ARM® CoreSight™ Architecture Specification

v3.0

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
ARM IHI 0029E (ID022717)

ARM CoreSight
Architecture Specification
v3.0

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.

Release Information

The following changes have been made to this document:

Change history

Date	Issue	Confidentiality	Change
29 September 2004	A	Non-Confidential	First release for v1.0.
24 March 2005	B	Non-Confidential	Second release for v1.0. Editorial changes and clarifications.
27 March 2012	C	Confidential	Limited beta release for v2.0.
26 September 2013	D	Non-Confidential	First release for v2.0.
27 February 2017	E	Non-Confidential	First release for v3.0.

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines `http://www.arm.com/about/trademarks/guidelines/index.php`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
ARM Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means “ARM or any of its subsidiaries as appropriate”.

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`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

ARM CoreSightArchitecture Specification v3.0

Preface

About this document x
Using this document xi
Conventions xiii
Additional reading xv
Feedback xvi

Part A CoreSight Architecture

Chapter A1 About the CoreSight Architecture

A1.1 Purpose of the CoreSight architecture A1-20
A1.2 Structure of the CoreSight architecture A1-21
A1.3 CoreSight component types A1-23
A1.4 CoreSight topology detection A1-24

Part B CoreSight Visible Component Architecture

Chapter B1 About the Visible Component Architecture

Chapter B2 CoreSight programmers' model

B2.1 About the programmers' model B2-32
B2.2 Component and Peripheral Identification Registers B2-38
B2.3 Component-specific registers for Class 0x9 CoreSight components B2-44
B2.4 Component-specific registers for Class 0xF CoreLink, PrimeCell, and system components B2-62

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B3 Topology Detection

B3.1 About topology detection B3-64
B3.2 Requirements for topology detection signals B3-65
B3.3 Combination with integration registers B3-66
B3.4 Interfaces that are not connected or implemented B3-67
B3.5 Variant interfaces B3-68
B3.6 Documentation requirements for topology detection registers B3-69

Part C CoreSight Reusable Component Architecture

Chapter C1 About the Reusable Component Architecture

Chapter C2 AMBA APB and ATB Interfaces

C2.1 AMBA APB interface C2-76
C2.2 AMBA ATB interface C2-78

Chapter C3 Event Interface

Chapter C4 Channel interface

C4.1 About the channel interface C4-82
C4.2 Channels C4-84
C4.3 Channel interface signals C4-85
C4.4 Channel connections C4-86
C4.5 Synchronous and asynchronous conversions C4-87

Chapter C5 Authentication Interface

C5.1 About the authentication interface C5-90
C5.2 Definitions of Secure, hypervisor, and invasive debug C5-91
C5.3 Authentication interface signals C5-92
C5.4 Authentication rules C5-93
C5.5 User mode debugging C5-97
C5.6 Control of the authentication interface C5-98
C5.7 Exemptions from implementing the authentication interface C5-99

Chapter C6 Timestamp Interface

Chapter C7 Topology Detection at the Component Level

C7.1 About topology detection at the component level C7-104
C7.2 Interface types for topology detection C7-105
C7.3 Interface requirements for topology detection C7-107
C7.4 Signals for topology detection C7-108

Part D CoreSight System Architecture

Chapter D1 About the System Architecture

Chapter D2 System Considerations

D2.1 Clock and power domains D2-114
D2.2 Control of authentication interfaces D2-115
D2.3 Memory system design D2-116

Chapter D3 Physical Interface

D3.1 ARM JTAG 20 ..... D3-120
D3.2 CoreSight 10 and CoreSight 20 connectors D3-122
D3.3 ARM MICTOR D3-126

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4 Target Connector Signal details D3-131

Chapter D4 **Trace Formatter**
D4.1 About trace formatters D4-136
D4.2 Frame descriptions D4-137
D4.3 Modes of operation D4-142
D4.4 Flush of trace data at the end of operation D4-143

Chapter D5 **About ROM Tables**
D5.1 ROM Tables Overview D5-146
D5.2 ROM Table Types D5-147
D5.3 Component and Peripheral ID Registers for ROM Tables D5-148
D5.4 The component address D5-149
D5.5 Location of the ROM Table D5-150
D5.6 ROM Table hierarchies D5-151

Chapter D6 **Class 0x1 ROM Tables**
D6.1 About Class 0x1 ROM Tables D6-156
D6.2 Class 0x1 ROM Table summary D6-157
D6.3 Use of power domain IDs D6-159
D6.4 Register Descriptions D6-161

Chapter D7 **Class 0x9 ROM Tables**
D7.1 About Class 0x9 ROM Tables D7-170
D7.2 Class 0x9 ROM Table summary D7-171
D7.3 Use of power domain IDs D7-174
D7.4 Reset control D7-180
D7.5 Register descriptions D7-182

Chapter D8 **Topology Detection at the System Level**
D8.1 About topology detection at the system level D8-212
D8.2 Detection D8-213
D8.3 Components that are not recognized D8-214
D8.4 Detection algorithm D8-215

Chapter D9 **Compliance Requirements**
D9.1 About compliance classes D9-218
D9.2 CoreSight debug D9-219
D9.3 CoreSight trace D9-221
D9.4 Multiple DPs D9-224

Part E **Appendixes**

Appendix E1 **Power Requestor**
E1.1 About the power requestor E1-230
E1.2 Register descriptions E1-231
E1.3 Powering non-visible components E1-248

Appendix E2 **Revisions**

Appendix E3 **Pseudocode Definition**
E3.1 About ARM pseudocode E3-252
E3.2 Data types E3-253
E3.3 Expressions E3-257
E3.4 Operators and built-in functions E3-259
E3.5 Statements and program structure E3-264

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *CoreSight Architecture Specification*. It contains the following sections:

- *About this document* on page x.
- *Using this document* on page xi.
- *Conventions* on page xiii.
- *Additional reading* on page xv.
- *Feedback* on page xvi.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

About this document

This document describes the CoreSight architecture that all versions of the CoreSight compliant cores, components, platforms, and systems use.

Intended audience

This specification targets the following audiences:

- Hardware engineers integrating CoreSight components into a CoreSight system.
- Hardware engineers designing CoreSight components.
- Software engineers writing development tools that support CoreSight system functionality.
- Designers of debug hardware that is used to connect to a CoreSight system, for example JTAG emulators, SWD emulators, and Trace Port Analyzers.
- Advanced designers of development tools that support CoreSight functionality.

This specification does not document the behavior of individual components unless they form a fundamental part of the architecture.

ARM recommends that engineers who use this document have experience of the ARM architecture.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Using this document

This document is organized into the following parts:

Part A, CoreSight Architecture

Part A contains an introduction to the CoreSight architecture. It contains the following chapter:

Chapter A1 ***About the CoreSight Architecture***

Read this chapter for an outline description of the components, memory maps, clock and reset requirements, system integration, and the test interface.

Part B, CoreSight Visible Component Architecture

Part B describes the CoreSight visible component architecture, which must be implemented by all CoreSight components that are visible to a debugger. It contains the following chapters:

Chapter B1 ***About the Visible Component Architecture***

Read this chapter for a description of the visible component architecture.

Chapter B2 ***CoreSight programmers' model***

Read this chapter for a description of the CoreSight technology programmers' model.

Chapter B3 ***Topology Detection***

Read this chapter for a description of the topology detection registers in CoreSight systems.

Part C, CoreSight Reusable Component Architecture

Part C describes the CoreSight reusable component architecture, which must be implemented by CoreSight components so that they can be used with other CoreSight components. It contains the following chapters:

Chapter C1 ***About the Reusable Component Architecture***

Read this chapter for a description of the reusable component architecture.

