LQWE} A Stop Event occurs when any of the following occur:

- A Flush Completion occurs when FFCR.StopOnFl is `0b1`.
- A Trigger Event occurs when FFCR.StopOnTrigEvt is `0b1`.

R_{HDGP} The ETR moves from Stopping state to Stopped state when all trace which was received by the ETR before entering Stopping state has been output to the trace buffer, including any stop sequence or padding frames.

R_{MFGS} In the Stopped state, the ETR moves to the Disabled state when CTL.TraceCaptEn is changed from `0b1` to `0b0`.

I_{OUTJ} In Software Read FIFO modes, the ETR might not move from the Stopping state to the Stopped state if there is insufficient space to insert the remaining trace data in the trace buffer. Reads of the trace buffer via the RRD or updates using the RURP might be needed before the ETR enters the Stopped state.

R_{PLOK} If CTL.TraceCaptEn is changed from `0b1` to `0b0` when not in the Stopped state, the trace captured in the trace buffer might be incomplete and the ETR pointers might not reflect the correct position of the trace buffer.

R_{ZEWQ} In the Stopping, Stopped, Disabling, and Disabled states, no new trace is captured by the ETR and:

- If MODE.StallOnStop is `0b0`, the ETR discards any new trace received.
- If MODE.StallOnStop is `0b1`, the ETR does not discard any new trace received.

2.4 Trigger and Flush Operation

I_{BUIR} The ETR architecture supports detection of a trigger condition. A trigger condition is typically used to stop trace capture to ensure trace is captured around a point of interest.

A Detected Trigger is where the ETR detects a trigger condition from an external source, such as a trace source or a Cross Trigger Interface. A Detected Trigger consists of one or more of the following:

- An external Trigger Input.
- A Trigger ID indicated with the trace from the trace source(s).

An Embedded Trigger is where the ETR inserts a special trace source ID into a formatted trace stream. Embedded Triggers are not supported if the trace formatter is not implemented.

A Trigger Event occurs when the Trigger Counter has counted the specified number of trace bytes after a Detected Trigger.

The Trigger Counter is a counter used to delay a Trigger Event for a specified number of trace bytes after a Detected Trigger.

I_{CHSD} The ETR architecture supports flushing of the ETR and any trace sources connected to the ETR. Flushing involves requesting all outstanding data from any upstream trace sources, and ensuring all of the flushed data is inserted into the trace buffer. Typically, a flush is used when stopping trace capture to ensure all outstanding data has been captured.

A Detected Flush is where the ETR detects a request for a flush, such as via the ETR programmers' model or a Cross Trigger Interface. A Detected Flush consists of one or more of the following:

- An external Flush Input when FFCR.FOnFlIn is `0b1`.
- A Manual Flush initiated via FFCR.FlushMan.
- When a Trigger Event occurs FFCR.FOnTrigEvt is `0b1`.

An Embedded Flush is where the ETR inserts a special trace source ID into a formatted trace stream. Embedded Flushes are not supported if the trace formatter is not implemented.

A Flush Completion occurs when the ETR has ensured that all outstanding data has been captured in the trace buffer.

R_{ZXMN} When the first Detected Trigger occurs, the Trigger Counter starts counting the number of bytes written to the trace buffer.

R_{KLFG} When the Trigger Counter has counted the number of bytes programmed into the TRG register, a Trigger Event occurs.

Note

The TRG register specifies a number of 32-bit words.

R_{VMCB} When the trace formatter is implemented and enabled, an Embedded Trigger is inserted into the formatted trace stream when any of the following occur:

- FFCR.TrigOnTrigEvt is `0b1` and a Trigger Event occurs.
- FFCR.TrigOnTrigIn is `0b1` and a Detected Trigger occurs via an external Trigger Input.
- FFCR.TrigOnFl is `0b1` and a Flush Completion occurs.

R_{PWTY} An Embedded Trigger consists of a single zero data byte with a trace ID of `0x7D`.

R_{HGFD} An Embedded Flush consists of a single zero data byte with a trace ID of `0x7B`.

R_{VORH} A Detected Flush occurs when any of the following occur:

- The external Flush Input is asserted and FFCR.FOnFlIn is `0b1`.
- FFCR.FlushMan is written with `0b1`.
- A Trigger Event occurs and FFCR.FOnTrigEvt is `0b1`.

R_{LDFG} When a Detected Flush occurs, the ETR performs the following sequence:

1. Requests all connected trace sources to output all data that has been accrued.
2. Outputs all the data received from the trace sources to the trace buffer.
3. Indicates a Flush Completion.

R_{SGCV} When the trace formatter is implemented and enabled, an Embedded Flush is inserted into the formatted trace stream when a Flush Completion occurs and FFCR.EmbedFlush is `0b1`. The Embedded Flush is inserted after the last byte of trace data and before the end of the padding data.

2.5 Scatter mode

I_{ASDF} Scatter mode is used in *CoreSight Trace Memory Controller (TMC)* to distribute the trace buffer in non-contiguous memory pages.

R_{IJKS} Scatter mode is not implemented, and DEVID.NOSCAT is `0b1`.

Chapter 3
Programmers' model

This chapter describes the Embedded Trace Router programmers' model. It contains the following sections:

3.1 Memory-mapped registers

Table 3-1 shows the register index for the Embedded Trace Router programmers' model.

Table 3-1 Index of ETR registers ordered by offset

Register	Offset	Length	Description, see
RSZ	0x004	32	*RSZ, RAM Size Register* on page 3-94
STS	0x00C	32	*STS, Status Register* on page 3-98
RRD	0x010	32	*RRD, RAM Read Data register* on page 3-92
RRP	0x014 0x038	64	*RRP, RAM Read Pointer* on page 3-93
RWP	0x018 0x03C	64	*RWP, RAM Write Pointer* on page 3-97
TRG	0x01C	32	*TRG, Trigger Counter Register* on page 3-102
CTL	0x020	32	*CTL, Control Register* on page 3-52
RWD	0x024	32	*RWD, RAM Write Data register* on page 3-96
MODE	0x028	32	*MODE, Mode Register* on page 3-79
LBUFLEVEL	0x02C	32	*LBUFLEVEL, Latched Buffer Fill Level* on page 3-76
CBUFLEVEL	0x030	32	*CBUFLEVEL, Current Buffer Fill Level* on page 3-45
BUFWM	0x034	32	*BUFWM, Buffer Watermark register* on page 3-33
BUSCTL	0x110	32	*BUSCTL, Bus Control Register* on page 3-34
DBA	0x118	64	*DBA, Data Buffer Address* on page 3-53
RURP	0x120	32	*RURP, RAM Update Read Pointer* on page 3-95
FFSR	0x300	32	*FFSR, Formatter and Flush Status Register* on page 3-68
FFCR	0x304	32	*FFCR, Formatter and Flush Control Register* on page 3-63
PSCR	0x308	32	*PSCR, Periodic Synchronization Control Register* on page 3-90
IRQCR0	0xE80	64	*IRQCR0, Interrupt Configuration Register* on page 3-70
IRQCR1	0xE88	32	*IRQCR1, Interrupt Configuration Register 1* on page 3-71
IRQCR2	0xE8C	32	*IRQCR2, Interrupt Configuration Register 2* on page 3-72
ITCTRL	0xF00	32	*ITCTRL, Integration Mode Control Register* on page 3-74
CLAIMSET	0xFA0	32	*CLAIMSET, Claim Tag Set Register* on page 3-51
CLAIMCLR	0xFA4	32	*CLAIMCLR, Claim Tag Clear Register* on page 3-50
LAR	0xFB0	32	*LAR, Lock Access Register* on page 3-75
LSR	0xFB4	32	*LSR, Lock Status Register* on page 3-77
AUTHSTATUS	0xFB8	32	*AUTHSTATUS, Authentication Status Register* on page 3-31
DEVARCH	0xFBC	32	*DEVARCH, Device Architecture Register* on page 3-54
DEVID1	0xFC4	32	*DEVID1, Device Configuration Register 1* on page 3-61

Table 3-1 Index of ETR registers ordered by offset (continued)

Register	Offset	Length	Description, see
DEVID	0xFC8	32	*DEVID, Device Configuration Register* on page 3-56
DEVTYPE	0xFCC	32	*DEVTYPE, CoreSight Device Type Register* on page 3-62
PIDR4	0xFD0	32	*PIDR4, Peripheral Identification Register* on page 3-86
PIDR5	0xFD4	32	*PIDR5, Peripheral Identification Register* on page 3-87
PIDR6	0xFD8	32	*PIDR6, Peripheral Identification Register* on page 3-88
PIDR7	0xFDC	32	*PIDR7, Peripheral Identification Register* on page 3-89
PIDR0	0xFE0	32	*PIDR0, Peripheral Identification Register* on page 3-82
PIDR1	0xFE4	32	*PIDR1, Peripheral Identification Register* on page 3-83
PIDR2	0xFE8	32	*PIDR2, Peripheral Identification Register* on page 3-84
PIDR3	0xFEC	32	*PIDR3, Peripheral Identification Register* on page 3-85
CIDR0	0xFF0	32	*CIDR0, Component Identification Register 0* on page 3-46
CIDR1	0xFF4	32	*CIDR1, Component Identification Register 1* on page 3-47
CIDR2	0xFF8	32	*CIDR2, Component Identification Register 2* on page 3-48
CIDR3	0xFFC	32	*CIDR3, Component Identification Register 3* on page 3-49

Table 3-2 shows the reset values for the Embedded Trace Router programmers' model.

Table 3-2 ETR register field reset values

Register field	Reset value
BUSCTL.SORGN	0
BUSCTL.SIRGN	0
BUSCTL.BAttr	0
CLAIMCLR.CLR<m>	0
CTL.TraceCaptEn	0
FFCR.EmbedFlush	0
FFCR.StopOnTrigEvt	0
FFCR.StopOnFl	0
FFCR.TrigOnFl	0
FFCR.TrigOnTrigEvt	0
FFCR.TrigOnTrigIn	0
FFCR.FlushMan	0
FFCR.FOnTrigEvt	0
FFCR.FOnFlIn	0

Table 3-2 ETR register field reset values (continued)

Register field	Reset value
FFCR.EnFmt	0
FFCR.StopOnTrigEvt	0
FFCR.StopOnFI	0
FFCR.FlushMan	0
FFCR.FOnTrigEvt	0
FFCR.FOnFIIn	0
ITCTRL.IME	0
PSCR.PSCount	10
STS.TMCReady	1

3.1.1 AUTHSTATUS, Authentication Status Register

The AUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Usage constraints

There are no usage constraints.

Configurations

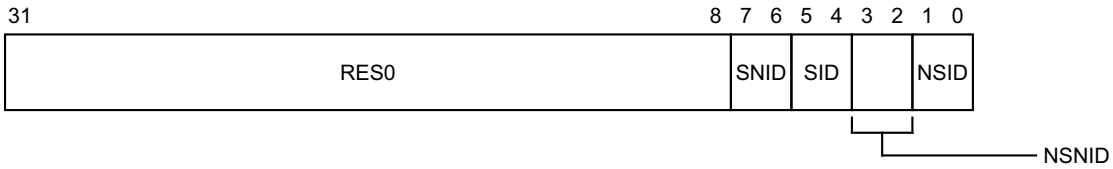
There are no configuration notes.

Attributes

AUTHSTATUS is a 32-bit register.

Field descriptions

The AUTHSTATUS bit assignments are:



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug is enabled. The defined values of this field are:

`0b00` Secure non-invasive debug not implemented.

This field reads as `0b00`.

SID, bits [5:4]

Secure Invasive Debug. Indicates whether Secure invasive debug is enabled. The defined values of this field are:

`0b00` Secure invasive debug not implemented.

`0b10` Secure invasive debug disabled.

`0b11` Secure invasive debug enabled.

NSNID, bits [3:2]

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug is enabled. The defined values of this field are:

`0b00` Non-secure non-invasive debug not implemented.

This field reads as `0b00`.

NSID, bits [1:0]

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug is enabled. The defined values of this field are:

`0b00` Non-secure invasive debug not implemented.

`0b10` Non-secure invasive debug disabled.

`0b11` Non-secure invasive debug enabled.

Arm recommends that Non-secure invasive debug is always enabled at the ETR.

Accessing the AUTHSTATUS:

AUTHSTATUS can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFB8`

This interface is accessible as follows:

- Access to this register is RO.

3.1.2 BUFWM, Buffer Watermark register

The BUFWM characteristics are:

Purpose

Provides threshold vacancy level in 32-bit words in the trace memory.

Usage constraints

Writes to BUFWM are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

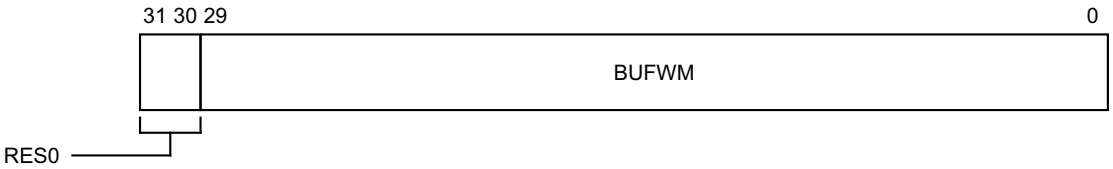
This register is present only when DEVID.MODES != 0b010. Otherwise, direct accesses to BUFWM are RES0.

Attributes

BUFWM is a 32-bit register.

Field descriptions

The BUFWM bit assignments are:



Bits [31:30]

Reserved, RES0.

BUFWM, bits [29:0]

Buffer Watermark. Indicates the desired threshold vacancy level in the trace memory in 32-bit words. When the available space in the trace memory is less than or equal to this value, STS.Full is set to 1.

Software must set this field to a multiple of the minimum alignment defined by DEVID.MEMWIDTH divided by 4.

This field is ignored by the ETR when MODE.MODE == `0b00`, meaning Circular Buffer mode is selected.

Accessing the BUFWM:

BUFWM can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x034`

This interface is accessible as follows:

- Access to this register is RW.

3.1.3 BUSCTL, Bus Control Register

The BUSCTL characteristics are:

Purpose

Controls ETR accesses to system memory.

Usage constraints

If any field in this register is set to a reserved value:

- The behavior is as if the field is set to an UNKNOWN implemented value.
- The ETR must not generate invalid bus transactions.
- This might lead to a loss of coherency.

Writes to BUSCTL are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

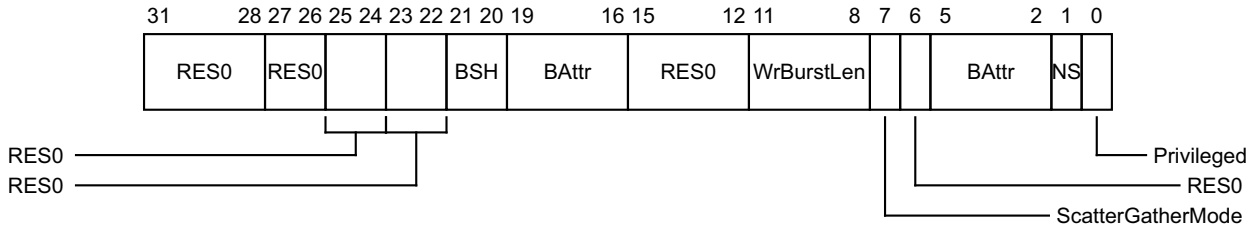
Some or all RW fields of this register have defined reset values.

Attributes

BUSCTL is a 32-bit register.

Field descriptions

The BUSCTL bit assignments are:



Bits [31:28]

Reserved, RES0.

Bits [27:26]

Reserved, RES0.

Bits [25:24]

Reserved, RES0.

Bits [23:22]

Reserved, RES0.

BSH, bits [21:20]

Buffer shareability domain. Defines the shareability domain for Normal memory used by the buffer. The possible values of this field are:

`0b00` Not shared.

`0b10` Outer shareable.

`0b11` Inner shareable.

All other values are reserved.

BAttr, bits [19:16]

When DEVID.CACHETYPE == 0b0x:

BAttr is a split field that occupies BUSCTL[19:16, 5:2].

Memory type. Defines the type of memory addressed by the buffer when an IMPLEMENTATION DEFINED bus interface, such as AXI, is implemented. The definition of this field is IMPLEMENTATION DEFINED. Recommended values for an AXI bus and a fully coherent system are defined. If DEVID.CACHETYPE == `0b00` and DEVID.NOSCAT == `0b0`, then this field applies also to the scatter table. When AMBA AXI3 bus, the possible values of BAttr[7:0] are:

BAttr	Meaning
`0b00000000`	Non-cacheable , Non-bufferable.
`0b00000001`	Non-cacheable , Bufferable.
`0b00000010`	Normal Memory, Cacheable, No-allocate.
`0b00000011`	Normal Memory, Cacheable and Bufferable, No-allocate.
`0b00000110`	Normal Memory, Write-through cacheable, Read-allocate.
`0b00000111`	Normal Memory, Write-back cacheable, Read-allocate.
`0b00001010`	Normal Memory, Write-through cacheable, Write-allocate.
`0b00001011`	Normal Memory, Write-back cacheable, Write-allocate.
`0b00001110`	Normal Memory, Write-through cacheable, Read and Write-allocate.
`0b00001111`	Normal Memory, Write-back cacheable, Read and Write-allocate.

All other values are reserved.