Chapter C2 ***AMBA APB and ATB Interfaces***

Read this chapter for a description of the AMBA® APB interface and the AMBA ATB interface.

Chapter C3 ***Event Interface***

Read this chapter for a description of the event interface.

Chapter C4 ***Channel interface***

Read this chapter for a description of the channel interface.

Chapter C5 ***Authentication Interface***

Read this chapter for a description of the authentication interface.

Chapter C6 ***Timestamp Interface***

Read this chapter for a description of the timestamp interface.

Chapter C7 ***Topology Detection at the Component Level***

Read this chapter for a description of topology detection at the component level.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part D, CoreSight System Architecture

Part D describes the CoreSight system architecture, which must be implemented by all CoreSight systems and provides information that is required by debuggers to enable them to use CoreSight systems. It contains the following chapters:

Chapter D1 ***About the System Architecture***

Read this chapter for a description of the CoreSight system architecture.

Chapter D2 ***System Considerations***

Read this chapter for a description of system design with the CoreSight system architecture.

Chapter D3 ***Physical Interface***

Read this chapter for a description of the physical interface for CoreSight connection to a debugger.

Chapter D4 ***Trace Formatter***

Read this chapter for a description of the CoreSight trace formatter.

Chapter D5 ***About ROM Tables***

Read this chapter for a general description of CoreSight ROM Tables.

Chapter D6 ***Class 0x1 ROM Tables***

Read this chapter for a description of Class 0x1 ROM Tables.

Chapter D7 ***Class 0x9 ROM Tables***

Read this chapter for a description of Class 0x9 ROM Tables.

Chapter D8 ***Topology Detection at the System Level***

Read this chapter for a description of topology detection at the system level.

Chapter D9 ***Compliance Requirements***

Read this chapter for a description of the criteria that systems must comply with to satisfy CoreSight requirements.

Part E, Appendixes

This specification contains the following appendixes:

Appendix E1 ***Power Requestor***

Read this chapter for a description of the power requestor.

Appendix E2 ***Revisions***

Read this chapter for a description of the technical changes between released versions of this specification.

Appendix E3 ***Pseudocode Definition***

Read this chapter for a definition of the pseudocode conventions that are used in this specification.

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

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Conventions

The following sections describe conventions that this document can use:

- *Typographic conventions*.
- *Signals*.
- *Timing diagrams*.
- *Numbers* on page xiv.
- *Pseudocode descriptions* on page xiv.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes internal cross-references and citations, or highlights an important note.

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, for example: `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of referenced information that is not on the current page, for example *Numbers* on page xiv.
- 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 *Embedded Trace Buffer (ETB)* .

Signals

In general, this document does not define processor signals, but it does include some signal examples and recommendations. 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.

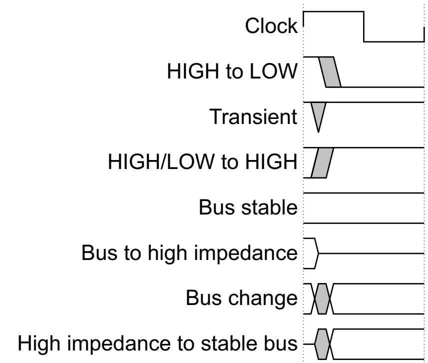
Lowercase n At the start or end of a signal name denotes an active-LOW signal.

Timing diagrams

The figure that is named *Key to timing diagram conventions* on page xiv explains the components that are used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

In shaded areas, bus and signal states are undefined and can assume any value. The actual level is irrelevant for normal operation.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential



Key to timing diagram conventions

Numbers

Numbers are normally written in decimal notation. 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 document 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 E3 *Pseudocode Definition*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 specification contains information that is specific to CoreSight. See the following documents for other relevant information:

- *ARM® CoreSight™ SoC-400 Technical Reference Manual* (ARM 100536).
- *ARM® CoreSight™ Technology System Design Guide* (ARM DGI 0012).
- *ARM® Embedded Trace Macrocell Architecture Specification, ETM v1.0 to ETMv3.5* (ARM IHI 0014).
- *ARM® ETM™ Architecture Specification, ETMv4* (ARM IHI 0064).
- *ARM® Debug Interface Architecture Specification v6* (ARM IHI 0074).
- *ARM® Debug Interface Architecture Specification v5* (ARM IHI 0031).
- *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).
- *ARM® AMBA® AXI and ACE Protocol Specification* (ARM IHI 0022).
- *ARM® AMBA® APB Protocol Specification* (ARM IHI 0024).
- *ARM® AMBA® 4 ATB Protocol Specification* (ARM IHI 0032).

Other publications

This section lists relevant documents that are published by third parties:

- IEEE, *Standard Test Access Port and Boundary Scan Architecture*, IEEE Std 1149.1-1990.
- JEDEC, *Standard Manufacturer's Identification Code*, JEP106.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, ARM IHI 0029E.
- 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part A
CoreSight Architecture

Chapter A1
About the CoreSight Architecture

This chapter introduces the CoreSight architecture. It contains the following sections:

- *Purpose of the CoreSight architecture* on page A1-20.
- *Structure of the CoreSight architecture* on page A1-21.
- *CoreSight component types* on page A1-23.
- *CoreSight topology detection* on page A1-24.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A1.1 Purpose of the CoreSight architecture

The CoreSight architecture provides a system-wide solution for real-time debug and collecting trace information. It addresses the following:

- The requirement for multi-core debug and trace.
- The requirement to debug and trace system components beyond the core, for example buses.
- The requirement for sharing resources, such as pins and trace storage, between debug and trace components, to reduce silicon costs.
- The requirement for debug and trace components from multiple vendors to be able to work together.
- The requirement for minimizing pin count.
- The requirement for supporting increasing trace bandwidth from many sources.
- The requirement to accommodate the existing trace solutions, rather than expecting them to be rewritten to support a new trace architecture.
- The requirement for development tools to automatically identify and configure themselves for different systems.
- The requirement for controlling access to debug and trace functionality in the field.
- The fact that the clock and power to parts of the system can be varied or disabled independently while debugging the rest of the system.
- The fact that the time available to design debug and trace functionality is often limited and the number of options must be minimized where possible.
- The requirement for debug monitors and other on-chip debug software to have access to the same debug and trace functionality as an external debugger.
- The fact that systems are often built from a hierarchy of reusable platforms, where each level must hide its internal complexities, which prevents designers from changing the platform when using it in another system.

The CoreSight architecture satisfies the following requirements for debug and trace:

- To access debug functionality without software interaction.
- To connect to a running system without performing a reset.
- To perform certain operations, such as real-time tracing, noninvasively, with no effect on the behavior of the system.
- To access noninvasive functionality noninvasively.
- To minimize power consumption of debug logic when it is not in use.
- To capture trace over a large period.

The CoreSight architecture can be used to design CoreSight components, that can be combined with other CoreSight components and processors that comply with the CoreSight architecture to make up a CoreSight system.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A1.2 Structure of the CoreSight architecture

The CoreSight architecture comprises:

- A visible component architecture.
- A reusable component architecture.
- The system design.

A1.2.1 Visible component architecture

The CoreSight visible component architecture specifies what a component must implement to comply with the CoreSight architecture.

The visible component architecture is visible to the programming interface and to tools that access the device. All CoreSight components must comply with the visible component architecture. The visible component architecture specifies:

- The requirements of the programmers' model that all CoreSight components must conform to.
- The requirements for topology detection that enable discovery of the component layout.

For details of the visible component architecture, see Part B *CoreSight Visible Component Architecture*.

A1.2.2 Reusable component architecture

The CoreSight reusable component architecture specifies the rules that the implementation of the physical interface of a CoreSight component must follow to work correctly with other reusable CoreSight components.

The reusable component architecture specifies:

- The AMBA APB interface for access to the registers in CoreSight components.
- The AMBA ATB interface for trace data transfer between CoreSight components.
- The channel interface for the communication of trigger events between CoreSight components.
- The authentication interface for control of access for debug.
- Topology detection infrastructure that specifies the signals that must be controlled at each interface.

It is possible to create a homogeneous component that performs various functions internally as separate components implementing the visible component architecture, provided one set of reusable component interfaces is available to enable integration into a larger CoreSight system.

Caution

Self-contained systems that implement only the visible, and not the reusable component architecture, are compatible with development tools, but have the following limitations:

- Integrating them with other CoreSight components might be impossible.
- Detecting them during topology detection might fail.

For details about the reusable component architecture, see Part C *CoreSight Reusable Component Architecture*.

A1.2.3 System architecture

The following CoreSight architectural requirements ensure seamless integration of elements that comply with the CoreSight architecture:

- System-level requirements for:
  - Clock and power domains visible to debuggers.
  - Control of the authentication interface.
  - Distinction between external and internal accesses through the AMBA APB interface memory map.
- The requirements for the physical interface to the debugger.
- The format that is used by the trace formatter. See *CoreSight component types* on page A1-23 and Chapter D4 *Trace Formatter*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- The design of the ROM Table. See *CoreSight component types* on page A1-23 and Chapter D5 *About ROM Tables*.
- How to enable CoreSight topology detection.
- Compliance requirements for CoreSight systems.

For details about the system architecture, see Part D *CoreSight System Architecture*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A1.3 CoreSight component types

The CoreSight architecture specifies a set of components for implementing specific SoC subsystems that support collection of debug and trace information. This section shows some example implementations of CoreSight components that are based on the CoreSight architecture.

Note

A CoreSight component is a component that implements the CoreSight visible component architecture.

The main elements are:

Control components

CoreSight systems can include *Embedded Cross Trigger* (ECT) control components. The ECT includes:

- A *Cross Trigger Interface* (CTI).
- A *Cross Trigger Matrix* (CTM).

Trace sources

CoreSight systems can include the following trace sources:

- *Embedded Trace Macrocells* (ETMs).
- *AMBA Trace Macrocells*.
- *Program Flow Trace Macrocells* (PTMs).
- *System Trace Macrocells* (STMs).

Trace links

CoreSight systems can include the following trace links:

- Trace funnels.
- Replicators.
- ATB bridges.

Trace sinks

CoreSight systems can include the following trace sinks:

- *Trace Port Interface Units* (TPIUs).
- *Embedded Trace Buffers* (ETBs).
- *Trace Memory Controllers* (TMCs).

Each trace sink can include a Trace Formatter.

Debug Ports and Access Ports

Debug Ports and Access Ports provide access to CoreSight components and other system features. Debug Ports and Access Ports are described by the ARM Debug Interface Architecture Specification, see *ARM® Debug Interface Architecture Specification ADIv5.0-5.2* and *ARM® Debug Interface Architecture Specification ADIv6.0.*

A Debug Port provides a mechanism that is specific to a wire protocol, and enables access to various components, including Access Ports. Some examples of common Debug Ports are:

- A *Serial Wire Debug Port* (SW-DP).
- A *JTAG Debug Port* (JTAG-DP).
- A *Serial Wire JTAG Debug Port* (SWJ-DP).

An Access Port provides a mechanism to access buses or other protocols, in particular to access CoreSight components. Some examples of common Access Ports are:

- An *APB Access Port* (APB-AP).
- An *AHB Access Port* (AHB-AP).
- An *AXI Access Port* (AXI-AP).
- A *JTAG Access Port* (JTAG-AP).

For more information on specific components, see the appropriate Technical Reference Manual.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4 CoreSight topology detection

Depending on the requirements of the system, CoreSight components can be connected together in many different ways. Debuggers use a process that is called topology detection to detect the component connections. The infrastructure for topology detection is reflected at each of the following levels:

- A visible component architecture.
- A reusable component architecture.
- The system design.

CoreSight systems can have several interface types, as masters or slaves, and each CoreSight component specifies which interfaces are present. The debugger probes each interface to determine which other components are connected to it.

Each interface type defines which signals must be controllable by the master and slave interfaces, and how the debugger can determine the connectivity using these signals. These signals are referred to as topology detection signals. For the specification of the requirements for standard interfaces that are used by ARM CoreSight components, see Chapter C7 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter D8 *Topology Detection at the System Level*.

A1.4.1 Basic topology detection infrastructure

This section describes the topology detection infrastructure in a bottom-up fashion, from the visible component level to the system level.

At the visible component architecture level, a CoreSight system provides topology detection registers. These registers are accessible through the programmers' model and contain information about the components in the system and permit a debugger to identify the components. See Chapter B3 *Topology Detection*.

At the reusable component architecture level, the system defines interfaces that enable communication between the various components and enable debuggers access to the system. See Chapter C7 *Topology Detection at the Component Level*.

At the system level there is:

- A hierarchy of one or more ROM Tables that describe the address map for the CoreSight system. See Chapter D5 *About ROM Tables*.
- A description of physical connections for the debugger hardware. See Chapter D3 *Physical Interface*.

There are registers to control the wires where buses exist, and enough of the system must be controllable to establish the existence of the link. For example, for ATB interface signals, only **ATVALID** and **ATREADY** must be controlled.

A1.4.2 Mechanism for topology detection

Topology detection is only required when the debugger does not already have information about the system being debugged.

Before it performs topology detection, the debugger uses the following procedure to determine which components are present in the system:

- It connects to the physical interface. See Chapter D3 *Physical Interface*.
- It establishes communication with the system, for example through an interface that complies with the *ARM Debug Interface* (ADI) architecture.
- It uses the ROM Table to determine which components are present.

The debugger continues with the following steps:

- For each component, it uses the component type to determine which interfaces are present and how to access signals on these interfaces.
- For each interface, it uses the interface type to determine which signals to access. Chapter C7 *Topology Detection at the Component Level* describes how to perform topology detection for each of the CoreSight interfaces.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- It uses the algorithm that is described in Chapter D8 *Topology Detection at the System Level* to perform topology detection, which asserts and deasserts signals on each master interface in turn to check each slave interface and determine where interfaces are connected together.
- It resets the system and saves the description.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

 Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part B
CoreSight Visible Component Architecture

Chapter B1
About the Visible Component Architecture

The visible component architecture specifies aspects of components that are visible to the programming interface and to tools that access the device.

The visible component architecture is described in the following chapters:

- Chapter B2 *CoreSight programmers' model*. The programmers' model specifies various registers for the identification and control of the component.
- Chapter B3 *Topology Detection*. The topology detection registers provide the means for the process of topology detection in the CoreSight system.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B2
CoreSight programmers' model

This chapter describes the CoreSight programmers' model. It contains the following sections:

- *About the programmers' model* on page B2-32.
- *Component and Peripheral Identification Registers* on page B2-38.
- *Component-specific registers for Class 0x9 CoreSight components* on page B2-44.
- *Component-specific registers for Class 0xF CoreLink, PrimeCell, and system components* on page B2-62.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.1 About the programmers' model

This section has the following goals:

- Define the standard set of registers that every CoreSight component must implement in addition to the control registers specific to that component.
- Explain how software can use integration registers for determining the topology of a CoreSight system.