When AMBA AXI4 bus, if Software Read FIFO and Scatter modes are not implemented, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
`0b00000000`	Device-nGnRnE.
`0b00000001`	Device-nGnRE.
`0b00000010`	Normal Memory, Non-cacheable, Non-bufferable.
`0b00000011`	Normal Memory, Non-Cacheable, Bufferable.
`0b00000110`	Normal Memory, Write-through cacheable, No-allocate.
`0b00000111`	Normal Memory, Write-back cacheable, No-allocate.
`0b00001110`	Normal Memory, Write-through cacheable, Write-allocate.
`0b00001111`	Normal Memory, Write-back cacheable, Write-allocate.

All other values are reserved.

When AMBA AXI4 bus, if Software Read FIFO or Scatter modes are implemented, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Device-nGnRnE.
0b00010001	Device-nGnRE.
0b00100010	Normal Memory, Non-cacheable, Non-bufferable.
0b00110011	Normal Memory, Non-Cacheable, Bufferable.
0b10100110	Normal Memory, Write-through cacheable, No-allocate.
0b10101110	Normal Memory, Write-through cacheable, Write-allocate.
0b10110111	Normal Memory, Write-back cacheable, No-allocate.
0b10111111	Normal Memory, Write-back cacheable, Write-allocate.
0b11100110	Normal Memory, Write-through cacheable, Read-allocate.
0b11101110	Normal Memory, Write-through cacheable, Read and Write-allocate.
0b11110111	Normal Memory, Write-back cacheable, Read-allocate.
0b11111111	Normal Memory, Write-back cacheable, Read and Write-allocate.

All other values are reserved.

When Fully coherent system, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Coherent.

All other values are reserved.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

Note

The recommended AXI bus values match the ARCACHE and AWCACHE encodings on AXI.

On a reset, this field resets to 0.

When DEVID.CACHETYPE == 0b1x:

BAttr is a split field that occupies BUSCTL[19:16, 5:2].

BAttr[7:4] Specifies the outer buffer memory type. Defines the Outer attributes for memory addressed by the buffer. The possible values of BAttr[7:4] are:

BAttr[7:4]	Meaning
0b0000	Device memory. The device memory type is defined by BAttr[3:0].
0b0001	Normal memory, Outer write-through cacheable, transient Write allocate.
0b0010	Normal memory, Outer write-through cacheable, transient Read allocate.
0b0011	Normal memory, Outer write-through cacheable, transient Read and Write allocate.

BAttr[7:4]	Meaning
0b0100	Normal memory, Outer non-cacheable.
0b0101	Normal memory, Outer write-back cacheable, transient Write allocate.
0b0110	Normal memory, Outer write-back cacheable, transient Read allocate.
0b0111	Normal memory, Outer write-back cacheable, transient Read and Write allocate.
0b1000	Normal memory, Outer write-through cacheable, No allocate.
0b1001	Normal memory, Outer write-through cacheable, Write allocate.
0b1010	Normal memory, Outer write-through cacheable, Read allocate.
0b1011	Normal memory, Outer write-through cacheable, Read and Write allocate.
0b1100	Normal memory, Outer write-back cacheable, No allocate.
0b1101	Normal memory, Outer write-back cacheable, Write allocate.
0b1110	Normal memory, Outer write-back cacheable, Read allocate.
0b1111	Normal memory, Outer write-back cacheable, Read and Write allocate.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

This field resets to 0x0.

Note

The generic encodings match the ARMv8 MAIR register values.

BAttr[3:0] specifies the inner buffer memory type. Defines the Inner attributes for memory addressed by the buffer.

When BAttr[7:4] != 0b0000 and separate Inner attributes are not supported, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0000	Should be zero.

All other values are reserved.

When BAttr[7:4] != 0b0000 and separate Inner attributes are supported, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0001	Normal memory, Inner write-through cacheable, transient Write allocate.
0b0010	Normal memory, Inner write-through cacheable, transient Read allocate.
0b0011	Normal memory, Inner write-through cacheable, transient Read and Write allocate.
0b0100	Normal memory, Inner non-cacheable.
0b0101	Normal memory, Inner write-back cacheable, transient Write allocate.
0b0110	Normal memory, Inner write-back cacheable, transient Read allocate.

BAttr[3:0]	Meaning
0b0111	Normal memory, Inner write-back cacheable, transient Read and Write allocate.
0b1000	Normal memory, Inner write-through cacheable, No allocate.
0b1001	Normal memory, Inner write-through cacheable, Write allocate.
0b1010	Normal memory, Inner write-through cacheable, Read allocate.
0b1011	Normal memory, Inner write-through cacheable, Read and Write allocate.
0b1100	Normal memory, Inner write-back cacheable, No allocate.
0b1101	Normal memory, Inner write-back cacheable, Write allocate.
0b1110	Normal memory, Inner write-back cacheable, Read allocate.
0b1111	Normal memory, Inner write-back cacheable, Read and Write allocate.

All other values are reserved.

When BAttr[7:4] == 0b0000, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0000	Device-nGnRnE.
0b0100	Device-nGnRE.
0b1000	Device-nGRE.
0b1100	Device-GRE.

All other values are reserved.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

This field resets to 0x0.

Note

The generic encodings match the ARMv8 MAIR register values.

Otherwise:

Reserved, RES0.

Bits [15:12]

Reserved, RES0.

WrBurstLen, bits [11:8]

Write Burst Length.

Controls the maximum number of data transfers that can occur within each burst initiated by the ETR on the bus master interface. This field is a hint to the ETR.

Arm recommends that this value is set to no more than half the write buffer depth. Arm also recommends that this value is set to enable a burst of at least one frame of trace data.

The possible values of this field are:

0b0000 One data transfer per burst. This is the default.

0b0001 Maximum of two data transfers per burst.

0b0010 Maximum of three data transfers per burst.

0b0011 Maximum of four data transfers per burst.

0b0100 Maximum of five data transfers per burst.

0b0101 Maximum of six data transfers per burst.

0b0110 Maximum of seven data transfers per burst.

0b0111 Maximum of eight data transfers per burst.

0b1000 Maximum of nine data transfers per burst.

0b1001 Maximum of ten data transfers per burst.

0b1010 Maximum of eleven data transfers per burst.

0b1011 Maximum of twelve data transfers per burst.

0b1100 Maximum of thirteen data transfers per burst.

0b1101 Maximum of fourteen data transfers per burst.

0b1110 Maximum of fifteen data transfers per burst.

0b1111 No maximum.

Note

In the CoreSight TMC programming an incompatible burst length results in UNPREDICTABLE behavior. In the ETR architecture, all values must give predictable behavior as the interpretation is IMPLEMENTATION DEFINED.

ScatterGatherMode, bit [7]

When DEVID.NOSCAT == 0b0:

Scatter mode. Controls whether trace memory is accessed as a single buffer in system memory or through a scatter linked-list memory. The possible values of this bit are:

0b0 Trace memory is a single contiguous block of system memory.

0b1 Trace memory is spread over multiple blocks of system memory using a linked-list.

This bit is ignored when in Disabled state.

Otherwise:

Reserved, RES0.

Bit [6]

Reserved, RES0.

BAttr, bits [5:2]

When DEVID.CACHETYPE == 0b0x:

BAttr is a split field that occupies BUSCTL[19:16, 5:2].

Memory type. Defines the type of memory addressed by the buffer when an IMPLEMENTATION DEFINED bus interface, such as AXI, is implemented. The definition of this field is IMPLEMENTATION DEFINED. Recommended values for an AXI bus and a fully coherent system are defined. If DEVID.CACHETYPE == 0b00 and DEVID.NOSCAT == 0b0, then this field applies also to the scatter table. When AMBA AXI3 bus, the possible values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Non-cacheable , Non-bufferable.
0b00000001	Non-cacheable , Bufferable.
0b00000010	Normal Memory, Cacheable, No-allocate.

BAttr	Meaning
0b00000011	Normal Memory, Cacheable and Bufferable, No-allocate.
0b00000110	Normal Memory, Write-through cacheable, Read-allocate.
0b00000111	Normal Memory, Write-back cacheable, Read-allocate.
0b00001010	Normal Memory, Write-through cacheable, Write-allocate.
0b00001011	Normal Memory, Write-back cacheable, Write-allocate.
0b00001110	Normal Memory, Write-through cacheable, Read and Write-allocate.
0b00001111	Normal Memory, Write-back cacheable, Read and Write-allocate.

All other values are reserved.

When AMBA AXI4 bus, if Software Read FIFO and Scatter modes are not implemented, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Device-nGnRnE.
0b00000001	Device-nGnRE.
0b00000010	Normal Memory, Non-cacheable, Non-bufferable.
0b00000011	Normal Memory, Non-Cacheable, Bufferable.
0b00000110	Normal Memory, Write-through cacheable, No-allocate.
0b00000111	Normal Memory, Write-back cacheable, No-allocate.
0b00001110	Normal Memory, Write-through cacheable, Write-allocate.
0b00001111	Normal Memory, Write-back cacheable, Write-allocate.

All other values are reserved.

When AMBA AXI4 bus, if Software Read FIFO or Scatter modes are implemented, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Device-nGnRnE.
0b00010001	Device-nGnRE.
0b00100010	Normal Memory, Non-cacheable, Non-bufferable.
0b00110011	Normal Memory, Non-Cacheable, Bufferable.
0b10100110	Normal Memory, Write-through cacheable, No-allocate.
0b10101110	Normal Memory, Write-through cacheable, Write-allocate.
0b10110111	Normal Memory, Write-back cacheable, No-allocate.
0b10111111	Normal Memory, Write-back cacheable, Write-allocate.
0b11100110	Normal Memory, Write-through cacheable, Read-allocate.

BAttr	Meaning
0b11101110	Normal Memory, Write-through cacheable, Read and Write-allocate.
0b11110111	Normal Memory, Write-back cacheable, Read-allocate.
0b11111111	Normal Memory, Write-back cacheable, Read and Write-allocate.

All other values are reserved.

When Fully coherent system, the possible values of values of BAttr[7:0] are:

BAttr	Meaning
0b00000000	Coherent.

All other values are reserved.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

Note

The recommended AXI bus values match the ARCACHE and AWCACHE encodings on AXI.

On a reset, this field resets to 0.

When DEVID.CACHETYPE == 0b1x:

BAttr is a split field that occupies BUSCTL[19:16, 5:2].

BAttr[7:4] Specifies the outer buffer memory type. Defines the Outer attributes for memory addressed by the buffer. The possible values of BAttr[7:4] are:

BAttr[7:4]	Meaning
0b0000	Device memory. The device memory type is defined by BAttr[3:0].
0b0001	Normal memory, Outer write-through cacheable, transient Write allocate.
0b0010	Normal memory, Outer write-through cacheable, transient Read allocate.
0b0011	Normal memory, Outer write-through cacheable, transient Read and Write allocate.
0b0100	Normal memory, Outer non-cacheable.
0b0101	Normal memory, Outer write-back cacheable, transient Write allocate.
0b0110	Normal memory, Outer write-back cacheable, transient Read allocate.
0b0111	Normal memory, Outer write-back cacheable, transient Read and Write allocate.
0b1000	Normal memory, Outer write-through cacheable, No allocate.
0b1001	Normal memory, Outer write-through cacheable, Write allocate.
0b1010	Normal memory, Outer write-through cacheable, Read allocate.
0b1011	Normal memory, Outer write-through cacheable, Read and Write allocate.
0b1100	Normal memory, Outer write-back cacheable, No allocate.

BAttr[7:4]	Meaning
0b1101	Normal memory, Outer write-back cacheable, Write allocate.
0b1110	Normal memory, Outer write-back cacheable, Read allocate.
0b1111	Normal memory, Outer write-back cacheable, Read and Write allocate.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

This field resets to 0x0.

Note

The generic encodings match the ARMv8 MAIR register values.

BAttr[3:0] specifies the inner buffer memory type. Defines the Inner attributes for memory addressed by the buffer.

When BAttr[7:4] != 0b0000 and separate Inner attributes are not supported, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0000	Should be zero.

All other values are reserved.

When BAttr[7:4] != 0b0000 and separate Inner attributes are supported, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0001	Normal memory, Inner write-through cacheable, transient Write allocate.
0b0010	Normal memory, Inner write-through cacheable, transient Read allocate.
0b0011	Normal memory, Inner write-through cacheable, transient Read and Write allocate.
0b0100	Normal memory, Inner non-cacheable.
0b0101	Normal memory, Inner write-back cacheable, transient Write allocate.
0b0110	Normal memory, Inner write-back cacheable, transient Read allocate.
0b0111	Normal memory, Inner write-back cacheable, transient Read and Write allocate.
0b1000	Normal memory, Inner write-through cacheable, No allocate.
0b1001	Normal memory, Inner write-through cacheable, Write allocate.
0b1010	Normal memory, Inner write-through cacheable, Read allocate.
0b1011	Normal memory, Inner write-through cacheable, Read and Write allocate.
0b1100	Normal memory, Inner write-back cacheable, No allocate.
0b1101	Normal memory, Inner write-back cacheable, Write allocate.
0b1110	Normal memory, Inner write-back cacheable, Read allocate.
0b1111	Normal memory, Inner write-back cacheable, Read and Write allocate.

All other values are reserved.

When BAttr[7:4] == 0b0000, the possible values of BAttr[3:0] are:

BAttr[3:0]	Meaning
0b0000	Device-nGnRnE.
0b0100	Device-nGnRE.
0b1000	Device-nGRE.
0b1100	Device-GRE.

All other values are reserved.

If this field is set to a type that this implementation does not support, the behavior is as if set to a compatible type and this field reads as an UNKNOWN value.

This field resets to 0x0.

Note

The generic encodings match the ARMv8 MAIR register values.

Otherwise:

Reserved, RES0.

NS, bit [1]

Secure memory attributes. Defines the Secure attribute for memory addressed by the buffer. The possible values of this bit are:

0b0 Secure.

0b1 Non-secure.

If the system implements both Secure and Non-secure memory, it is IMPLEMENTATION DEFINED which of the following is true:

- This field is RES1, and the ETR can only write to Non-secure memory.
- This field is RES0, and the ETR can only write to Secure memory.
- This field is RW, and the ETR can write to either Secure or Non-secure memory based on the value of this field.

If the system implements only Secure memory, it is IMPLEMENTATION DEFINED which of the following is true:

- This field is RES0, and the ETR can only write to Secure memory.
- This field is RW, but if the field is set to 1 it is UNKNOWN which of the following occurs:
  - The ETR ignores this bit.
  - The ETR sets STS.MemErr when a memory access is performed, and does not perform the access
  - The memory system returns an error response to the ETR when a memory access is performed.

If the system implements only Non-secure memory, it is IMPLEMENTATION DEFINED which of the following is true:

- This field is RES1, and the ETR can only write to Non-secure memory.
- This field is RW, but if the field is set to 0b0 it is UNKNOWN which of the following occurs:
  - The ETR ignores this bit.

— The ETR sets STS.MemErr when a memory access is performed, and does not perform the access.

— The memory system returns an error response to the ETR when a memory access is performed.

Privileged, bit [0]

Privileged memory attributes. Defines the Privileged attribute for memory addressed by the buffer. The possible values of this bit are:

`0b0` Unprivileged.

`0b1` Privileged.

This bit is RES1 if this bit is not supported.

Accessing the BUSCTL:

BUSCTL can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x110`

This interface is accessible as follows:

- Access to this register is RW.

3.1.4 CBUFLEVEL, Current Buffer Fill Level

The CBUFLEVEL characteristics are:

Purpose

The current fill level of the trace memory in 32-bit words.

Usage constraints

The value of this register is UNKNOWN when CTL.TraceCaptEn == 0

Configurations

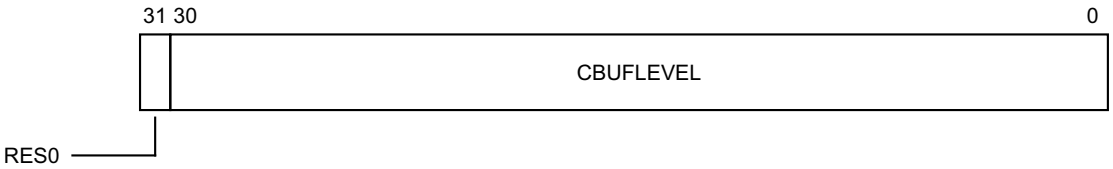
This register is present only when DEVID.MODES != 0b010. Otherwise, direct accesses to CBUFLEVEL are RES0.

Attributes

CBUFLEVEL is a 32-bit register.

Field descriptions

The CBUFLEVEL bit assignments are:



Bit [31]

Reserved, RES0.

CBUFLEVEL, bits [30:0]

Current Buffer Fill Level. The current fill level of the trace memory in 32-bit words. This field is not valid and reads UNKNOWN if any of the following are true:

- MODE.MODE == 0b00, meaning Circular Buffer mode is selected
- CTL.TraceCaptEn == 0, meaning Trace capture is disabled

Accessing the CBUFLEVEL:

CBUFLEVEL can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x030

This interface is accessible as follows:

- Access to this register is RO.

3.1.5 CIDR0, Component Identification Register 0

The CIDR0 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

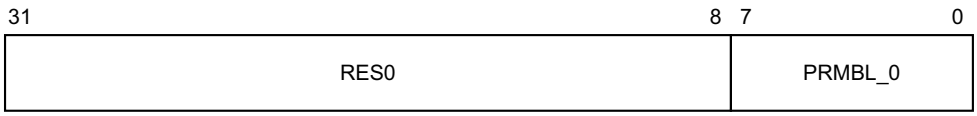
There are no configuration notes.

Attributes

CIDR0 is a 32-bit register.

Field descriptions

The CIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See CoreSight Architecture Specification.

This field reads as `0x0D`

Accessing the CIDR0:

CIDR0 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFF0`

This interface is accessible as follows:

- Access to this register is RO.