It contains the following subsections:

- *Basic structure of the programmers' model*
- *The Unique Component Identifier* on page B2-33.
- *Conventions for registers with less than 32 valid bits* on page B2-34
- *Components that occupy more than 4KB of address space* on page B2-34
- *Programmers' Model Quick Reference* on page B2-36

B2.1.1 Basic structure of the programmers' model

The basic register structure is outlined in Table B2-3 on page B2-36. The structure is based on the Peripheral ID Register structure for ARM CoreLink components, and comprises a set of word-aligned 32-bit registers that can be divided into the following categories:

- Component and Peripheral Identification Registers:
  - A *Component Identification Register*, which extends the original CoreLink specification with a component class that indicates whether extra registers are present. For a description of the requirements for the Component Identification Register, see *Component and Peripheral Identification Registers* on page B2-38.
  - A *Peripheral Identification Register* that uniquely identifies the component. For a description of the requirements for the Peripheral Identification Register, see *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.
- Component-specific control registers. The set of required control registers depends on the component class. For a list of valid components classes, see the description of the PIDR3.CLASS field in *CIDR0-CIDR3, Component Identification Registers*.

This register structure must be supported by every component that implements a CoreSight compliant programmers' model.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.1.2 The Unique Component Identifier

To ensure that a debugger can identify a component, unique components must have a Unique Component Identifier. A Unique Component Identifier is a unique combination of values that are assigned to fields from the Component Identification Registers, the Peripheral Identification Registers, and, if implemented, several component-specific registers. The fields that can contribute to the Unique Component Identifier are listed in Table B2-1. For details about the individual fields, see the field descriptions in the relevant register descriptions.

Table B2-1 Register fields that contribute to the Unique Component Identifier

Register Type	Register	Fields	Size
Component Identification Register	CIDR1	CLASS	4 bits
Peripheral Identification Registers	PIDR0	PART_0	8 bits
	PIDR1	DES_0	4 bits
		PART_1	4 bits
	PIDR2	REVISION	4 bits
		DES_1	3 bits
	PIDR3	REVAND	4 bits
	PIDR4	DES_2	4 bits
Component-specific registers	DEVARCH[a]	ARCHITECT	11 bits
		REVISION	4 bits
		ARCHID	16 bits
	DEVTYPE[a]	SUB	4 bits
		MAJOR	4 bits

a. These registers only contribute to the Unique Component Identifier of components with a CIDR1.CLASS value of 0x9.

The following rules apply to the Unique Component Identifier:

- A common function is defined as a cluster of homogeneous processor components. An example of a common function is a set of debug control registers, a performance monitor unit (PMU), a cross-trigger-interface (CTI), a trace macrocell (ETM), and a ROM Table describing the layout of these components.

  The following rules apply to components that are part of a common function:

  — ARM recommends that the values of PIDR0.PART_0 and PIDR1.PART_1 are different for components that are not part of the same common function.

  — Although components that are part of the same common function can share the value of PIDR0.PIDR0.PART_0 and PIDR1.PART_1, each component must have its own Unique Component Identifier.

  — Because CIDR1.CLASS is part of the Unique Component Identifier, CoreSight version 3.0 permits ROM Tables that are part of a common function to share the part number as the other components of different classes that are part of the same the common function, and to use CIDR1.CLASS to distinguish between them.

- Multiple instances of the same component are not considered to be unique and usually have the same Unique Component Identifier.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a component has multiple possible configurations and each configuration is a subset of one single configuration, it is not necessary for each configuration to have a separate Unique Component Identifier. For example, a trace macrocell that has a configurable number of comparators does not need a separate Unique Component Identifier for each configuration with a different number of comparators.
- If a component in a subsystem that is described by a ROM Table is changed, the revision number in its Unique Component Identifier must be changed, even if the revision number that is part of the Unique Component Identifier of the ROM Table is not changed. See also *Component Revision Numbers* on page D5-151.

Note

- If multiple components share the value for the part number in PIDR0.PART_0 and PIDR1.PART_1, they are differentiated by the different values of the other fields in the Unique Component Identifier.
- Component designers who require more than 12 bits for the part number, for example when using a 16-bit part numbering scheme, can use the PIDR2.REVISION and PIDR3.REVAND fields to indicate the part number. Effectively, the PIDR0.PART_0, PIDR1.PART_1, PIDR2.REVISION, and PIDR3.REVAND fields provide a total of 20 bits that can be used to create part numbers and revisions of the component.
- CoreSight versions before version 3.0 permitted using the PIDR3.CMOD field to distinguish between different components. This permission is removed in CoreSight version 3.0.

B2.1.3 Conventions for registers with less than 32 valid bits

The CoreSight programmers' model presents registers as a set of word-aligned registers, which means that every register occupies exactly one word (32-bits), regardless of its information content. The following convention is used in cases where a register has less than 32 bits of valid information:

- Valid information is stored in the least significant bits of the register.
- The most significant bits of the register are reserved, with access permissions that depend on the register.
- When accessed as a 32-bit register, all registers are accessed in little-endian format.

Note

Although a component can be designed to implement only the valid bits of a CoreSight register, software can always access the register as a 32-bit register.

B2.1.4 Components that occupy more than 4KB of address space

The memory layout that is shown in Table B2-3 on page B2-36 covers all components that are contained in a single 4KB block of address space. If the registers that are defined for a component, including the 256 bytes reserved for CoreSight management registers, do not fit within 4KB, it is necessary to allocate extra address space. The following rules apply:

- Extra address space must be allocated in 4KB blocks, and the total number of blocks must be a power of 2. The maximum number of blocks is 16,384, for an address space of 64MBytes.
- All 4KB blocks comprising a component must be allocated as a contiguous segment, without gaps.
- The CoreSight programmers' model must be implemented in one of the 4KB blocks making up the component. It is not necessary to implement the programmers' model in the other 4KB blocks that are part of the same component.

  Note

  From CoreSight version 3.0 onwards, the CoreSight programmers' model does not need to be implemented in the last 4KB block of each individual component.

- ARM recommends that debug tools determine the size of the component from the part number in the Peripheral ID registers and other IMPLEMENTATION DEFINED registers in the component.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

From v3.0 onwards, using the PIDR4.SIZE field to indicate the size of the component is deprecated:

— ARM recommends that PIDR4.SIZE is always `0x0`. The PIDR4.SIZE field might not correctly indicate the size of the component.

— ARM recommends that debug tools ignore the value of PIDR4.SIZE.

For more information about PIDR4, see *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

- The bus that is used to access the component must have enough address lines to span the entire allocated address space. See Table B2-2.

The following alternative methods for extending the address space available to a component are possible, but not recommended:

Implementing a second CoreSight component

The address space can be extended by implementing an additional CoreSight component. Doing so, however, requires a method for linking this component back to the original component through topology detection, and is not recommended.

Implementing an extra linked address space

The address space can be extended with the address space of a component with an area that is not used by any CoreSight components that were designed according to the programmers' model. Doing so, however, requires a means for determining the address of the extended address space in the programmers' model of the component, which limits the system design options and is therefore not recommended.

Table B2-2 summarizes the relationship between the address space size, the number of available registers, and the number of required address lines.

Table B2-2 Spanning multiple 4KB windows

Number of 4KB blocks	Total memory window used	Component-specific registers available	Expected PADDRDBG input[a]
1	4KB, 1K words	960 words	**PADDRDBG[11:2]**
2	8KB	1984 words	**PADDRDBG[12:2]**
4	16KB, 4K words	4032 words	**PADDRDBG[13:2]**
8	32KB, 8K words	8128 words	**PADDRDBG[14:2]**
16	64KB	16320 words	**PADDRDBG[15:2]**
32	128KB	32704 words	**PADDRDBG[16:2]**
64	256KB, 64K words	65472 words, 63.94K words	**PADDRDBG[17:2]**
128	512KB	127.94K words	**PADDRDBG[18:2]**
256	1MB, 256K words	255.9K words	**PADDRDBG[19:2]**
512	2MB	~512K words	**PADDRDBG[20:2]**
1024	4MB	~1M words	**PADDRDBG[21:2]**
2048	8MB	~2M words	**PADDRDBG[22:2]**
4096	16MB	~4M words	**PADDRDBG[23:2]**
8192	32MB	~8M words	**PADDRDBG[24:2]**
16384	64MB, 16M words	~16M words	**PADDRDBG[25:2]**
Reserved	-	-	-

a. This table uses the AMBA APB protocol as an example protocol used to access a component, where the **PADDRDBG** bus is the address bus that is used to select the component registers. **PADDRDBG[1:0]** are not required on components because all transfers are 32-bit word-aligned.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.1.5 Programmers' Model Quick Reference

Table B2-3 shows the address offsets for the CoreSight component registers, in order of their offset in the 4KB block.

Table B2-3 CoreSight component register address offsets

Offset	Type	Name	Description	
0xF00	RW	ITCTRL	Integration Mode Control Register	
0xF04-0xF9C	RES0	-	Reserved	
0xFA0	RW	CLAIMSET	Claim Tag Set Register	Claim Tag Registers
0xFA4	RW	CLAIMCLR	Claim Tag Clear Register	
0xFA8	RO	DEVAFF0	Device Affinity Registers	
0xFAC	RO	DEVAFF1		
0xFB0	WO	LAR	Software Lock Access Register	Software Lock Registers
0xFB4	RO	LSR	Software Lock Status Register	
0xFB8	RO	AUTHSTATUS	Authentication Status Register	
0xFBC	RO	DEVARCH	Device Architecture Register	
0xFC0	RO	DEVID2	Device Configuration Register 2	
0xFC4	RO	DEVID1	Device Configuration Register 1	
0xFC8	RO	DEVID	Device Configuration Register	
0xFCC	RO	DEVTYPE	Device Type Identifier Register	
0xFD0	RO	PIDR4	Component size (deprecated) and JEP106 identification	Peripheral Identification Registers
0xFD4	RO	PIDR5		
0xFD8	RO	PIDR6		
0xFDC	RO	PIDR7		
0xFE0	RO	PIDR0	Part number	
0xFE4	RO	PIDR1	JEP106 identification and Part number	
0xFE8	RO	PIDR2	Revision and JEP106 identification	
0xFEC	RO	PIDR3	RevAnd and Customer modified	
0xFF0	RO	CIDR0	Preamble	Component Identification Registers
0xFF4	RO	CIDR1	Component class and Preamble	
0xFF8	RO	CIDR2	Preamble	
0xFFC	RO	CIDR3	Preamble	

The words at offsets `0xF00-0xFFC` are the CoreSight management registers. These registers are common to all CoreSight components. This area is reserved for CoreSight registers, and device-specific control registers must not use it. The CoreSight management registers include:

- The Peripheral ID Registers, at offsets `0xFD0-0xFEC`.
- The Component ID Registers, at offsets `0xFF0-0xFFC`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The remaining words can be used for component-specific registers. ARM recommends using the following conventions:

- Control registers start at address `0x000` and continue upwards.
- Any registers that are used purely for integration purposes start at address `0xEFC` and continue downwards.

The register type of device-specific registers is IMPLEMENTATION DEFINED and can be RW, RO, or WO.

Table B2-4 defines the behavior on accesses to reserved registers and fields.

Table B2-4 Behavior on accesses to reserved registers and fields

Access to	Behavior
Reserved registers	RES0
Unimplemented registers	RAZ/WI
Reserved fields in registers	RES0 or RES1
Unimplemented fields in registers	RES0 or RES1
Unimplemented bits in implemented fields	RAZ/WI

Locations that are marked as RES0 are reserved. Reads of write-only registers are considered accesses to Reserved registers. Writes to read-only registers are considered accesses to Reserved registers. The following tables show the specific meaning of each of the behaviors.

Table B2-5 shows the required behavior of a CoreSight component, for registers that are defined as RW, RO, or WO.

Table B2-5 CoreSight component behavior

Behavior	Component behavior on reads			Component behavior on writes		
	RW	RO	WO	RW	RO	WO
RES0	RAZ	RAZ	RAZ	WI	WI	WI
RES1	RAO	RAO	RAO	WI	WI	WI
RAZ/WI	RAZ	RAZ	RAZ	WI	WI	WI

Table B2-6 shows the required behavior of software when accessing a CoreSight component, for registers that are defined as RW, RO, or WO.

Table B2-6 Software behavior

Behavior	Software behavior on reads			Software behavior on writes		
	RW	RO	WO	RW	RO	WO
RES0	Treat as UNKNOWN	Treat as UNKNOWN	Do not read	Preserve	Do not write	Preserve
RES1	Treat as UNKNOWN	Treat as UNKNOWN	Do not read	Preserve	Do not write	Preserve
RAZ/WI	Expect zero	Expect zero	Do not read	Are ignored	Do not write	Are ignored

Programming a reserved value into a register, or a field within a register, might result in CONSTRAINED UNPREDICTABLE behavior of the component, and usually involves mapping the behavior to one or more permitted behaviors.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2 Component and Peripheral Identification Registers

This section describes the following registers:

- *CIDR0-CIDR3, Component Identification Registers*.
- *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

B2.2.1 CIDR0-CIDR3, Component Identification Registers

The CIDR characteristics are:

Purpose

Provide information that can be used to identify a CoreSight component.

Usage constraints

CIDR0-CIDR3 are accessible as follows:

Default
RO

Configurations

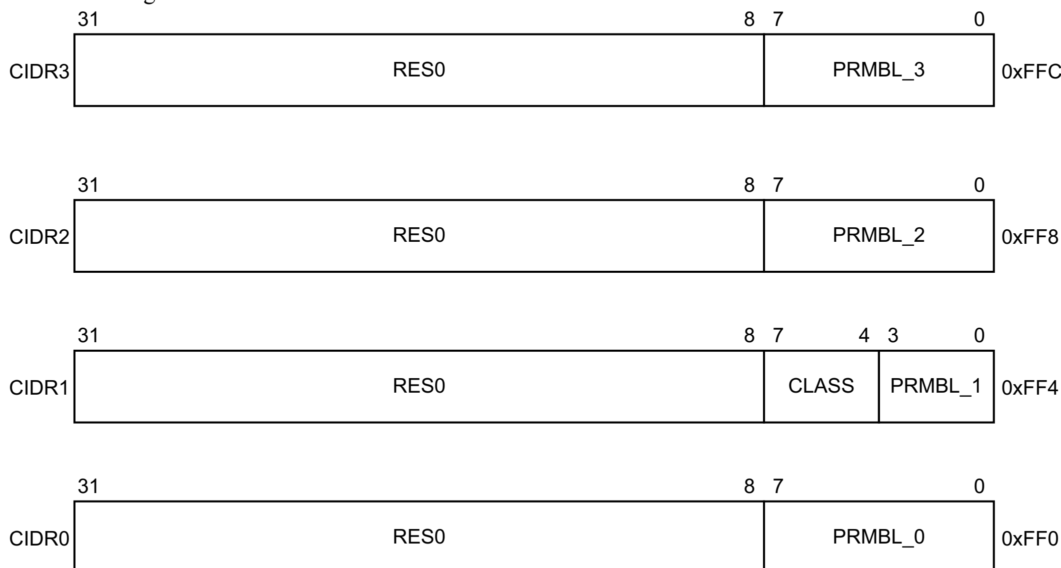
Included in all implementations.

Attributes

CIDR0-CIDR3 are four 32-bit management registers.

Field Descriptions

The CIDR bit assignments are:



Bits[31:8] of CIDR3

RES0.

PRMBL_3, CIDR3 bits[7:0]

Preamble, segment 3. Must be `0xB1`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Bits[31:8] of CIDR2

RES0.

PRMBL_2, CIDR2 bits[7:0]

Preamble, segment 2. Must be `0x05`.