3.1.6 CIDR1, Component Identification Register 1

The CIDR1 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

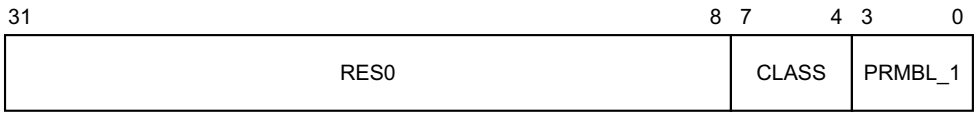
There are no configuration notes.

Attributes

CIDR1 is a 32-bit register.

Field descriptions

The CIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

CoreSight component class. See CoreSight Architecture Specification.

This field reads as `0x9`.

PRMBL_1, bits [3:0]

CoreSight component identification preamble. See CoreSight Architecture Specification.

This field reads as `0x0`.

Accessing the CIDR1:

CIDR1 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFF4`

This interface is accessible as follows:

- Access to this register is RO.

3.1.7 CIDR2, Component Identification Register 2

The CIDR2 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

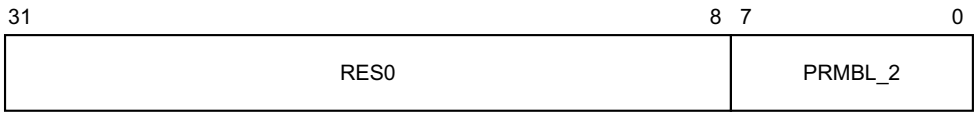
There are no configuration notes.

Attributes

CIDR2 is a 32-bit register.

Field descriptions

The CIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See CoreSight Architecture Specification.

This field reads as `0x05`.

Accessing the CIDR2:

CIDR2 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFF8`

This interface is accessible as follows:

- Access to this register is RO.

3.1.8 CIDR3, Component Identification Register 3

The CIDR3 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

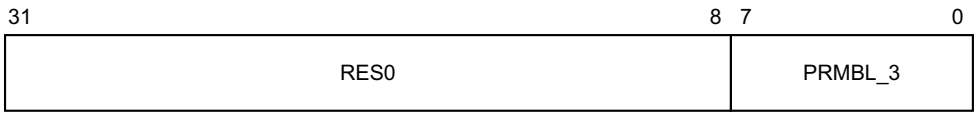
There are no configuration notes.

Attributes

CIDR3 is a 32-bit register.

Field descriptions

The CIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

CoreSight component identification preamble. See CoreSight Architecture Specification.

This field reads as `0xB1`.

Accessing the CIDR3:

CIDR3 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFFC`

This interface is accessible as follows:

- Access to this register is RO.

3.1.9 CLAIMCLR, Claim Tag Clear Register

The CLAIMCLR characteristics are:

Purpose

In conjunction with CLAIMSET, provides bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

Usage constraints

There are no usage constraints.

Configurations

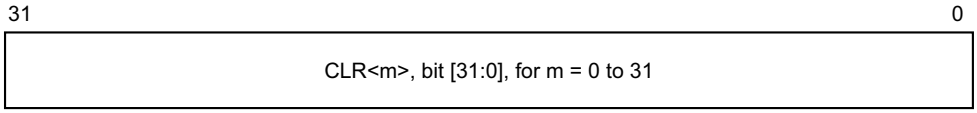
Some or all RW fields of this register have defined reset values.

Attributes

CLAIMCLR is a 32-bit register.

Field descriptions

The CLAIMCLR bit assignments are:



CLR<m>, bit [31:0], for m = 0 to 31

Claim Tag Clear. Each bit indicates the current status of the claim tag bit, and is used to clear the relevant claim tag bit.

The defined values of this bit are:

`0b0` **When Read** Claim tag bit not set.

`0b1` **When Read** Claim tag bit set.

`0b0` **When Written** Ignored.

`0b1` **When Written** Clear claim tag bit to 0.

The number of claim tag bits implemented is IMPLEMENTATION DEFINED and indicated in CLAIMSET. The ETR architecture does not require any claim tag bits to be implemented.

On a reset, this field resets to `0`.

Accessing the CLAIMCLR:

CLAIMCLR can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFA4`

This interface is accessible as follows:

- Access to this register is RW.

3.1.10 CLAIMSET, Claim Tag Set Register

The CLAIMSET characteristics are:

Purpose

In conjunction with CLAIMCLR, provides bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

Usage constraints

There are no usage constraints.

Configurations

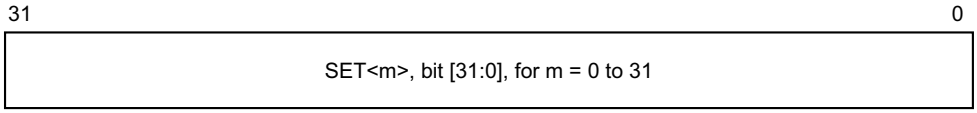
There are no configuration notes.

Attributes

CLAIMSET is a 32-bit register.

Field descriptions

The CLAIMSET bit assignments are:



SET<m>, bit [31:0], for m = 0 to 31

Claim Tag Set. Each bit indicates whether the claim tag bit is implemented, and is used to set the relevant claim tag bit.

The defined values of this bit are:

`0b0` **When Read** Claim tag bit not implemented.

`0b1` **When Read** Claim tag bit implemented.

`0b0` **When Written** Ignored.

`0b1` **When Written** Set claim tag bit to 1.

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. The ETR architecture does not require any claim tag bits to be implemented.

Accessing the CLAIMSET:

CLAIMSET can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFA0`

This interface is accessible as follows:

- Access to this register is RW.

3.1.11 CTL, Control Register

The CTL characteristics are:

Purpose

Controls trace stream capture.

Usage constraints

There are no usage constraints.

Configurations

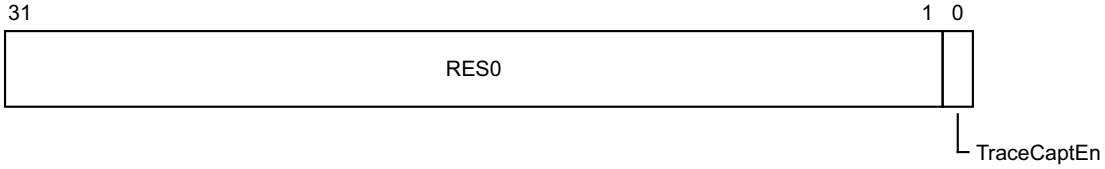
Some or all RW fields of this register have defined reset values.

Attributes

CTL is a 32-bit register.

Field descriptions

The CTL bit assignments are:



Bits [31:1]

Reserved, RES0.

TraceCaptEn, bit [0]

Trace Capture Enable. Controls trace capture. The possible values of this bit are:

`0b0` Trace capture disabled.

`0b1` Trace capture enabled.

On a reset, this field resets to `0`.

Accessing the CTL:

CTL can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x020`

This interface is accessible as follows:

- Access to this register is RW.

3.1.12 DBA, Data Buffer Address

The DBA characteristics are:

Purpose

Locates the trace buffer in system memory.

Usage constraints

Writes to DBA are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

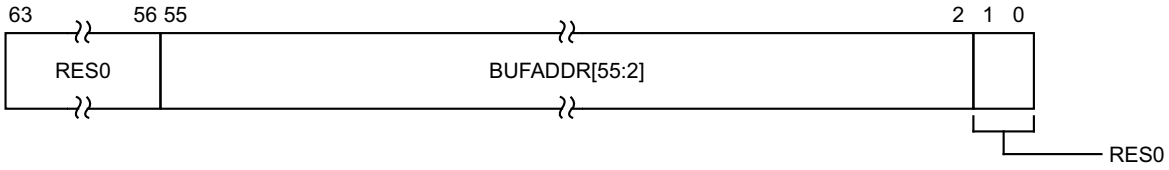
There are no configuration notes.

Attributes

DBA is a 64-bit register.

Field descriptions

The DBA bit assignments are:



Bits [63:56]

Reserved, RES0.

BUFADDR[55:2], bits [55:2]

Buffer Address. Locates the trace buffer in system memory.

Software must set this field to a multiple of both the minimum alignment defined by DEVID.MEMWIDTH and 4.

Unimplemented most-significant address bits are RES0.

Bits [1:0]

Reserved, RES0.

Accessing the DBA:

DBA can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x118

This interface is accessible as follows:

- Access to this register is RW.

3.1.13 DEVARCH, Device Architecture Register

The DEVARCH characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

There are no configuration notes.

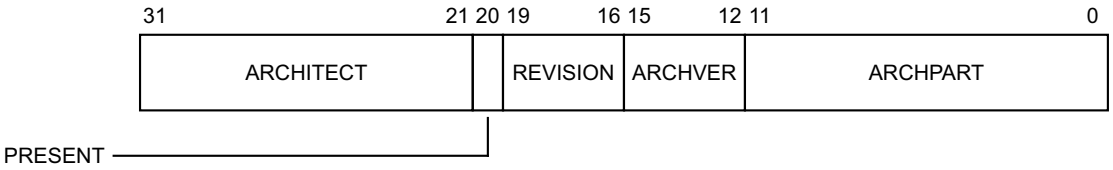
Attributes

DEVARCH is a 32-bit register.

Field descriptions

The DEVARCH bit assignments are:

When DEVARCH.PRESENT == 1:



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:

`0b01000111011`

JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present. The defined values of this bit are:

`0b0` DEVARCH information not present. The rest of the register is RES0.

`0b1` DEVARCH information present.

This bit reads as 1.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component. The defined values of this field are:

`0b0000` Revision 0 of Version 0 of the ETR Architecture.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component. The defined values of this field are:

`0b0000` Version 0 of the ETR Architecture.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, such that ARCHVER is ARCHID[15:12].

ARCHPART, bits [11:0]

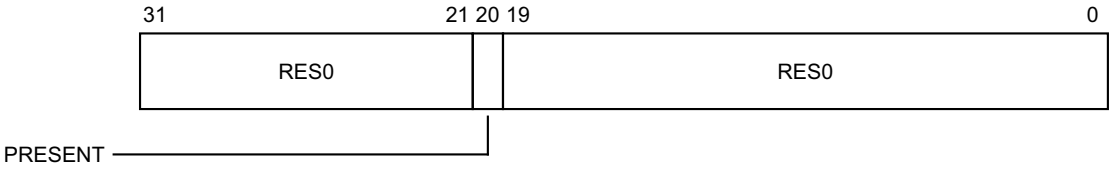
Architecture Part. Defines the architecture of the component. The defined values of this field are:

`0xA21` ETR Architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, such that ARCHPART is ARCHID[11:0].

This field reads as `0xA21`.

When DEVARCH.PRESENT == 0:



Bits [31:21]

Reserved, RES0.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present. The defined values of this bit are:

`0b0` DEVARCH information not present. The rest of the register is RES0.

`0b1` DEVARCH information present.

This bit reads as 0

Bits [19:0]

Reserved, RES0.

Accessing the DEVARCH:

DEVARCH can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFBC`

This interface is accessible as follows:

- Access to this register is RO.

3.1.14 DEVID, Device Configuration Register

The DEVID characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

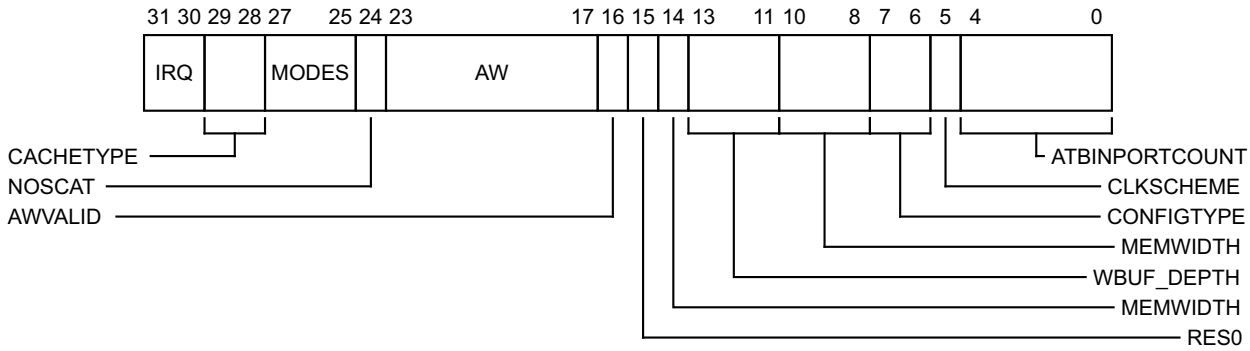
There are no configuration notes.

Attributes

DEVID is a 32-bit register.

Field descriptions

The DEVID bit assignments are:



IRQ, bits [31:30]

Interrupt Controls. Defines which interrupt controls are implemented. The defined values of this field are:

`0b00` No interrupt controls implemented. MODE.IRQOnReady, MODE.IRQOnFull, STS.MSI, and the IRQCR<n> registers are RES0.

The ETR behaves as if MODE.IRQOnFull is `0b1` and MODE.IRQOnReady is `0b0`.

Note

If Software Read FIFO mode 2 is implemented then an interrupt for the Software Read FIFO mode 2 operation is always implemented, even if this field is set to `0b00`.

`0b01` Wired interrupt controls implemented. MODE.IRQOnReady and MODE.IRQOnFull are RW, and STS.MSI and the IRQCR<n> registers are RES0.

`0b10` Message-signaled interrupt controls implemented. MODE.IRQOnReady, MODE.IRQOnFull and the IRQCR<n> registers are RW, and STS.MSI is a RO status bit.

Other values are Reserved.

CACHETYPE, bits [29:28]

When DEVID.CONFIGTYPE == 0b01:

Cacheability Controls. Defines the format of the BUSCTL bus control fields. The defined values of this field are:

`0b00` BUSCTL.{BSH} are not implemented. BUSCTL.BAttr uses an IMPLEMENTATION DEFINED non-generic format, and applies to both scatter table and buffer accesses.

`0b01` BUSCTL.{BSH, BAttr} use an IMPLEMENTATION DEFINED non-generic format.

`0b10` BUSCTL.{BSH, BAttr} use a generic format with only Outer attributes.

`0b11` BUSCTL.{BSH, BAttr} use a generic format with separate Inner and Outer attributes.

Otherwise:

Reserved, RES0.

MODES, bits [27:25]

When DEVID.CONFIGTYPE == 0b01:

Implemented modes. Indicates which modes, and associated registers, are implemented. The defined values of this field are:

`0b000` Circular Buffer and Software Read FIFO mode 1 implemented. The RRP, RRD, LBUFLEVEL, CBUFLEVEL and BUFWM registers are all implemented. The RURP register is not implemented.

`0b001` Circular Buffer, Software Read FIFO mode 1, and Software Read FIFO mode 2 implemented. The RRP, RURP, RRD, LBUFLEVEL, CBUFLEVEL and BUFWM registers are all implemented.

`0b010` Circular Buffer mode only. The RRP, RURP, RRD, LBUFLEVEL, CBUFLEVEL and BUFWM registers are not implemented.

`0b011` Circular Buffer and Software Read FIFO mode 2 implemented. The RRP, LBUFLEVEL, CBUFLEVEL and BUFWM registers are all implemented. The RRD register is not implemented.

All other values are reserved.

Otherwise:

Reserved, UNKNOWN.

NOSCAT, bit [24]

When DEVID.CONFIGTYPE == 0b01:

No Scatter Mode. Indicates whether Scatter mode is supported. The defined values of this bit are:

`0b0` Scatter mode is implemented.

`0b1` Scatter mode is not implemented.

Otherwise:

Reserved, RES0.

AW, bits [23:17]

Address Width. Defines the width of the RRP, RURP, RWP and DBA registers. The defined values of this field are:

`0b0000000` Configured as a streaming device.

`0b0100000` 32 bits.

`0b0100100` 36 bits.

`0b0101000` 40 bits.

`0b0101100` 44 bits.

`0b0110000` 48 bits.

0b0110100 52 bits.

0b0111000 56 bits.

Other values are Reserved.

Only valid if DEVID.AWVALID indicates this field is valid.

AWVALID, bit [16]

Address Width Valid. The defined values of this bit are:

0b0 AW field is not valid.

0b1 AW field is valid.

Bit [15]

Reserved, RES0.

MEMWIDTH, bit [14]

MEMWIDTH is a split field that occupies DEVID[14, 10:8].

Minimum Alignment. This value indicates the required alignment of the RRP, RURP, RWP, DBA, BUFWM and RSZ registers. The defined values of this field are:

0b0000 No minimum alignment.

0b0001 Minimum alignment is 2 bytes.

0b0010 Minimum alignment is 4 bytes.

0b0011 Minimum alignment is 8 bytes.

0b0100 Minimum alignment is 16 bytes.

0b0101 Minimum alignment is 32 bytes.

0b0110 Minimum alignment is 64 bytes.

0b0111 Minimum alignment is 128 bytes.

0b1000 Minimum alignment is 256 bytes.

0b1001 Minimum alignment is 512 bytes.

0b1010 Minimum alignment is 1024 bytes.

0b1011 Minimum alignment is 2,048 bytes.

All other values are reserved.

- The RRP and RWP address pointers must be aligned to a multiple of the minimum alignment.
- The value written to RURP (if implemented) must be a multiple of the minimum alignment.
- If the trace formatter is not implemented, or not enabled, DBA, BUFWM and RSZ must be aligned to a multiple of the smaller of 4 bytes and the minimum alignment.
- If the trace formatter is implemented and enabled, DBA, BUFWM and RSZ must be aligned to a multiple of the smaller of 16 bytes and the minimum alignment.