Bits[31:8] of CIDR1

RES0.

CLASS, CIDR1 bits[7:4]

The component class, which can be one of the values that are listed in Table B2-7.

Table B2-7 CLASS field encodings

Value	Description
`0x0`	Generic verification component.
`0x1`	ROM Table. See *ROM Table Types* on page D5-147.
`0x2-0x8`	Reserved.
`0x9`	CoreSight component. See *Component-specific registers for Class 0x9 CoreSight components* on page B2-44.
`0xA`	Reserved.
`0xB`	Peripheral Test Block.
`0xC-0xD`	Reserved.
`0xE`	Generic IP component.
`0xF`	CoreLink, PrimeCell, or system component with no standardized register layout, for backwards compatibility. See *Component-specific registers for Class 0xF CoreLink, PrimeCell, and system components* on page B2-62.

PRMBL_1, CIDR1 bits[3:0]

Preamble, segment 1. Must be `0x0`.

Bits[31:8] of CIDR0

RES0.

PRMBL_0, CIDR0 bits[7:0]

Preamble, segment 0. Must be `0x0D`.

Accessing CIDR

CIDR0-CIDR3 can be accessed at the following addresses:

Offset			
CIDR0	**CIDR1**	**CIDR2**	**CIDR3**
`0xFF0`	`0xFF4`	`0xFF8`	`0xFFC`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2.2 PIDR0-PIDR7, Peripheral Identification Registers

The PIDR characteristics are:

Purpose

Provide information that can be used to identify a CoreSight component. Most of the fields making up the PIDR are included in the Unique Component Identifier. The Unique Component Identifier can also include fields from the CIDR1, DEVARCH, and DEVTYPE registers. For details, see *The Unique Component Identifier* on page B2-33.

Usage constraints

PIDR0-PIDR7 are accessible as follows:

Default
RO

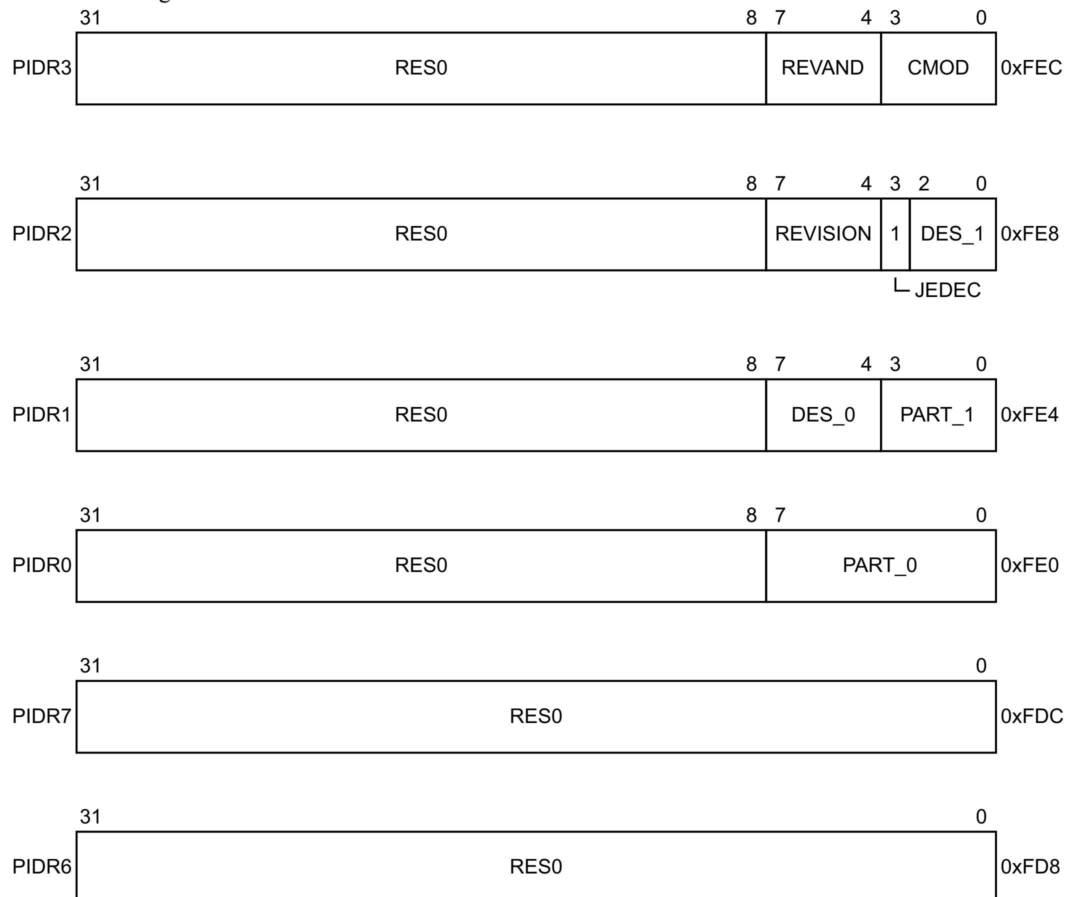
Configurations

Included in all implementations.

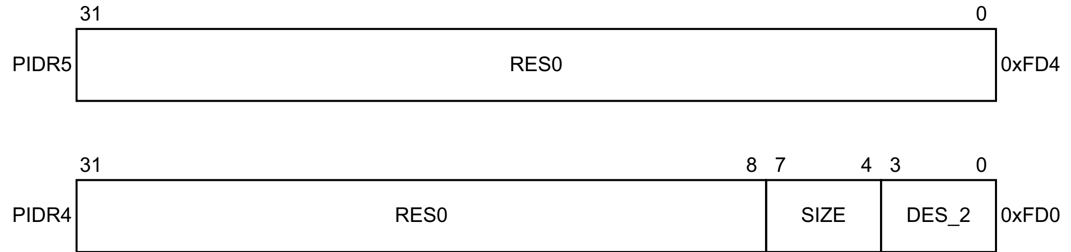
Attributes PIDR0-PIDR7 are eight 32-bit management registers.

Field Descriptions

The PIDR bit assignments are:



Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential



PIDR3 bits[31:8]

RES0.

REVAND, PIDR3 bits[7:4]

The REVAND field indicates minor errata fixes specific to this design, for example metal fixes after implementation. Usually this field is zero. If the field is required, ARM recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

Together with PIDR2.REVISION, PIDR3.REVAND forms the revision number of the component. When a component is changed, one or more of the fields making up the revision number must be changed to ensure that debug tools can differentiate the different versions of the component.

CMOD, PIDR3 bits[3:0]

Customer Modified. If the component is reusable IP, the CMOD field indicates whether the customer has modified the behavior of the component. CMOD can have one of the following values:

0x0 The component is not modified from the original design.

Any other value

The component has been modified.

ARM recommends that the user or debugger reads the documentation for the component to determine the modifications that are made to the component.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have been subjected to the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

See also *The Unique Component Identifier* on page B2-33.

Note

CoreSight versions before version 3.0 permitted using the CMOD field to distinguish between different components. This permission is removed in CoreSight version 3.0.

PIDR2 bits[31:8]

RES0.

REVISION, PIDR2 bits[7:4]

The REVISION field is an incremental value starting at `0x0` for the first design of a component. The value is increased by 1 for both major and minor revisions and is used as a look-up to establish the exact major and minor revision.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Together with PIDR3.REVAND, PIDR2.REVISION forms the revision number of the component. When a component is changed, one or more of the fields making up the revision number must be changed to ensure that debug tools can differentiate the different versions of the component.

JEDEC, PIDR2 bits[3]

Must be `0b1` to indicate that a JEDEC-assigned value is used.