However, the RSZ register always allows the value 1 (4 bytes), regardless of the minimum alignment.

Note

The BUFWM and RSZ registers always specify a number of 32-bit words.

WBUF_DEPTH, bits [13:11]

Write Buffer Depth. Indicates, in powers of two, the number of entries in the Write buffer. Each entry is of size MEMWIDTH. The defined values of this field are:

0b000 IMPLEMENTATION DEFINED.

0b010 Depth of the Write buffer is 4 entries.

0b011 Depth of the Write buffer is 8 entries.

0b100 Depth of the Write buffer is 16 entries.

0b101 Depth of the Write buffer is 32 entries.

Other values are Reserved.

MEMWIDTH, bits [10:8]

MEMWIDTH is a split field that occupies DEVID[14, 10:8].

Minimum Alignment. This value indicates the required alignment of the RRP, RURP, RWP, DBA, BUFWM and RSZ registers. The defined values of this field are:

0b0000 No minimum alignment.

0b0001 Minimum alignment is 2 bytes.

0b0010 Minimum alignment is 4 bytes.

0b0011 Minimum alignment is 8 bytes.

0b0100 Minimum alignment is 16 bytes.

0b0101 Minimum alignment is 32 bytes.

0b0110 Minimum alignment is 64 bytes.

0b0111 Minimum alignment is 128 bytes.

0b1000 Minimum alignment is 256 bytes.

0b1001 Minimum alignment is 512 bytes.

0b1010 Minimum alignment is 1024 bytes.

0b1011 Minimum alignment is 2,048 bytes.

All other values are reserved.

- The RRP and RWP address pointers must be aligned to a multiple of the minimum alignment.
- The value written to RURP (if implemented) must be a multiple of the minimum alignment.
- If the trace formatter is not implemented, or not enabled, DBA, BUFWM and RSZ must be aligned to a multiple of the smaller of 4 bytes and the minimum alignment.
- If the trace formatter is implemented and enabled, DBA, BUFWM and RSZ must be aligned to a multiple of the smaller of 16 bytes and the minimum alignment.

However, the RSZ register always allows the value 1 (4 bytes), regardless of the minimum alignment.

Note

The BUFWM and RSZ registers always specify a number of 32-bit words.

CONFIGTYPE, bits [7:6]

Configuration Type. This value indicates TMC configuration types. The defined values of this field are:

0b00 ETB.

0b01 ETR.

0b10 ETF.

This field reads as 0b01.

CLKSCHEME, bit [5]

Reserved. This field is RES0.

ATBINPORTCOUNT, bits [4:0]

Reserved. This field is RES0.

Accessing the DEVID:

DEVID can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFC8

This interface is accessible as follows:

- Access to this register is RO.

3.1.15 DEVID1, Device Configuration Register 1

The DEVID1 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

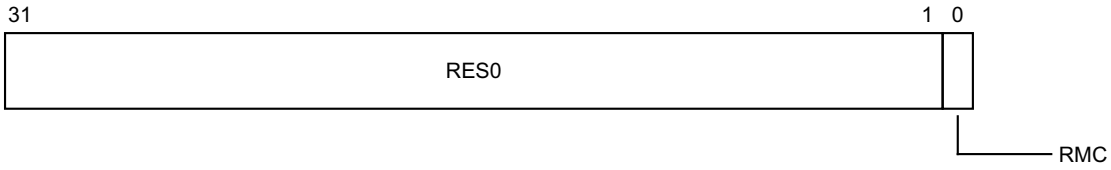
There are no configuration notes.

Attributes

DEVID1 is a 32-bit register.

Field descriptions

The DEVID1 bit assignments are:



Bits [31:1]

Reserved, RES0.

RMC, bit [0]

Register Management Configuration mode.

The defined values of this bit are:

`0b0` Register Management Configuration mode is IMPLEMENTATION DEFINED. Some registers values might be automatically managed when entering or leaving the Disabled state.

`0b1` Register Management Configuration mode 1. Registers behave as defined in this architecture specification.

All implementations compliant to this specification implement Register Management Configuration mode 1, however some implementations which are compliant to this specification might have this field set to 0. Arm recommends that all implementations compliant to this specification set this field to 1.

Accessing the DEVID1:

DEVID1 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFC4`

This interface is accessible as follows:

- Access to this register is RO.

3.1.16 DEVTYPE, CoreSight Device Type Register

The DEVTYPE characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

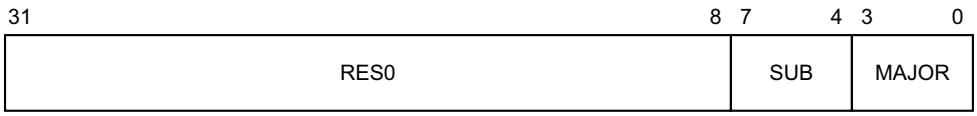
There are no configuration notes.

Attributes

DEVTYPE is a 32-bit register.

Field descriptions

The DEVTYPE bit assignments are:



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

The defined values of this field are:

`0b0010` ETR or ETB.

`0b0011` ETF.

This field reads as `0x2`.

MAJOR, bits [3:0]

The defined values of this field are:

`0b0001` ETR or ETB.

`0b0010` ETF.

This field reads as `0x1`.

Accessing the DEVTYPE:

DEVTYPE can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xFCC`

This interface is accessible as follows:

- Access to this register is RO.

3.1.17 FFCR, Formatter and Flush Control Register

The FFCR characteristics are:

Purpose

Controls the formatter.

Usage constraints

There are no usage constraints.

Configurations

Some or all RW fields of this register have defined reset values.

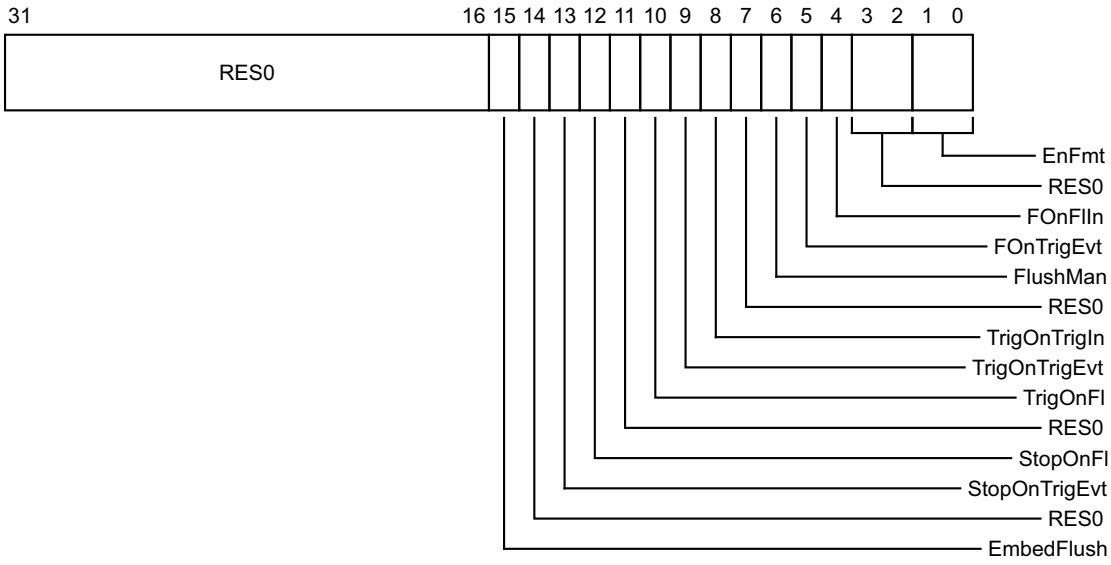
Attributes

FFCR is a 32-bit register.

Field descriptions

The FFCR bit assignments are:

When FFSR.FtNotPresent == 0:



Bits [31:16]

Reserved, RES0.

EmbedFlush, bit [15]

Embed flush completion packet. The possible values of this bit are:

`0b0` Disabled.

`0b1` Embed a flush immediately following the last data byte after a flush completes.

An embedded flush consists of a single zero data byte with a trace ID of `0x7B`.

This bit is ignored by the ETR when FFCR.EnFmt == `0b00`, meaning the formatter is in Bypass mode.

On a reset, this field resets to `0`.

Bit [14]

Reserved, RES0.

StopOnTrigEvt, bit [13]

Stop On Trigger Event. The possible values of this bit are:

`0b0` Disabled.

`0b1` Stop the formatter when a Trigger Event has been observed.

A Trigger Event occurs when TRG decrements to zero.

Enabling the ETR with this bit set to 1 is UNPREDICTABLE when MODE.MODE != `0b00` (Circular Buffer).

On a reset, this field resets to `0`.

StopOnFl, bit [12]

Stop On Flush. The possible values of this bit are:

`0b0` Disabled.

`0b1` Stop the formatter on a Flush Completion.

On a reset, this field resets to `0`.

Bit [11]

Reserved, RES0.

TrigOnFl, bit [10]

Trigger On Flush. The possible values of this bit are:

`0b0` Disabled.

`0b1` Embed a trigger on a Flush Completion.

An embedded trigger consists of a single zero data byte with a trace ID of `0x7D`.

This bit is ignored by the ETR when FFCR.EnFmt == `0b00`, meaning the formatter is in Bypass mode.

On a reset, this field resets to `0`.

TrigOnTrigEvt, bit [9]

Trigger On Trigger Event. The possible values of this bit are:

`0b0` Disabled.

`0b1` Insert an Embedded Trigger on a Trigger Event.

A Trigger Event occurs when TRG decrements to zero.

An Embedded Trigger consists of a single zero data byte with a trace ID of `0x7D`.

Enabling the ETR with this bit set to 1 is UNPREDICTABLE when MODE.MODE != `0b00` (Circular Buffer).

This bit is ignored by the ETR when FFCR.EnFmt == `0b00`, meaning the formatter is in Bypass mode.

On a reset, this field resets to `0`.

TrigOnTrigIn, bit [8]

Trigger On **TRIGIN**. The possible values of this bit are:

`0b0` Disabled.

`0b1` Insert an Embedded Trigger on assertion of the IMPLEMENTATION DEFINED **TRIGIN** signal.

An Embedded Trigger consists of a single zero data byte with a trace ID of `0x7D`.

It is IMPLEMENTATION DEFINED whether this bit is R/W or RAO. If the **TRIGIN** signal is not supported, this field is RES0.

This bit is ignored by the ETR when FFCR.EnFmt == 0b00, meaning the formatter is in Bypass mode.

Note

If an Embedded Cross Trigger is implemented, Arm recommends that the **TRIGIN** signal is implemented and connected to a CTI output trigger.

On a reset, this field resets to 0.

Bit [7]

Reserved, RES0.

FlushMan, bit [6]

Manually Generate Flush.

0b0 **When Read** Manual flush not in progress.

0b1 **When Read** Manual flush in progress.

0b0 **When Written** Ignored.

0b1 **When Written** Cause a Detected Flush.

This bit ignores writes if CTL.TraceCaptEn == 0.

On a reset, this field resets to 0.

FOnTrigEvt, bit [5]

Flush On Trigger Event. The possible values of this bit are:

0b0 Disabled.

0b1 Cause a Detected Flush when a Trigger Event occurs.

A Trigger Event occurs when TRG decrements to zero.

Enabling the ETR with this bit set to 1 is UNPREDICTABLE when MODE.MODE != 0b00 (Circular Buffer).

This bit is ignored by the ETR when FFCR.StopOnTrigEvt == 1.

On a reset, this field resets to 0.

FOnFlIn, bit [4]

Flush On **FLUSHIN**. The possible values of this bit are:

0b0 Disabled.

0b1 Cause a Detected Flush on assertion of the IMPLEMENTATION DEFINED **FLUSHIN** signal, if the formatter is not already stopped.

On a reset, this field resets to 0.

Bits [3:2]

Reserved, RES0.

EnFmt, bits [1:0]

Formatter control. Selects the output formatting mode. The possible values of this field are:

0b00 Bypass. Disable formatting. Trace data on reserved trace IDs other than 0x00 is discarded.

0b01 Normal. Enable formatting but do not embed triggers in the formatted output.

0b10 Reserved. The ETR behaves as 0b11, and the value read back from this field is UNKNOWN.

0b11 Continuous. Enable formatting and embed triggers in the formatted output.

It is IMPLEMENTATION DEFINED whether Normal mode is supported.

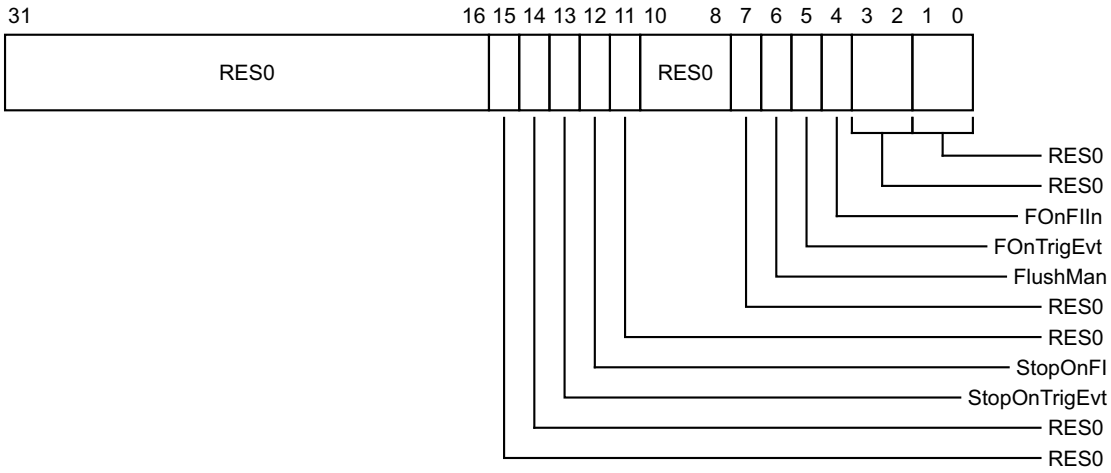
It is IMPLEMENTATION DEFINED whether Continuous mode is supported.

When the formatter is implemented, at least one of Normal or Continuous modes are implemented.

Changing the value of this bit when not in the Disabled state is UNPREDICTABLE.

This field resets to 0.

When FFSR.FtNotPresent == 1:



Bits [31:16]

Reserved, RES0.

Bit [15]

Reserved, RES0.

Bit [14]

Reserved, RES0.

StopOnTrigEvt, bit [13]

Stop On Trigger Event. The possible values of this bit are:

0b0 Disabled.

0b1 Stop the formatter when a Trigger Event has been observed.

A Trigger Event occurs when TRG decrements to zero.

Enabling the ETR with this bit set to 1 is UNPREDICTABLE when MODE.MODE != 0b00 (Circular Buffer).

On a reset, this field resets to 0.

StopOnFl, bit [12]

Stop On Flush. The possible values of this bit are:

0b0 Disabled.

0b1 Stop the formatter on a Flush Completion.

On a reset, this field resets to 0.

Bit [11]

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

Bit [7]

Reserved, RES0.

FlushMan, bit [6]

Manually Generate Flush.

`0b0` **When Read** Manual flush not in progress.

`0b1` **When Read** Manual flush in progress.

`0b0` **When Written** Ignored.

`0b1` **When Written** Cause a Detected Flush.

This bit ignores writes if CTL.TraceCaptEn == 0.

On a reset, this field resets to `0`.

FOnTrigEvt, bit [5]

Flush On Trigger Event. The possible values of this bit are:

`0b0` Disabled.

`0b1` Cause a Detected Flush when a Trigger Event occurs.

A Trigger Event occurs when TRG decrements to zero.

Enabling the ETR with this bit set to 1 is UNPREDICTABLE when MODE.MODE != `0b00` (Circular Buffer).

This bit is ignored by the ETR when FFCR.StopOnTrigEvt == 1.

On a reset, this field resets to `0`.

FOnFlIn, bit [4]

Flush On **FLUSHIN.** The possible values of this bit are:

`0b0` Disabled.

`0b1` Cause a Detected Flush on assertion of the IMPLEMENTATION DEFINED **FLUSHIN** signal, if the formatter is not already stopped.

On a reset, this field resets to `0`.

Bits [3:2]

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing the FFCR:

FFCR can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x304`

This interface is accessible as follows:

- Access to this register is RW.

3.1.18 FFSR, Formatter and Flush Status Register

The FFSR characteristics are:

Purpose

Shows the status and capabilities of the formatter.

Usage constraints

There are no usage constraints.

Configurations

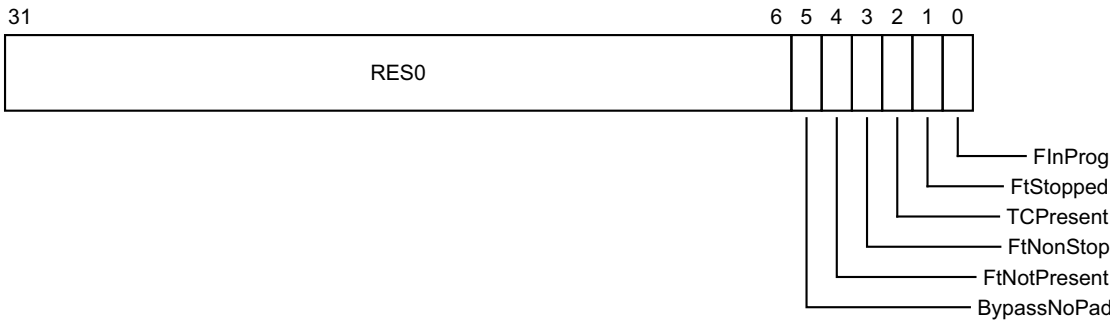
There are no configuration notes.

Attributes

FFSR is a 32-bit register.

Field descriptions

The FFSR bit assignments are:



Bits [31:6]

Reserved, RES0.

BypassNoPad, bit [5]

No padding in Bypass. Indicates whether the ETR inserts the Stop Sequence when trace is disabled and Bypass mode is selected. The defined values of this bit are:

`0b0` ETR inserts the Stop Sequence at end of trace in Bypass mode.

`0b1` ETR does not insert the Stop Sequence at end of trace in Bypass mode.

If the ETR does not insert the Stop Sequence, the trace source must ensure that the trace buffer is aligned to a multiple of DEVID.MEMWIDTH when trace is disabled.

This bit reads-as-zero if the formatter is implemented and cannot be stopped.

FtNotPresent, bit [4]

Formatter not present. Indicates the presence of the formatter. The defined values of this bit are:

`0b0` Formatter is implemented.

`0b1` Formatter is not implemented.

The formatter must be implemented if the ETR supports multiple trace sources.

FtNonStop, bit [3]

Non-stop. Indicates the trace cannot be stopped. The defined values of this bit are:

`0b0` Formatter can be stopped.

`0b1` Formatter cannot be stopped.

The ETR can always be stopped.

This bit reads as zero.

TCPresent, bit [2]

TRACECTL present. Indicates the presence of the TRACECTL pin.

The ETR does not include a trace port and so never implements a TRACECTL pin.

This bit reads as zero.

FtStopped, bit [1]

Formatter stopped. The defined values of this bit are:

`0b0` Formatter is enabled.

`0b1` The formatter has received a stop request signal and all trace data and post-amble has been output. Any further trace data is ignored.

This bit reads-as-zero if any of the following are true:

- The formatter is not implemented.
- The formatter cannot be stopped.

FInProg, bit [0]

Flush in progress. Set to one when a Detected Flush occurs and clears to zero on a Flush Completion. A Flush Completion occurs when all data received before the flush is acknowledged has been output to the trace buffer.

Accessing the FFSR:

FFSR can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x300`

This interface is accessible as follows:

- Access to this register is RO.

3.1.19 IRQCR0, Interrupt Configuration Register

The IRQCR0 characteristics are:

Purpose

Interrupt configuration register.

Usage constraints

Writes to IRQCR0 are UNPREDICTABLE unless the ETR is in the Disabled state and STS.MSI is 0.

Configurations

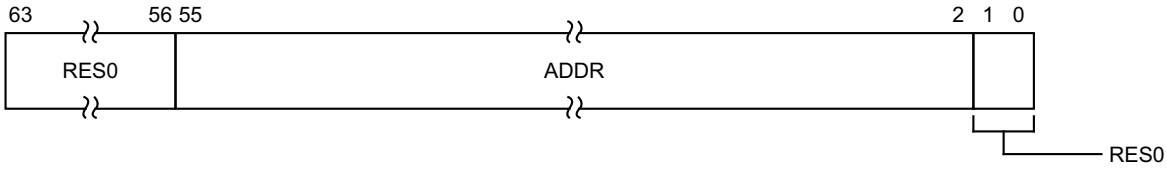
Present only if message-signaled interrupts are implemented. RES0 otherwise.

Attributes

IRQCR0 is a 64-bit register.

Field descriptions

The IRQCR0 bit assignments are:



Bits [63:56]

Reserved, RES0.

ADDR, bits [55:2]

Message Signaled Interrupt address. Specifies the address that ETR writes to when signaling an interrupt.

The size of a physical address is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

Bits [1:0]

Reserved, RES0.

Accessing the IRQCR0:

IRQCR0 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xE80

This interface is accessible as follows:

- Access to this register is RW.

3.1.20 IRQCR1, Interrupt Configuration Register 1

The IRQCR1 characteristics are:

Purpose

Interrupt configuration register.

Usage constraints

Writes to IRQCR1 are UNPREDICTABLE unless the ETR is in the Disabled state and STS.MSI is 0.

Configurations

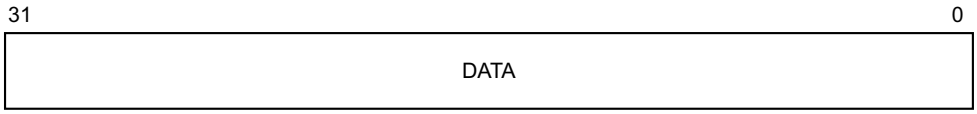
Present only if message-signaled interrupts are implemented. RES0 otherwise.

Attributes

IRQCR1 is a 32-bit register.

Field descriptions

The IRQCR1 bit assignments are:



DATA, bits [31:0]

Payload for a message signaled interrupt.

Accessing the IRQCR1:

IRQCR1 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xE88

This interface is accessible as follows:

- Access to this register is RW.

3.1.21 IRQCR2, Interrupt Configuration Register 2

The IRQCR2 characteristics are:

Purpose

Interrupt configuration register.

Usage constraints

Writes to IRQCR2 are UNPREDICTABLE unless the ETR is in the Disabled state and STS.MSI is 0.

Configurations

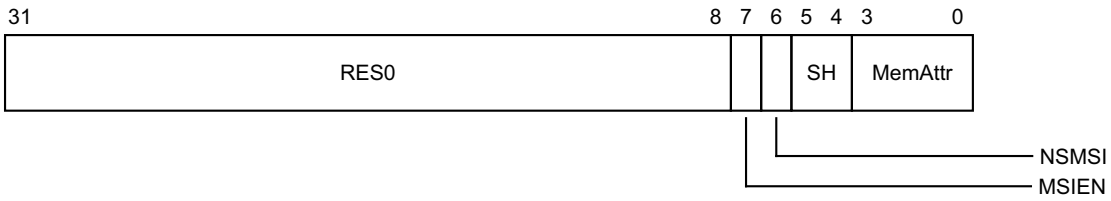
Present only if message-signaled interrupts are implemented. RES0 otherwise.

Attributes

IRQCR2 is a 32-bit register.

Field descriptions

The IRQCR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

MSIEN, bit [7]

Message signaled interrupt enable. Enables generation of message signaled interrupts.

The possible values of this bit are:

`0b0` Disabled.

`0b1` Enabled.

NSMSI, bit [6]

Non-secure. Defines the physical address space for message signaled interrupts. The possible values of this bit are:

`0b0` Secure.

`0b1` Non-secure.

If the ETR supports both Secure and Non-secure message signaled interrupts, this field is RW, and the MSI is Secure or Non-secure based on the value of this field.

If the ETR supports only Secure message signaled interrupts, this field is RES0, and the MSI is Secure.

If the ETR supports only Non-secure message signaled interrupts, this field is RES0, and the MSI is Non-secure.

If Invasive debug is disabled, message signaled interrupts are not generated.

If Secure Invasive debug is disabled, Secure message signaled interrupts are not generated.

SH, bits [5:4]

Shareability. Defines the shareability domain for message signaled interrupts. The possible values of this field are:

`0b00` Not shared.

`0b10` Outer shareable.

`0b11` Inner shareable.

This field is RES0 if the ETR does not support configuring the shareability domain, meaning shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

MemAttr, bits [3:0]

Memory type. Defines the memory type for message signaled interrupts. The possible values of this field are:

`0b0000` Device-nGnRnE.

`0b0001` Device-nGnRE.

`0b0010` Device-nGRE.

`0b0011` Device-GRE.

`0b0101` Outer Non-cacheable, Inner Non-cacheable.

`0b0110` Outer Non-cacheable, Inner Write-through Cacheable.

`0b0111` Outer Non-cacheable, Inner Write-back Cacheable.

`0b1001` Outer Write-through Cacheable, Inner Non-cacheable.

`0b1010` Outer Write-through Cacheable, Inner Write-through Cacheable.

`0b1011` Outer Write-through Cacheable, Inner Write-back Cacheable.

`0b1101` Outer Write-back Cacheable, Inner Non-cacheable.

`0b1110` Outer Write-back Cacheable, Inner Write-through Cacheable.

`0b1111` Outer Write-back Cacheable, Inner Write-back Cacheable.

This field is RES0 if the ETR does not support configuring the memory type, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

Note

This is the same format as the VMSAv8-64 stage 2 memory region attributes.

Accessing the IRQCR2:

IRQCR2 can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xE8C`

This interface is accessible as follows:

- Access to this register is RW.

3.1.22 ITCTRL, Integration Mode Control Register

The ITCTRL characteristics are:

Purpose

A component can use this register to switch between functional mode and integration mode.

Usage constraints

There are no usage constraints.

Configurations

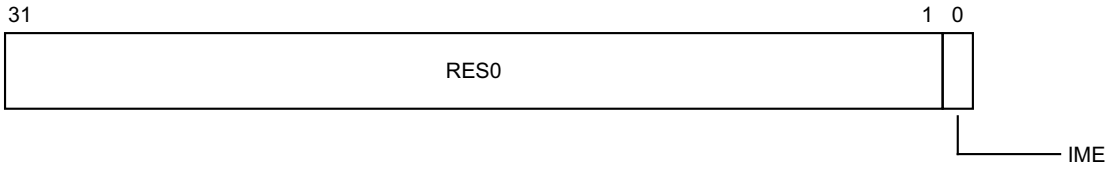
Some or all RW fields of this register have defined reset values.

Attributes

ITCTRL is a 32-bit register.

Field descriptions

The ITCTRL bit assignments are:



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration Mode Enable. The values of this bit are:

`0b0` Functional mode.

`0b1` Integration mode.

Support for integration mode is optional. If no integration mode is implemented, this field is RES0.

After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

On a reset, this field resets to `0`.

Accessing the ITCTRL:

ITCTRL can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0xF00`

This interface is accessible as follows:

- Access to this register is RW.

3.1.23 LAR, Lock Access Register

The LAR characteristics are:

Purpose

Used to lock and unlock the software lock.

Usage constraints

There are no usage constraints.

Configurations

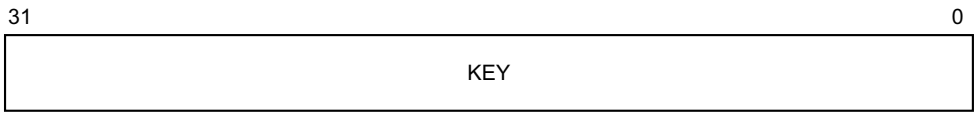
There are no configuration notes.

Attributes

LAR is a 32-bit register.

Field descriptions

The LAR bit assignments are:



KEY, bits [31:0]

Software Lock Key. Writes to this register are ignored.

Accessing the LAR:

LAR can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFB0

This interface is accessible as follows:

- Access to this register is WO.

3.1.24 LBUFLEVEL, Latched Buffer Fill Level

The LBUFLEVEL characteristics are:

Purpose

The maximum fill level of the trace memory in 32-bit words since the register was last read.

Usage constraints

There are no usage constraints.

Configurations

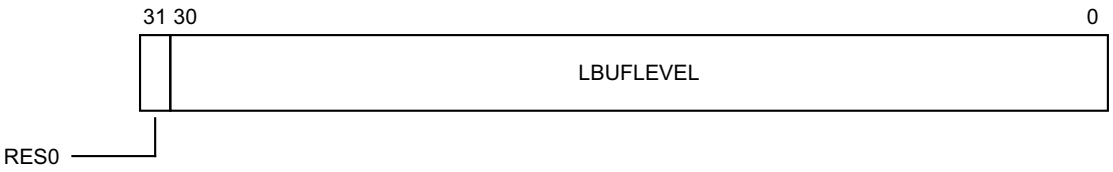
Present only if Software Read FIFO modes 1 or 2 are implemented. RES0 otherwise.

Attributes

LBUFLEVEL is a 32-bit clears-to-zero-after-read memory-mapped register (RC).

Field descriptions

The LBUFLEVEL bit assignments are:



Bit [31]

Reserved, RES0.

LBUFLEVEL, bits [30:0]

Latched Buffer Fill Level. The maximum fill level of the trace memory in 32-bit words since the register was last read.

This field is cleared to zero:

- After being read, if CTL.TraceCaptEn == 1
- On exiting the Disabled state.

This field is not valid and reads UNKNOWN if MODE.MODE == `0b00`, meaning Circular Buffer mode is selected.

Accessing the LBUFLEVEL:

LBUFLEVEL can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x02C`

This interface is accessible as follows:

- Access to this register is RC.

3.1.25 LSR, Lock Status Register

The LSR characteristics are:

Purpose

Indicates whether the software lock is implemented, and the current status of the software lock.

Usage constraints

There are no usage constraints.

Configurations

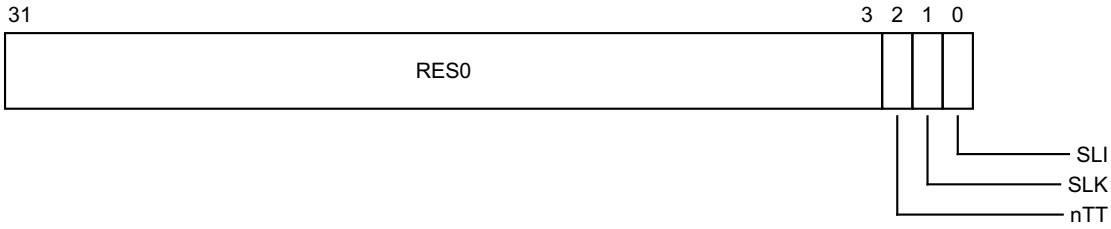
There are no configuration notes.

Attributes

LSR is a 32-bit register.

Field descriptions

The LSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

This field is always zero. This bit reads as zero.

SLK, bit [1]

This field is used to indicate the current software lock status. The defined values of this bit are:

`0b0` Software lock is not locked. Writing to the other registers in the ETR is permitted.

`0b1` Software lock is locked. Writing to the other registers in the ETR except the LAR is blocked.

This bit reads as zero.

SLI, bit [0]

This field is used to indicate whether the software lock is implemented. The defined values of this bit are:

`0b0` Software lock is not implemented. Writes to the LAR are ignored.

`0b1` Software lock is implemented.

This bit reads as zero.

Accessing the LSR:

LSR can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFB4

This interface is accessible as follows:

- Access to this register is RO.

3.1.26 MODE, Mode Register

The MODE characteristics are:

Purpose

Controls ETR operating mode and buffer interrupt generation.

Usage constraints

Always implemented.

Configurations

There are no configuration notes.

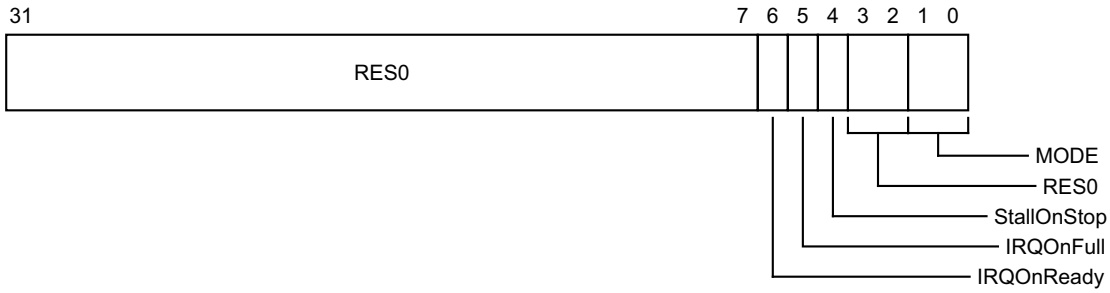
Attributes

MODE is a 32-bit register.

Field descriptions

The MODE bit assignments are:

When DEVID.IRQ != 0b00:



Bits [31:7]

Reserved, RES0.

IRQOnReady, bit [6]

Interrupt on Ready. Controls buffer interrupt generation when the ETR is in the Stopped state. The possible values of this bit are:

`0b0` No interrupt generated.

`0b1` Buffer interrupt generated when the ETR enters the Stopped state.

The buffer interrupt is generated when CTL.TraceCaptEn == 1 and:

- If the interrupt is message-signaled or edge-triggered, when STS.TMCReady changes from 0 to 1.
- If the interrupt is level-sensitive, while STS.TMCReady == 1.

Writes to MODE must not change this bit while STS.MSI == 1.

IRQOnFull, bit [5]

Interrupt on Full. Controls buffer interrupt generation when STS.Full == 1. The possible values of this bit are:

`0b0` No interrupt generated.

`0b1` Buffer interrupt generated when STS.Full becomes set to 1.

The buffer interrupt is generated when the ETR is not in the Disabled state and:

- If the interrupt is message-signaled or edge-triggered, when STS.Full changes from 0 to 1.

- If the interrupt is level-sensitive, while STS.Full == 1.

Writes to MODE must not change this bit while STS.MSI == 1.

StallOnStop, bit [4]

Stall trace input when formatter stops. The possible values of this bit are:

`0b0` Discard trace when in the Stopped or Disabled states.

`0b1` Do not discard trace when in the Stopped or Disabled states.

Setting this bit to 1 might avoid loss of trace while reprogramming the ETR. However, this cannot be guaranteed because the trace source might start discarding trace if it cannot stop generating trace and there is not sufficient storage for it.

Support for this bit is IMPLEMENTATION DEFINED. When not implemented, this field is RES0 and the ETR behaves as if the bit is 0.

Bits [3:2]

Reserved, RES0.