DES_1, PIDR2 bits[2:0]

JEP106 identification and continuation codes, which are stored in PIDR1, PIDR2, and PIDR4 as follows:

DES_0, PIDR1 bits[7:4] JEP106 identification code bits[3:0].

DES_1, PIDR2 bits[2:0] JEP106 identification code bits[6:4].

DES_2, PIDR4 bits[3:0] JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC `http://www.jedec.org`.

A JEDEC code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

For example, ARM Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, for ARM Limited this code is `0x4`.
- The identification code is bits[6:0] of the final number. For example, ARM Limited has the code `0x3B`.

PIDR1 bits[31:8]

RES0.

DES_0, PIDR1 bits[7:4]

JEP106 identification and continuation codes, which are stored in PIDR1, PIDR2, and PIDR4 as follows:

DES_0, PIDR1 bits[7:4] JEP106 identification code bits[3:0].

DES_1, PIDR2 bits[2:0] JEP106 identification code bits[6:4].

DES_2, PIDR4 bits[3:0] JEP106 continuation code.

For details about the JEP106 codes, see the description of the PIDR2.DES1 field.

PART_1, PIDR1 bits[3:0]

Part number, which is selected by the designer of the component, and stored in PIDR0 and PIDR1 as follows:

PART_0, PIDR0 bits[7:0] Part number bits[7:0].

PART_1, PIDR1 bits[3:0] Part number bits[11:8].

PIDR0 bits[31:8]

RES0.

PART_0, PIDR0 bits[7:0]

Part number, which is selected by the designer of the component, and stored in PIDR0 and PIDR1 as follows:

PART_0, PIDR0 bits[7:0] Part number bits[7:0].

PART_1, PIDR1 bits[3:0] Part number bits[11:8].

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

PIDR7 bits[31:0]

RES0.

PIDR6 bits[31:0]

RES0.

PIDR5 bits[31:0]

RES0.

PIDR4 bits[31:8]

RES0.

SIZE, PIDR4 bits[7:4]

The SIZE field indicates the memory size that is used by this component. It is expressed as the logarithm to the base 2 of the number of 4KB blocks the component occupies. The value `0x0` indicates that either:

- The component uses a single 4KB block.
- The component uses an UNKNOWN number of 4KB blocks.

Using the SIZE field to indicate the size of the component is deprecated. The SIZE field might not correctly indicate the size of the component. ARM recommends that debug tools determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

DES_2, PIDR4 bits[3:0]

JEP106 identification and continuation codes, which are stored in PIDR1, PIDR2, and PIDR4 as follows:

DES_0, PIDR1 bits[7:4] JEP106 identification code bits[3:0].

DES_1, PIDR2 bits[2:0] JEP106 identification code bits[6:4].

DES_2, PIDR4 bits[3:0] JEP106 continuation code.

For details about the JEP106 codes, see the description of the PIDR2.DES1 field.

Accessing the PIDR

PIDR0-PIDR7 can be accessed at the following addresses:

Offset							
PIDR0	**PIDR1**	**PIDR2**	**PIDR3**	**PIDR4**	**PIDR5**	**PIDR6**	**PIDR7**
`0xFE0`	`0xFE4`	`0xFE8`	`0xFEC`	`0xFD0`	`0xFD4`	`0xFD8`	`0xFDC`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3 Component-specific registers for Class 0x9 CoreSight components

Components that have the value `0x9` assigned to the CIDR1.CLASS field in the Component Identification Register are CoreSight components. For details, see *CIDR0-CIDR3, Component Identification Registers* on page B2-38.

CoreSight components must implement an extra set of registers, referred to as the CoreSight management registers, which are described in this section. Addresses `0xF00` to `0xFCC` are reserved for use by CoreSight management registers.

When implementing a CoreSight component, ensure that the following requirements are met:

- Any reads from unimplemented or reserved registers in `0xF00` to `0xFFF` must return zero, and writes must be ignored. For details about the required behavior of reserved locations, see *Programmers' Model Quick Reference* on page B2-36.
- Two or more functionally different CoreSight components are permitted to share a part number, as long as they each have a different Unique Component Identifier. See *The Unique Component Identifier* on page B2-33.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.1 AUTHSTATUS, Authentication Status Register

The AUTHSTATUS characteristics are:

Purpose

Reports the required security level and status of the authentication interface. Where functionality changes on a given security level, the change in status must be reported in this register. For details about the authentication interface, see Chapter C5 *Authentication Interface*.

Usage constraints

Some components might not distinguish between Secure and Non-secure debug. For example, a trace component for a simple bus might connect to a Secure or a Non-secure bus, while its enable signals connect differently depending on which bus the component connects to. A failure to distinguish between Secure and Non-secure debug could result in:

- A component that indicates only Non-secure debug capabilities while performing only Secure debug functions.
- A component that indicates only Secure debug capabilities while performing only Non-secure debug functions.

Debuggers must be able to accommodate this possibility.

AUTHSTATUS is accessible as follows:

Default
RO

Configurations

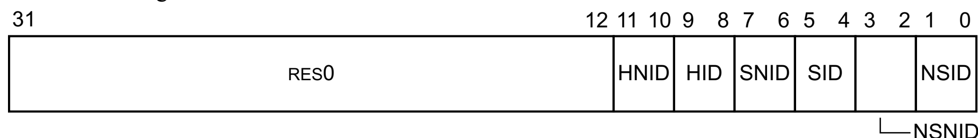
Included in all implementations.

Attributes

AUTHSTATUS is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The AUTHSTATUS bit assignments are:



Bits[31:12]

RES0.

HNID, bits[11:10]

Hypervisor non-invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor non-invasive debug are not implemented, or no hypervisor is implemented. For ARMv7 processors that implement the Virtualization Extensions, and for ARMv8 processors that implement EL2, if separate controls for hypervisor debug visibility are not implemented, the hypervisor debug visibility is indicated by the relevant Non-secure debug visibility fields NSNID and NSID. See the relevant ARM Architecture Manual for more information about Virtualization Extensions and EL2.

`0b10` Supported and disabled.
(**HIDEN** | **HNIDEN**) & (**DBGEN** | **NIDEN**) == FALSE.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`0b11` Supported and enabled.
(**HIDEN** | **HNIDEN**) & (**DBGEN** | **NIDEN**) == TRUE.

All other values are reserved.

HID, bits[9:8]

Hypervisor invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor invasive debug are not implemented, or no hypervisor is implemented. For ARMv7 processors that implement the Virtualization Extensions, and for ARMv8 processors that implement EL2, if separate controls for hypervisor debug visibility are not implemented, the hypervisor debug visibility is indicated by the relevant Non-secure debug visibility fields NSNID and NSID. See the relevant ARM Architecture Manual for more information about Virtualization Extensions and EL2.

`0b10` Supported and disabled. (**HIDEN** & **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**HIDEN** & **DBGEN**) == TRUE.

All other values are reserved.

SNID, bits[7:6]

Secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled.
(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == FALSE.

`0b11` Supported and enabled.
(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == TRUE.

All other values are reserved.

SID, bits[5:4]

Secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**SPIDEN** & **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**SPIDEN** & **DBGEN**) == TRUE.

All other values are reserved.

NSNID, bits[3:2]

Non-secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**NIDEN** | **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**NIDEN** | **DBGEN**) == TRUE.

All other values are reserved.

NSID, bits[1:0]

Non-secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. **DBGEN** == FALSE.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`0b11` Supported and enabled. **DBGEN** == TRUE.

All other values are reserved.

Accessing AUTHSTATUS

AUTHSTATUS can be accessed at the following address:

Offset
`0xFB8`

B2.3.2 CLAIMSET and CLAIMCLR, Claim Tag Set Register and Claim Tag Clear Register

The characteristics of CLAIMSET and CLAIMCLR are:

Purpose

Often there are several debug agents that must cooperate to control the resources that the CoreSight components make available. For example, an external debugger and a debug monitor running on the target might both require control of the breakpoint resources of a processor. It is important that a debug agent does not reprogram debug resources that another debug agent is using.