MODE, bits [1:0]

Mode. Selects the operating mode. The possible values of this field are:

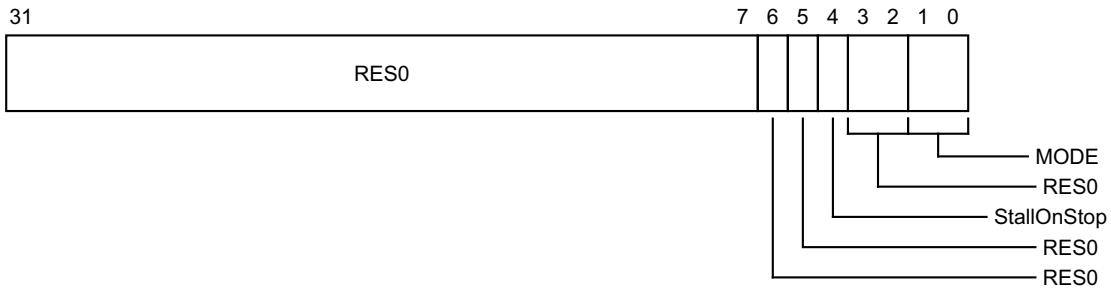
`0b00` Circular Buffer mode.

`0b01` Software Read FIFO mode 1.

`0b11` Software Read FIFO mode 2. The buffer interrupt signal is asserted when the trace memory fill level (RWP - RRP) crosses the programmed watermark level (RSZ - BUFWM). If the buffer interrupt is level-sensitive it stays asserted until the fill level drop below the watermark. The buffer interrupt is enabled regardless of the value of the MODE.IRQOnReady and MODE.IRQOnFull bits.

All other values are reserved. If DEVID.MODES<0> == 0, the value `0b11` is reserved. If DEVID.MODES<1> == 1, the value `0b01` is reserved. If this field is programmed with a reserved value, the ETR behaves as if this field has a defined value, other than for a direct read of the register. Software must not rely on the behavior of reserved values, as they might change in a future version of the architecture.

When DEVID.IRQ == 0b00:



Bits [31:7]

Reserved, RES0.

Bit [6]

Reserved, RES0.

Bit [5]

Reserved, RES0.

StallOnStop, bit [4]

Interrupt on Full. Controls buffer interrupt generation when STS.Full == 1. The possible values of this bit are:

`0b0` Discard trace when in the Stopped or Disabled states.

`0b1` Do not discard trace when in the Stopped or Disabled states.

Setting this bit to 1 might avoid loss of trace while reprogramming the ETR. However, this cannot be guaranteed because the trace source might start discarding trace if it cannot stop generating trace and there is not sufficient storage for it. Support for this bit is IMPLEMENTATION DEFINED. When not implemented, this field is RES0 and the ETR behaves as if the bit is 0.

Bits [3:2]

Reserved, RES0.

MODE, bits [1:0]

Mode. Selects the operating mode. The possible values of this field are:

`0b00` Circular Buffer mode.

`0b01` Software Read FIFO mode 1.

`0b11` Software Read FIFO mode 2. The buffer interrupt signal is asserted when the trace memory fill level (RWP - RRP) crosses the programmed watermark level (RSZ - BUFWM). If the buffer interrupt is level-sensitive it stays asserted until the fill level drop below the watermark.

All other values are reserved. If DEVID.MODES<0> == 0, the value `0b11` is reserved. If DEVID.MODES<1> == 1, the value `0b01` is reserved. If this field is programmed with a reserved value, the ETR behaves as if this field has a defined value, other than for a direct read of the register. Software must not rely on the behavior of reserved values, as they might change in a future version of the architecture.

Accessing the MODE:

MODE can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x028`

This interface is accessible as follows:

- Access to this register is RW.

3.1.27 PIDR0, Peripheral Identification Register

The PIDR0 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

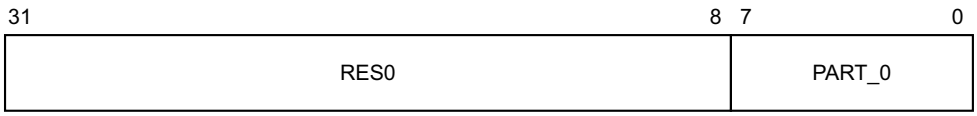
There are no configuration notes.

Attributes

PIDR0 is a 32-bit register.

Field descriptions

The PIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number bits [7:0]. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

Accessing the PIDR0:

PIDR0 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFE0

This interface is accessible as follows:

- Access to this register is RO.

3.1.28 PIDR1, Peripheral Identification Register

The PIDR1 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

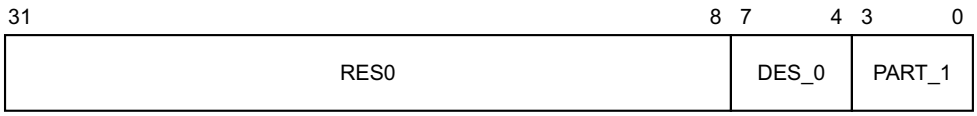
There are no configuration notes.

Attributes

PIDR1 is a 32-bit register.

Field descriptions

The PIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

JEP106 identification code bits [3:0]. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

PART_1, bits [3:0]

Part number bits [11:8]. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

Accessing the PIDR1:

PIDR1 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFE4

This interface is accessible as follows:

- Access to this register is RO.

3.1.29 PIDR2, Peripheral Identification Register

The PIDR2 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

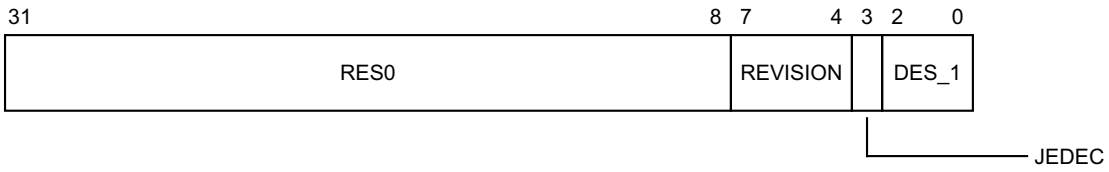
There are no configuration notes.

Attributes

PIDR2 is a 32-bit register.

Field descriptions

The PIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component revision. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

JEDEC, bit [3]

JEDEC assignee value is used. See CoreSight Architecture Specification.

This bit reads as one.

DES_1, bits [2:0]

JEP106 identification code bits [6:4]. See CoreSight Architecture Specification.

Accessing the PIDR2:

PIDR2 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFE8

This interface is accessible as follows:

- Access to this register is RO.

3.1.30 PIDR3, Peripheral Identification Register

The PIDR3 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

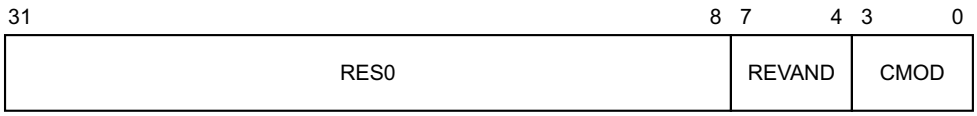
There are no configuration notes.

Attributes

PIDR3 is a 32-bit register.

Field descriptions

The PIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

RevAnd. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

CMOD, bits [3:0]

Customer Modified. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

Accessing the PIDR3:

PIDR3 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFEC

This interface is accessible as follows:

- Access to this register is RO.

3.1.31 PIDR4, Peripheral Identification Register

The PIDR4 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

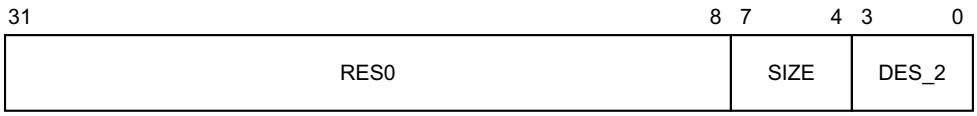
There are no configuration notes.

Attributes

PIDR4 is a 32-bit register.

Field descriptions

The PIDR4 bit assignments are:



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

4KB count. See CoreSight Architecture Specification.

This field reads as zero.

DES_2, bits [3:0]

JEP106 continuation code. See CoreSight Architecture Specification.

This field reads as an IMPLEMENTATION DEFINED value.

Accessing the PIDR4:

PIDR4 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFD0

This interface is accessible as follows:

- Access to this register is RO.

3.1.32 PIDR5, Peripheral Identification Register

The PIDR5 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

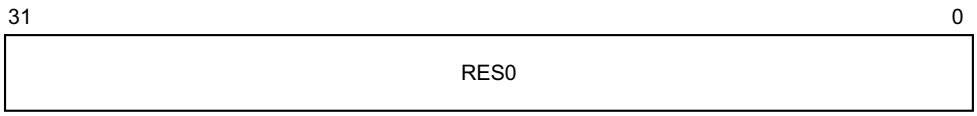
There are no configuration notes.

Attributes

PIDR5 is a 32-bit register.

Field descriptions

The PIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

Accessing the PIDR5:

PIDR5 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFD4

This interface is accessible as follows:

- Access to this register is RO.

3.1.33 PIDR6, Peripheral Identification Register

The PIDR6 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

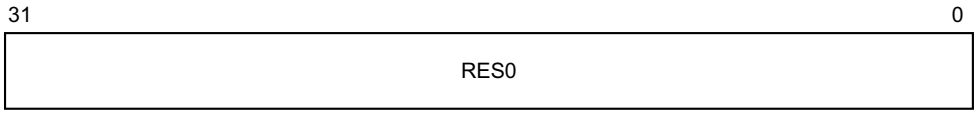
There are no configuration notes.

Attributes

PIDR6 is a 32-bit register.

Field descriptions

The PIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

Accessing the PIDR6:

PIDR6 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFD8

This interface is accessible as follows:

- Access to this register is RO.

3.1.34 PIDR7, Peripheral Identification Register

The PIDR7 characteristics are:

Purpose

Provides CoreSight discovery information.

Usage constraints

There are no usage constraints.

Configurations

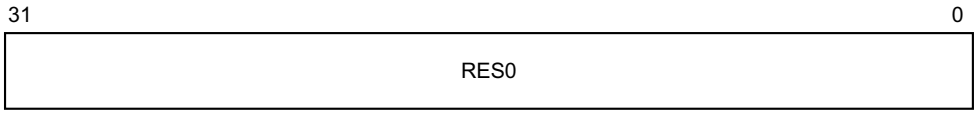
There are no configuration notes.

Attributes

PIDR7 is a 32-bit register.

Field descriptions

The PIDR7 bit assignments are:



Bits [31:0]

Reserved, RES0.

Accessing the PIDR7:

PIDR7 can be accessed through its memory-mapped interface:

Component	Offset
ETR	0xFDC

This interface is accessible as follows:

- Access to this register is RO.

3.1.35 PSCR, Periodic Synchronization Control Register

The PSCR characteristics are:

Purpose

Defines the reload value for the periodic Synchronization Counter register. The Synchronization Counter decrements for each byte that is output, and, on reaching zero, forces completion of the current formatter frame if the formatter is implemented and enabled.

Usage constraints

Writes to PSCR are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

Some or all RW fields of this register have defined reset values.

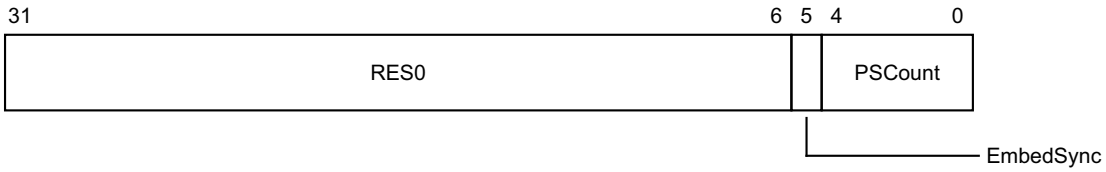
Implementation of this register is OPTIONAL.

Attributes

PSCR is a 32-bit register.

Field descriptions

The PSCR bit assignments are:



Bits [31:6]

Reserved, RES0.

EmbedSync, bit [5]

When FFSR.FtNotPresent == 0b0:

Embed Frame Sync Packet in the trace stream. Enables the formatter to insert frame synchronization packets in the trace stream at periodic intervals. The possible values of this bit are:

`0b0` Do not insert frame sync packet.

`0b1` If the Synchronization Counter is enabled, the formatter inserts a 32-bit frame sync packet in the trace stream when the counter reaches 0.

This field is optional and when not implemented is RAZ.

This bit is ignored by the ETR when FFCR.EnFmt == `0b00`, meaning the formatter is in Bypass mode.

Otherwise:

Reserved, RES0.

PSCount, bits [4:0]

Determines the reload value of the Synchronization Counter. The reload value takes effect the next time the counter reaches zero. Reads from this register return the reload value programmed into this register. The possible values of this field are:

`0b00000` Synchronization disabled.

`0b00111` 128 bytes.

`0b01000` 256 bytes.

`0b01001` 512 bytes.

`0b01010` 1024 bytes.

`0b01011` 2 to the power of 11 bytes.

`0b01100` 2 to the power of 12 bytes.

`0b01101` 2 to the power of 13 bytes.

`0b01110` 2 to the power of 14 bytes.

`0b01111` 2 to the power of 15 bytes.

`0b10000` 2 to the power of 16 bytes.

`0b10001` 2 to the power of 17 bytes.

`0b10010` 2 to the power of 18 bytes.

`0b10011` 2 to the power of 19 bytes.

`0b10100` 2 to the power of 20 bytes.

`0b10101` 2 to the power of 21 bytes.

`0b10110` 2 to the power of 22 bytes.

`0b10111` 2 to the power of 23 bytes.

`0b11000` 2 to the power of 24 bytes.

`0b11001` 2 to the power of 25 bytes.

`0b11010` 2 to the power of 26 bytes.

`0b11011` 2 to the power of 27 bytes.

`0b11100` 2 to the power of 28 bytes.

`0b11101` 2 to the power of 29 bytes.

`0b11110` 2 to the power of 30 bytes.

`0b11111` 2 to the power of 31 bytes.

All other values are reserved. If this field is programmed with a reserved value, the ETR behaves as if this field has a defined value, other than for a direct read of the register. Software must not rely on the behavior of reserved values, as they might change in a future version of the architecture.

The Synchronization Counter might have a maximum value smaller than 2 to the power of 31. In this case, if the programmed reload value is greater than the maximum value, then the Synchronization Counter is reloaded with its maximum value.

This field is optional and when not implemented is RAZ, and synchronization requests are not implemented.

This field resets to `0x0A`.

Accessing the PSCR:

PSCR can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x308`

This interface is accessible as follows:

- Access to this register is RW.

3.1.36 RRD, RAM Read Data register

The RRD characteristics are:

Purpose

Reads data from the trace memory.

Usage constraints

There are no usage constraints.

Configurations

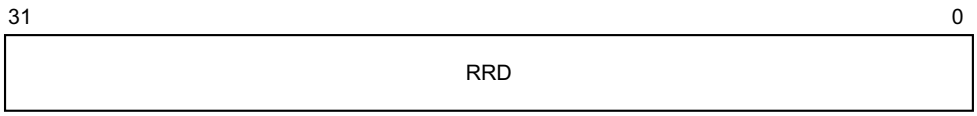
Present only if Software Read FIFO mode 1 is implemented. RES0 otherwise.

Attributes

RRD is a 32-bit register.

Field descriptions

The RRD bit assignments are:



RRD, bits [31:0]

RAM Read Data.

Returns read data from trace memory. If STS.MemErr == 1, a read returns an error response.

Returns the contents of the memory at the location addressed by the RRP register, and the RRP register is incremented by 4, if any of:

- CTL.TraceCaptEn == 0 and STS.TMCReady == 1
- MODE.MODE == `0b00` (Circular Buffer), CTL.TraceCaptEn == 1, the trace buffer is not empty, and STS.TMCReady == 1
- MODE.MODE == `0b01` (Software Read FIFO 1), the trace buffer is not empty, and either CTL.TraceCaptEn == 1 or STS.TMCReady == 0

Otherwise returns `0xFFFFFFFF` and RRP is unchanged.

Accessing the RRD:

RRD can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x010`

This interface is accessible as follows:

- Access to this register is RO.

3.1.37 RRP, RAM Read Pointer

The RRP characteristics are:

Purpose

The read pointer used to read entries from trace buffer through RRD.

Usage constraints

Writes to RRP are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

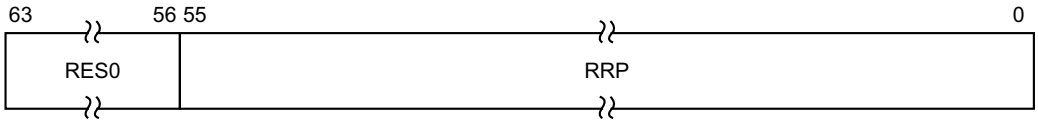
Present only if Software Read FIFO modes 1 or 2 are implemented. RES0 otherwise.

Attributes

RRP is a 64-bit register.

Field descriptions

The RRP bit assignments are:



Bits [63:56]

Reserved, RES0.

RRP, bits [55:0]

RAM Read Pointer.

When initializing this field, software must write a value that is aligned to the minimum alignment defined by DEVID.MEMWIDTH. When saving and restoring this field, software must preserve all bits.

Unimplemented most-significant address bits are RES0.

Accessing the RRP:

RRP[31:0] can be accessed through its memory-mapped interface:

Component	Offset	Range
ETR	0x014	31:0

RRP[63:32] can be accessed through its memory-mapped interface:

Component	Offset	Range
ETR	0x038	63:32

These interfaces are accessible as follows:

- Access to RRP[31:0] is RW.
- Access to RRP[63:32] is RW.

3.1.38 RSZ, RAM Size Register

The RSZ characteristics are:

Purpose

Defines the size, in 32-bit words, of the trace buffer.

Usage constraints

Writes to RSZ are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

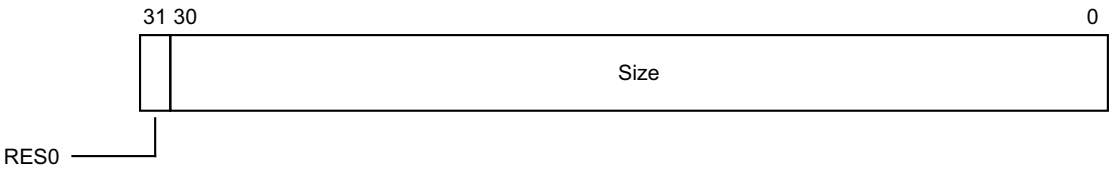
There are no configuration notes.

Attributes

RSZ is a 32-bit register.

Field descriptions

The RSZ bit assignments are:



Bit [31]

Reserved, RES0.

Size, bits [30:0]

Size. Size of the trace buffer in 32-bit words.

Values greater than `0x40000000` (4GB) are reserved.

Software must set this field to either 1 or a multiple of the minimum alignment defined by DEVID.MEMWIDTH divided by 4.

If this field is set to 1, the ETR writes all trace in single writes to the same location. Each write writes DEVID.MEMWIDTH bytes.

Accessing the RSZ:

RSZ can be accessed through its memory-mapped interface:

Component	Offset
ETR	`0x004`

This interface is accessible as follows:

- Access to this register is RW.

3.1.39 RURP, RAM Update Read Pointer

The RURP characteristics are:

Purpose

Increment RAM Read Pointer.

Usage constraints

There are no usage constraints.

Configurations

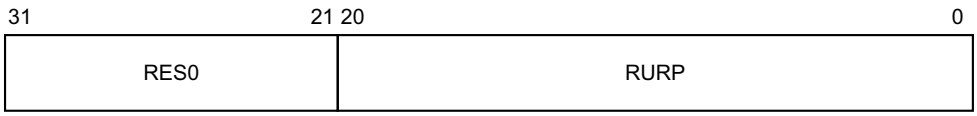
Present only if Software Read FIFO mode 2 is implemented. RES0 otherwise.

Attributes

RURP is a 32-bit register.

Field descriptions

The RURP bit assignments are:



Bits [31:21]

Reserved, RES0.

RURP, bits [20:0]

Update RAM Read Pointer. The RAM Read Pointer is incremented by the amount written to this field, modulo the size of the buffer.

The value written to RURP must be a multiple of the larger of:

- The memory width indicated by DEVID.MEMWIDTH.
- 16 bytes, if the formatter is enabled.

Accessing the RURP:

RURP can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x120

This interface is accessible as follows:

- Access to this register is WO.

3.1.40 RWD, RAM Write Data register

The RWD characteristics are:

Purpose

Write data to the trace memory.

Usage constraints

Writes to RWD are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

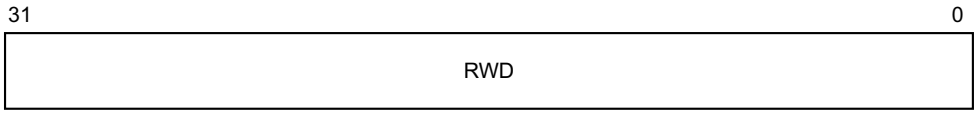
Implementation of this register is OPTIONAL.

Attributes

RWD is a 32-bit register.

Field descriptions

The RWD bit assignments are:



RWD, bits [31:0]

RAM Write Data. This register is used for testing.

Accessing the RWD:

RWD can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x024

This interface is accessible as follows:

- Access to this register is WO.

3.1.41 RWP, RAM Write Pointer

The RWP characteristics are:

Purpose

The write pointer used to write entries into trace buffer.

Usage constraints

Writes to RWP are UNPREDICTABLE unless the ETR is in the Disabled state.

Configurations

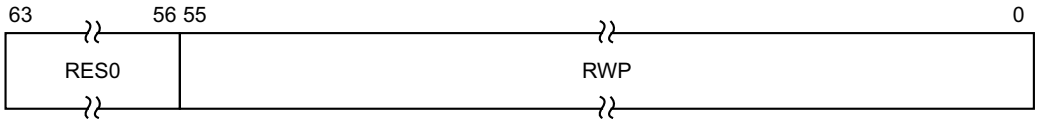
There are no configuration notes.

Attributes

RWP is a 64-bit register.

Field descriptions

The RWP bit assignments are:



Bits [63:56]

Reserved, RES0.

RWP, bits [55:0]

RAM Write Pointer. Indicates the next location where the ETR will write data in system memory.

High-order unimplemented address bits are RES0.

When initializing this field, software must write a value that is aligned to the minimum alignment defined by DEVID.MEMWIDTH. When saving and restoring this field, software must preserve all bits.

Accessing the RWP:

RWP[31:0] can be accessed through its memory-mapped interface:

Component	Offset	Range
ETR	0x018	31:0

RWP[63:32] can be accessed through its memory-mapped interface:

Component	Offset	Range
ETR	0x03C	63:32

These interfaces are accessible as follows:

- Access to RWP[31:0] is RW.
- Access to RWP[63:32] is RW.

3.1.42 STS, Status Register

The STS characteristics are:

Purpose

Status register.

Usage constraints

None.

Configurations

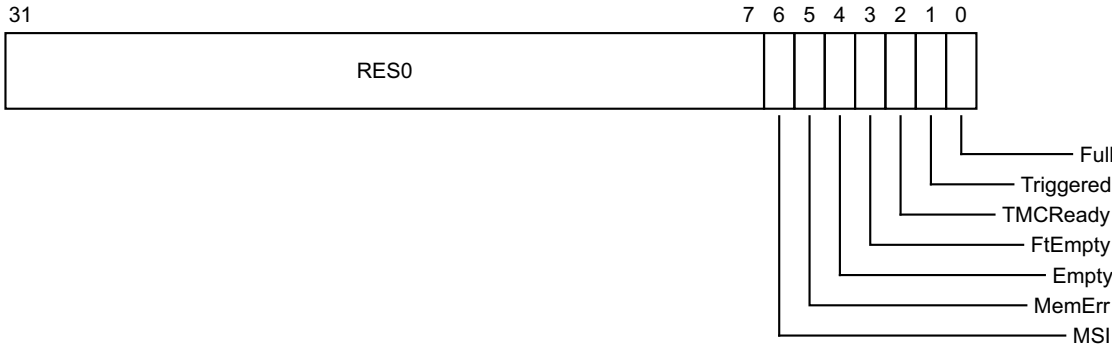
Some or all RW fields of this register have defined reset values.

Attributes

STS is a 32-bit register.

Field descriptions

The STS bit assignments are:



Bits [31:7]

Reserved, RES0.

MSI, bit [6]

When DEVID.IRQ == 0b10:

Message Signaled Interrupt status.

Indicates that a Message Signaled Interrupt is in flight. This bit is set to 1 when a message signaled interrupt is sent by the ETR and cleared to zero when the message acknowledge is received.

The defined values of this bit are:

`0b0` No interrupt is in flight.

`0b1` An interrupt message has been sent and the acknowledge not yet received.

This bit is RES0 if message-signaled interrupts are not implemented.

This bit is read-only.

Otherwise:

Reserved, RES0.

MemErr, bit [5]

Memory Error.

Indicates that an error has occurred on the memory interface. The error could be because of an error response from a slave or because of the status of the Authentication interface inputs.

The possible values of this bit are:

`0b0` No error has occurred since the last time this bit was cleared to zero.

`0b1` Since the last time this bit was cleared to zero, one of:

- A read or write error response was encountered.
- An access would have been initiated when Non-secure Invasive Debug is disabled.
- A secure access would have been initiated when Secure Invasive Debug is disabled.

While set to 1:

- No more accesses are performed other than any commands in the internal pipeline.
- Outstanding read and write responses are ignored. Pending responses are counted, but the responses received are discarded.
- A read of RRD or a write to RWD returns an error. This includes accesses to these registers that would otherwise have been ignored without returning an error, for example, when the ETR is enabled and MODE.MODE == `0b00` (Circular Buffer).
- If the formatter is implemented, the formatter stops, and sets STS.TMCReady when it has stopped.
- Reading the RWP and RRP registers returns UNKNOWN values.

This bit is cleared to zero by:

- Writing to STS with this bit set to 1 when STS.TMCReady was previously set to 1.
- Setting CTL.TraceCaptEn to 1 when CTL.TraceCaptEn was previously set to 0.

Empty, bit [4]

Indicates the trace memory does not contain any valid trace data. The defined values of this bit are:

`0b0` Trace memory does contain valid trace data or CTL.TraceCaptEn == 0.

`0b1` Trace memory does not contain valid trace data and CTL.TraceCaptEn == 1.

Note

However trace data might be present internally to the ETR and not yet written to memory. Software must use STS.TMCReady to determine whether there is trace data present internally to the ETR.

This bit resets to an UNKNOWN value. On leaving the Disabled state, this bit dynamically indicates the current status of the trace buffer. This bit retains its current value on entering the Disabled state.

This bit is read-only.

Note

If MODE.MODE == `0b00` (Circular Buffer), it is possible that STS.Empty and STS.Full read as one at the same time.

FtEmpty, bit [3]

Formatter empty. Trace capture has stopped. Set to 1 when trace capture has stopped, and all internal pipelines and buffers have drained. Unlike STS.TMCReady, it is not affected by buffer drains and accesses.

This bit is read-only.

TMCReady, bit [2]

Ready. The defined values of this bit are:

`0b0` ETR not ready.

`0b1` All of the following are true:

- Trace capture has stopped and all internal pipelines and buffers have drained. This is equivalent to being in the Stopped or Disabled state.
- The memory interface is not busy. This case can be used to detect page table reads in Scatter mode when in Stopped state.

This bit is read-only.

On a reset, this field resets to `1`.

Triggered, bit [1]

Triggered.

Set to 1 when all of the following apply:

- MODE.MODE == `0b00` (Circular Buffer).
- Trace capture is in progress.
- The ETR has detected a Trigger Event.

This bit is cleared to 0 when leaving Disabled state.

This bit does not indicate that a trigger has been embedded in the formatted output trace data from the ETR. Trigger indication on the output trace stream is controlled by the FFCR.

This bit reads-as-zero if MODE.MODE != `0b00`, meaning Circular Buffer mode is not selected.

Full, bit [0]

Used to help to determine how much of the trace buffer contains valid data.

When MODE.MODE != `0b00` (Software Read FIFO) and not in Disabled state, the possible values of this bit are:

`0b0` The current space in the trace memory is greater than the value programmed in BUFWM

`0b1` The current space in the trace memory is less than or equal to the value programmed in BUFWM. That is, Fill level >= (RSZ - BUFWM).

Note

If the ETR is programmed for Scatter operation, this bit indiscates whether the Trace memory is currently full regardless of the value programmed in the BUFWM Register.

When MODE.MODE == `0b00` (Circular buffer) and not in Disabled state, the possible values of this bit are:

`0b0` The RAM Write pointer has not wrapped around the top of the buffer since this bit was last cleared by software.

`0b1` The RAM Write pointer has wrapped around the top of the buffer since this bit was last cleared by software.

This bit is not updated by the ETR while the ETR is in the Disabled state.

This bit retains its value on entering the Disabled state.

Note

In Scatter mode, the ETR might be unable to accept new trace without this bit being set to 1, if a read from the Scatter tables is delayed sufficiently.

In previous versions of the ETR, this bit is cleared to 0 when exiting Disabled state. This ETR architecture requires software to initialize this bit before leaving Disabled, which also permits saving and restoring of the ETR state.

Accessing the STS:

STS can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x00C

This interface is accessible as follows:

- Access to this register is RW.

3.1.43 TRG, Trigger Counter Register

The TRG characteristics are:

Purpose

Specifies the number of words to capture following a Detected Trigger before a Trigger Event.

Usage constraints

There are no usage constraints.

Configurations

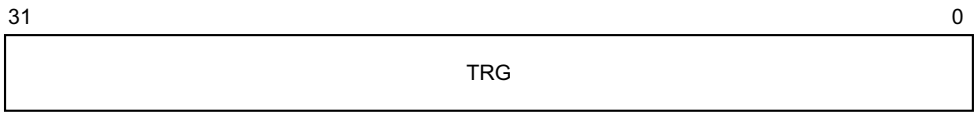
There are no configuration notes.

Attributes

TRG is a 32-bit register.

Field descriptions

The TRG bit assignments are:



TRG, bits [31:0]

Trigger count. Specifies the number of words to capture following a Detected Trigger before a Trigger Event.

Accessing the TRG:

TRG can be accessed through its memory-mapped interface:

Component	Offset
ETR	0x01C

This interface is accessible as follows:

- Access to this register is RW.

Appendix A
Pseudocode Definition

This appendix provides a definition of the pseudocode used in this document, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

A.1 About Arm pseudocode

Arm pseudocode provides precise descriptions of some areas of the architecture. The following sections describe the ARMv7 pseudocode in detail:

- *Data types* on page A-105.
- *Expressions* on page A-109.
- *Operators and built-in functions* on page A-111.
- *Statements and program structure* on page A-116.

A.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `SUBARCHITECTURE_DEFINED`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

For more information, see *Simple statements* on page A-116.

A.2 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page A-106.
- *Reals* on page A-106.
- *Booleans* on page A-106.
- *Enumerations* on page A-106.
- *Lists* on page A-107.
- *Arrays* on page A-108.

A.2.1 General data type rules

Arm architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- List.
- Array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1, y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y`, and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

A.2.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons, see *Equality and non-equality testing* on page A-111.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length N is bit (N–1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of, for example, registers, memory locations, and instructions. All of the remaining data types are abstract.

A.2.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} must be written in hexadecimal, it must be written as `-0x80000000`.

A.2.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point. This means `0` is an integer constant but `0.0` is a real constant.

A.2.5 Booleans

A Boolean is a logical true or false value.

The type name for Booleans is `boolean`. This is not the same type as `bit`, which is a length–1 bitstring. Boolean constants are `TRUE` and `FALSE`.

A.2.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration InstrSet {InstrSet_A32, InstrSet_T32, InstrSet_A64};
```

An enumeration always contains at least one symbolic constant, and a symbolic constant must not be shared between enumerations.

Enumerations must be declared explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the symbolic constants are its possible *constants*.

Note

A Boolean is a pre-declared enumeration that does not follow the normal naming convention and it has a special role in some pseudocode constructs, such as `if` statements, for example:

```
enumeration boolean {FALSE, TRUE};
```

A.2.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, for example:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this list at the start of this section is the return type of the function `Shift_C()` that performs a standard Arm shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than the `(…)` parentheses. These are:

- Bitstring extraction operators, that use lists of bit numbers or ranges of bit numbers surrounded by angle brackets `<…>`.
- Array indexing, that uses lists of array indexes surrounded by square brackets `[…]`.
- Array-like function argument passing, that uses lists of function arguments surrounded by square brackets `[…]`.

Each combination of data types in a list is a separate type, with type name given by listing the data types. This means that the example list at the start of this section is of type `(bits(32), bit)`. The general principle that types can be declared by assignment extends to the types of the individual list items in a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n`, and `(shift_t, shift_n)` to be of types `bits(2)`, `integer`, and `(bits(2), integer)`, respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as `abc.shift`, and `abc.amount`. This qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the definition of `ShiftSpec`, `ShiftSpec`, and `(bits(2), integer)` are two different names for the same type, not the names of two different types. To avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to can be written as `"-"` to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, for example the `('00', 0)` in the earlier example.

A.2.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

```
// The names of the Banked core registers.

enumeration RName {RName_0usr, RName_1usr, RName_2usr, RName_3usr, RName_4usr, RName_5usr,
                   RName_6usr, RName_7usr, RName_8usr, RName_8fiq, RName_9usr, RName_9fiq,
                   RName_10usr, RName_10fiq, RName_11usr, RName_11fiq, RName_12usr, RName_12fiq,
                   RName_SPusr, RName_SPfiq, RName_SPirq, RName_SPsvc,
                   RName_SPabt, RName_SPund, RName_SPmon, RName_SPhyp,
                   RName_LRusr, RName_LRfiq, RName_LRirq, RName_LRsvc,
                   RName_LRabt, RName_LRund, RName_LRmon,
                   RName_PC};


array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because:

- Enumerations always contain at least one symbolic constant.
- Integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing.

A.3 Expressions

This section describes:

- *General expression syntax*.
- *Operators and functions - polymorphism and prototypes* on page A-110.
- *Precedence rules* on page A-110.

A.3.1 General expression syntax

An expression is one of the following:

- A constant.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software.

Note

Some earlier documentation describes this as an UNPREDICTABLE value. UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type:

- For a constant, this data type is determined by the syntax of the constant.
- For a variable, there are the following possible sources for the data type:
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.
- For a language-defined operator, the definition of the operator determines the data type.

- For a function, the definition of the function determines the data type.

A.3.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each resulting form of an operator or function has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals, and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, or similar, in the prototype definition.

A.3.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables, and function invocations are evaluated with higher priority than any operators using their results.

2. Expressions on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.

3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but this is not necessary if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j`, and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

A.4 Operators and built-in functions

This section describes:

- *Operations on generic types*.
- *Operations on Booleans*.
- *Bitstring manipulation*.
- *Arithmetic* on page A-114.

A.4.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values `x` and `y` of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison is defined with a bitstring constant that includes `'x'` bits in addition to `'0'` and `'1'` bits. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`.

Note

This special form is permitted in the implied equality comparisons in `when` parts of `case … of …` structures.

Conditional selection

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

A.4.2 Operations on Booleans

If `x` is a Boolean, then `!x` is its logical inverse.

If `x` and `y` are Booleans, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

If `x` and `y` are Booleans, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

If `x` and `y` are Booleans, then `x ^ y` is the result of exclusive-ORing them together.

A.4.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.
- `TopBit(x)` is the leftmost bit of x. Using bitstring extraction, this means:

  `TopBit(x)= x<Len(x)-1>`.

Bitstring concatenation and replication

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

If x is a bitstring and n is an integer with `n > 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of n copies of x concatenated together
- `Zeros(n)` = `Replicate('0', n)`, `Ones(n)` = `Replicate('1', n)`.

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is `x<integer_list>`, where x is the integer or bitstring being extracted from, and `<integer_list>` is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in `<integer_list>`. In `x<integer_list>`, each of the integers in `<integer_list>` must be:

- `>= 0`
- `< Len(x)` if x is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `x<i, j, k,…, n>` is defined to be the concatenation:

  `x<i> : x<j> : x<k> : … : x<n>`.

- If `integer_list` consists of one integer `i`, `x<i>` is defined to be:
  - If x is a bitstring, `'0'` if bit `i` of x is a zero and `'1'` if bit `i` of x is a one.
  - If x is an integer, let y be the unique integer in the range `0` to `2^(i+1)-1` that is congruent to x modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

In `<integer_list>`, the notation `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` is shorthand for `instr<31, 30, 29, 28>`.

The expression `x<integer_list>` is assignable provided x is an assignable bitstring and no integer appears more than once in `<integer_list>`. In particular, `x<i>` is assignable if x is an assignable bitstring and `0 <= i < Len(x)`.

Encoding diagrams for registers frequently show named bits or multi-bit fields.

Logical operations on bitstrings

If x is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of x that are ones.

Testing a bitstring for being all zero or all ones

If x is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of x are zeros and `FALSE` if any of them are ones.
- `IsZeroBit(x)` produces `'1'` if all of the bits of x are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnesBit(x)` work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)

IsOnes(x)    = (BitCount(x) == Len(x))

IsZeroBit(x) = if IsZero(x) then '1' else '0'

IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If x is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to N. This means:

  `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.
- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N–1. This means:

  `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)`.

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then `ZeroExtend(x, i)` is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If x is a bitstring and i is an integer, then `SignExtend(x, i)` is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

Converting bitstrings to integers

If x is a bitstring, `SInt(x)` is the integer whose two's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

`UInt(x)` is the integer whose unsigned representation is x:

```
// UInt()
```

```
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

`Int(x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

A.4.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Unary plus, minus, and absolute value

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed, and `Abs(x)` is the absolute value of `x`. All three are of the same type as `x`.

Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant `N` bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

Comparisons

If `x` and `y` are integers or reals, then `x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, and `x >= y` are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results. In the case of `==` and `!=`, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Division and modulo

If x and y are integers or reals, then x/y is the result of dividing x by y, and is always of type real.

If x and y are integers, then x DIV y and x MOD y are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of x/y, x MOD y, or x DIV y in any context where y can be zero.

Square root

If x is an integer or a real, Sqrt(x) is its square root, and is always of type real.

Rounding and aligning

If x is a real:

- RoundDown(x) produces the largest integer n such that n <= x.
- RoundUp(x) produces the smallest integer n such that n >= x.
- RoundTowardsZero(x) produces RoundDown(x) if x > 0.0, 0 if x == 0.0, and RoundUp(x) if x < 0.0.

If x and y are both of type integer, Align(x, y) = y * (x DIV y) is of type integer.

If x is of type bitstring and y is of type integer, Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0> is a bitstring of the same length as x.

It is a pseudocode error to use either form of Align(x, y) in any context where y can be 0. In practice, Align(x, y) is only used with y a constant power of two, and the bitstring form used with y = 2^n has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If n is an integer, 2^n is the result of raising 2 to the power n and is of type real.

If x and n are of type integer, then:

- x << n = RoundDown(x * 2^n).
- x >> n = RoundDown(x * 2^(-n)).

Maximum and minimum

If x and y are integers or reals, then Max(x, y) and Min(x, y) are their maximum and minimum respectively. Both are of type integer if x and y are both of type integer, and real otherwise.

A.5 Statements and program structure

The following sections describe the control statements used in the pseudocode:

- *Simple statements*.
- *Compound statements* on page A-117.
- *Comments* on page A-121.

A.5.1 Simple statements

Each of the following simple statements must be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {<text>};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

SUBARCHITECTURE_DEFINED

This subsection describes the statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is SUBARCHITECTURE DEFINED. An optional `<text>` field can give more information.

A.5.2 Compound statements

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if … then … else …` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if … then … else …

A multi-line `if … then … else …` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    …
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    …
    <statement z>
else
    <statement A>
    <statement B>
    …
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and in the `else` part, if it is present, such as:

```
if <boolean_expression> then <statement 1>

if <boolean_expression> then <statement 1> else <statement A>

if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if … then … else …` expression.

repeat … until …

A `repeat … until …` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    …
    <statement n>
until <boolean_expression>;
```

while … do

A `while … do` structure takes the form:

```
while <boolean_expression> do
    <statement 1>
    <statement 2>
    …
    <statement n>
```

for …

A `for …` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

case … of …

A `case … of …` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        …
        <statement n>
        … more "when" groups …
    otherwise
        <statement A>
        <statement B>
        …
        <statement Z>
```

In this structure, `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. For details see *Equality and non-equality testing* on page A-111.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function is similar but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

A.5.3 Comments

Two styles of pseudocode comment exist:

- // starts a comment that is terminated by the end of the line.
- /* starts a comment that is terminated by */.

Glossary

This glossary describes some of the terms that are used in Arm documentation.

Abort
An abort occurs when an illegal memory access causes an exception. An abort can be generated by the hardware that manages memory accesses, or by the external memory system.

ADI
See Arm Debug Interface (ADI).

AHB
An AMBA bus protocol supporting pipelined operation, with the address and data phases occurring during different clock periods, meaning that the address phase of a transfer can occur during the data phase of the previous transfer. AHB provides a subset of the functionality of the AMBA AXI protocol.

See also AMBA.

Aligned
A data item stored at an address that is exactly divisible by the number of bytes that defines its data size. Aligned doublewords, words, and halfwords have addresses that are divisible by eight, four, and two respectively. An aligned access is one where the address of the access is aligned to the size of each element of the access.

AMBA
The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.

APB
An AMBA bus protocol for ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. It connects to the main system bus through a system-to-peripheral bus bridge that helps reduce system power consumption.

Arm Debug Interface (ADI)

The ADI connects a debugger to a device. The ADI is used to access memory-mapped components in a system, such as processors and CoreSight components. The ADI protocol defines the physical wire protocols permitted, and the logical programmers model.

AXI
An AMBA bus protocol that supports:

- Separate phases for address or control and data.

- Unaligned data transfers using byte strobes.
- Burst-based transactions with only start address issued.
- Separate read and write data channels.
- Issuing multiple outstanding addresses.
- Out-of-order transaction completion.
- Optional addition of register stages to meet timing or repropagation requirements.

The AXI protocol includes optional signaling extensions for low-power operation.

Big-endian In the context of the Arm architecture, big-endian is defined as the memory organization in which the least significant byte of a word is at a higher address than the most significant byte, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Little-endian and Endianness.

Boundary scan chain

A boundary scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. A core can contain several shift registers, enabling a scan to access selected parts of the device.

Burst A group of transfers that form a single transaction. With AMBA protocols, only the first transfer of the burst includes address information, and the transfer type determines the addresses used for subsequent transfers.

Cold reset A cold reset has the same effect as starting the processor by turning the power on. This clears main memory and many internal settings. Some program failures can lock up the core and require a cold reset to restart the system.

This is also known as power-on or powerup reset.

See also Processing Element (PE), Warm reset.

Core reset *See* Warm reset.

DAP *See* Debug Access Port (DAP).

Data Link layer The layer of an ADIv5 implementation that provides the functional and procedural means to transfer data between the external debugger and the *Debug Port* (DP). ADIv5 and upwards define two Data Link layers, one based on the IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture, referred to as JTAG, and one based on the Arm Serial Wire Debug protocol interface, referred to as SW-DP.

DATA LINK DEFINED

Means that the behavior is not defined by the base architecture, but must be defined and documented by individual Data Link layers of the architecture.

When DATA LINK DEFINED appears in body text, it is always in SMALL CAPITALS.

DBGTAP *See* Debug Test Access Port (DBGTAP).

Debug Access Port (DAP)

A block that acts as an AMBA, AHB, or AHB-Lite master on a system bus, to provide access to the debug target.

Debug Test Access Port (DBGTAP)

A debug control and data interface based on IEEE 1149.1 JTAG Test Access Port (TAP).

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned

A data item having a memory address that is divisible by eight.

Embedded Trace Macrocell (ETM)

A hardware macrocell that, when connected to a core, outputs trace information on a trace port. The ETM provides core-driven trace through a trace port compliant to the ATB protocol. An ETM always supports instruction trace, and might support data trace.

Endianness The scheme that determines the order of the successive bytes of data in a larger data structure when that structure is stored in memory.

See also Little-endian and Big-endian.

ETM See *Embedded Trace Macrocell (ETM)*.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned

A data item having a memory address that is divisible by 2.

Host A computer that provides data and other services to another computer. In the context of an Arm debugger, a computer providing debugging services to a target being debugged.

IMP DEF *See* IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED

Behavior that is not defined by the architecture, but must be defined and documented by individual implementations.

When IMPLEMENTATION DEFINED appears in body text, it is always in SMALL CAPITALS.

Joint Test Action Group (JTAG)

An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a Joint Test Action Group (JTAG) interface port to communicate with processors.

See IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture specification available from the IEEE Standards Association.

JTAG *See* Joint Test Action Group (JTAG).

JTAG Access Port (JTAG-AP)

An optional component of the DAP that provides debugger access to on-chip scan chains.

JTAG Debug Port (JTAG-DP)

An optional external interface for the DAP that provides a standard JTAG interface for debug access.

JTAG-AP *See* JTAG Access Port (JTAG-AP).

JTAG-DP *See* JTAG Debug Port (JTAG-DP).

Little-endian In the context of the Arm architecture, little-endian is defined as the memory organization in which the most significant byte of a word is at a higher address than the least significant byte.

See also Big-endian and Endianness.

PE *See* Processing Element (PE).

Powerup reset *See* Cold reset.

Processing Element (PE)

The abstract machine defined in the Arm architecture, as documented in the Arm *Architecture Reference Manual*. A PE implementation that is compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm *Architecture Reference Manual*.

RAO *See* Read-As-One (RAO).

RAO/WI Read-as-One, Writes Ignored.

Hardware must implement the field as Read-as-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, and on writes being ignored. This description can apply to a single bit that reads as `0b1`, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ *See* Read-As-Zero (RAZ).

RAZ/WI Read-as-Zero, Writes ignored.

Hardware must implement the field as Read-as-Zero, and must ignore writes to the field. Software can rely on the field reading as all 0s, and all writes being ignored. This description can apply to a single bit that reads as `0b0`, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-As-One (RAO)

Hardware must implement the field as reading as all 1s. Software can rely on the field reading as all 1s. This description can apply to a single bit that reads as `0b1`, or to a field that reads as all 1s.

Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s. Software can rely on the field reading as all 0s. This description can apply to a single bit that reads as `0b0`, or to a field that reads as all 0s.

RES0 A reserved bit or field with Should-Be-Zero-or-Preserved (SBZP) behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Note

RES0 is not used in descriptions of instruction encodings.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES0 for register fields is:

If a bit is RES0 in all contexts

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to `0b0`. In this case:
   - Reads of the bit always return `0b0`.
   - Writes to the bit are ignored.

   The bit might be described as RES0, WI, to distinguish it from a bit that behaves as described in 2.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to `0b0`.
   - A read of the bit returns the last value successfully written to the bit.

     Note

     As indicated in this list, this value might be written by an indirect write to the register.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the core, other than determining the value read back from the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is implementation defined on a field-by-field basis.

If a bit is RES0 only in some contexts

When the bit is described as RES0:

- An indirect write to the register sets the bit to `0b0`.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an unknown value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the core, other than determining the value read back from that bit.

For any RES0 bit, software:

- Must not rely on the bit reading as `0b0`.
- Must use an *SBZP* policy to write to the bit.

The RES0 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as `0b0`, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as *SBZ*.

This RES0 description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 0s.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1

A reserved bit or field with Should-Be-One-or-Preserved (SBOP) behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Note

RES1 is not used in descriptions of instruction encodings.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES1 for register fields is:

If a bit is RES1 in all contexts

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to `0b1`. In this case:
   - Reads of the bit always return `0b1`.
   - Writes to the bit are ignored.

   The bit might be described as RES1, WI, to distinguish it from a bit that behaves as described in 2.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to `0b1`.
   - A read of the bit returns the last value successfully written to the bit.

     Note

     As indicated in this list, this value might be written by an indirect write to the register.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the core, other than determining the value read back from the bit.

   Whether RES1 bits or fields follow behavior 1 or behavior 2 is implementation defined on a field-by-field basis.

If a bit is RES1 only in some contexts

When the bit is described as RES1:

- An indirect write to the register sets the bit to `0b1`.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an unknown value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the core, other than determining the value read back from that bit.

For any RES1 bit, software:

- Must not rely on the bit reading as `0b1`.
- Must use an *SBOP* policy to write to the bit.

The RES1 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as `0b1`, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as *SBO*.

This RES1 description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 1s.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved Unless otherwise stated in the architecture or product documentation:

- Reserved instruction and 32-bit system control register encodings are unpredictable.
- Reserved 64-bit system control register encodings are undefined.
- Reserved register bit fields are UNK/SBZP.

SBO *See* Should-Be-One (SBO).

SBOP *See* Should-Be-One-or-Preserved (SBOP).

SBZ *See* Should-Be-Zero (SBZ).

SBZP *See* Should-Be-Zero-or-Preserved (SBZP).

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

Serial Wire debug (SWD)
A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals required for a JTAG connection.

Serial-Wire Debug Port (SW-DP)

The interface for Serial Wire Debug.

Serial Wire JTAG Debug Port (SWJ-DP)

The SWJ-DP is a combined JTAG-DP and SW-DP that you can use to connect either a Serial Wire Debug (SWD) or JTAG probe to a target.

Should-Be-One (SBO)

Hardware must ignore writes to the field.

Software should write the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as `0b1`, or to a field that should be written as all 1s.

Should-Be-One-or-Preserved (SBOP)

The ARMv7 Large Physical Address Extension modified the definition of SBOP to apply to register fields that are SBOP in some but not all contexts. From the introduction of ARMv8 such register fields are described as RES1, see *RES1*. The definition of SBOP given here applies only to fields that are SBOP in all contexts.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it should preserve the value of the field by writing the value that it previously read from the field. Otherwise, it should write the field as all 1s.

If software writes a value to the field that is not a value previously read for the field and is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as `0b1`, or to a field that should be written as its preserved value or as all 1s.

See also Should-Be-Zero-or-Preserved (SBZP), Should-Be-One (SBO).

Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Software should write the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as `0b0`, or to a field that should be written as all 0s.

Should-Be-Zero-or-Preserved (SBZP)

The ARMv7 Large Physical Address Extension modified the definition of SBZP to apply to register fields that are SBZP in some but not all contexts. From the introduction of ARMv8 such register fields are described as RES0, see *RES0*. The definition of SBZP given here applies only to field that are SBZP in all contexts.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value previously read for the field and is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 0s.

See also Should-Be-One-or-Preserved (SBOP), Should-Be-Zero (SBZ).

SWD *See* Serial Wire debug (SWD).

SW-DP *See* Serial-Wire Debug Port (SW-DP).

SWJ-DP *See* Serial Wire JTAG Debug Port (SWJ-DP)

TAP *See* Test Access Port (TAP).

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. In the JTAG standard, the **nTRST** signal is optional, but this signal is mandatory in Arm processors because it is used to reset the debug logic.

See also Joint Test Action Group (JTAG), Debug Test Access Port (DBGTAP).

Trace port A port on a device, such as a processor or ASIC, to output trace information.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of the elements of the access.

See also Aligned.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE and do not return UNKNOWN values.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

When UNKNOWN appears in body text, it is always in SMALL CAPITALS.

UNP *See* UNPREDICTABLE.

UNPREDICTABLE

For an Arm processor, UNPREDICTABLE means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect. An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In an implementation that supports Virtualization, the Non-secure execution of unpredictable instructions at a lower level of privilege can be trapped to the hypervisor, provided that at least one instruction that is not unpredictable can be trapped to the hypervisor if executed at that lower level of privilege.

For an Arm trace macrocell, UNPREDICTABLE means that the behavior of the macrocell cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is UNPREDICTABLE. UNPREDICTABLE behavior can affect the behavior of the entire system, because the trace macrocell can cause the core to enter Debug state, and external outputs can be used for other purposes.

Note

In issue A of this document, UNPREDICTABLE also meant an UNKNOWN value.

When UNPREDICTABLE appears in body text, it is always in SMALL CAPITALS.

W1C Hardware must implement the bit as follows:

- Writing a `0b1` to the bit clears the bit to `0b0`.
- Writing a `0b0` to the bit has no effect.

Warm reset Also known as a core reset. Initializes most of the processor functionality, excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

See also Cold reset.

WI Hardware must ignore writes to the field. Software can rely on writes being ignored. This description can apply to a single bit, or to a field.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned A data item having a memory address that is divisible by four.