The Claim tag registers provide various bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent. All debug agents must implement a common protocol to use these bits.

This specification does not define the claim tag protocol, but consider the following examples that illustrate how these bits can be used:

Protocol 1: Set common bit to claim

In this scenario, debug functionality is only claimed on a few rare, well-defined points, for example when the target is powered up or when a debugger is connected.

Each bit in the claim tag corresponds to an area of debug functionality, which is shared between all debug agents. For example, 4 bits can control four areas of functionality. The following shows a pseudocode implementation of this protocol:

```
read claim tag bit
if (bit is set)
    functionality is not available
else
    set bit
    use functionality
```

Protocol 2: Set private bit to claim

In this scenario, debug functionality is also only claimed on a few rare, well-defined points, but it is necessary to be able to determine which other agent has claimed functionality.

Each bit in the claim tag corresponds to an area of debug functionality for a debug agent. For example, 4 bits can control two areas of functionality each for two debug agents. The following shows a pseudocode implementation of this protocol:

```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
else
    set bit for this agent
    use functionality
```


Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Protocol 3: Set private bit and check for race

In this scenario, debug functionality is claimed regularly and it is possible for two debug agents to attempt to claim it at the same time. Each bit in the claim tag corresponds to an area of debug functionality for a debug agent, as in protocol 2. The following shows a pseudocode implementation of this protocol:

```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
else
    set bit for this agent
    read all claim tag bits for this functionality
    if (any bits are set by other agents)
        clear bit for this agent
        wait a random amount of time
        go back to start
    else
        use functionality
```

Usage constraints

The value of CLAIMCLR must be zero at reset.

CLAIMSET and CLAIMCLR are accessible as follows:

Default
RW

Configurations

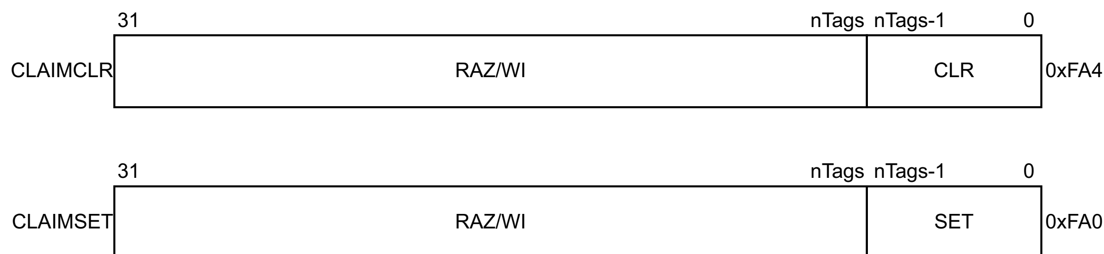
Included in all implementations.

Attributes

CLAIMSET and CLAIMCLR are 32-bit registers.

Field Descriptions

The CLAIMSET and CLAIMCLR bit assignments are:



CLAIMCLR bits[31:nTags]

RAZ/WI

CLR, CLAIMCLR bits[nTags-1:0]

The size of this field, nTags, is IMPLEMENTATION DEFINED, and equals the number of bits set in CLAIMSET.

Allowed values of CLR[*n*] are:

Write `0` No effect.

Write `1` Clear the claim tag for bit[*n*].

Read `0` The claim tag bit is not set.

Read `1` The claim tag bit is set.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIMSET bits[31:nTags]

RAZ/WI

SET, CLAIMSET bits[nTags-1:0]

The size of this field, nTags, is IMPLEMENTATION DEFINED, and equals the number of claim bits that are implemented.

Permitted values of SET[*n*] are:

Write `0` No effect.

Write `1` Set the claim tag for bit[*n*].

Read `0` The claim tag that is represented by bit[*n*] is not implemented.

Read `1` The claim tag that is represented by bit[*n*] is implemented.

Accessing CLAIMCLR and CLAIMSET

CLAIMCLR and CLAIMSET can be accessed at the following address:

Offset	
CLAIMCLR	**CLAIMSET**
`0xFA4`	`0xFA0`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.3 DEVAFF0-DEVAFF1, Device Affinity Registers

The DEVAFF characteristics are:

Purpose

Enables a debugger to determine whether two components have an affinity with each other.

For example, when a trace macrocell connects to a processor, the DEVAFF0 and DEVAFF1 in both components must contain identical values that are unique. Doing so enables the debugger to identify how the components relate to each other, without performing topology detection.

Usage constraints

DEVAFF0-DEVAFF1 are accessible as follows:

Default
RO

Configurations

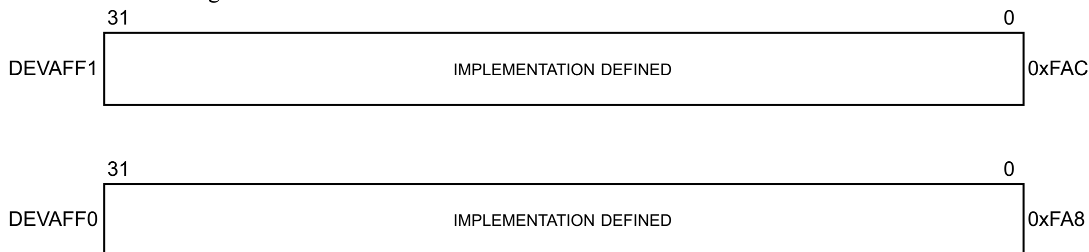
Included in all implementations.

Attributes

DEVAFF0-DEVAFF1 is a set of 32-bit registers that return an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVAFF bit assignments are:



DEVAFF1, bits[31:0]
DEVAFF0, bits[31:0]

IMPLEMENTATION DEFINED. If a component has no unique association with another component, these fields are RAZ.

Examples of the content that DEVAFF returns are the MPIDRs of ARM architecture processors and CTIs that connect to them:

- DEVAFF0 returns MPIDR, bits[31:0].
- DEVAFF1 returns MPIDR, bits[63:32].

Accessing DEVAFF0-DEVAFF1

DEVAFF0-DEVAFF1 can be accessed at the following address:

Offset	
DEVAFF0	**DEVAFF1**
`0xFA8`	`0xFAC`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.4 DEVARCH, Device Architecture Register

The DEVARCH characteristics are:

Purpose

Identifies the architect and architecture of a CoreSight component. The architect might differ from the designer of a component, for example when ARM defines the architecture but another company designs and implements the component.

Usage constraints

DEVARCH is accessible as follows:

Default
RO

Configurations

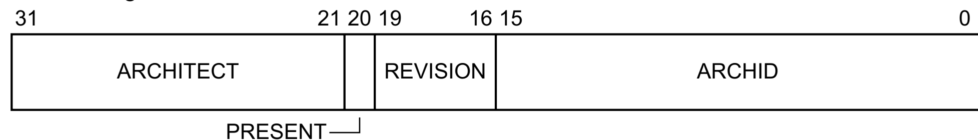
Included in all implementations.

Attributes

DEVARCH is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVARCH bit assignments are:



ARCHITECT, bits[31:21]

Defines the architect of the component:

Bits[31:28] Indicates the JEP106 continuation code.

Bits[27:21] Indicates the JEP106 identification code.

See the *Standard Manufacturers Identification Code* for information about JEP106. For components where ARM is the architect, this 11-bit field returns `0x23B`.

PRESENT, bit[20]

Indicates the presence of this register:

0 = DEVARCH is not present. Bits[31:0] must be RAZ.

1 = DEVARCH is present.

REVISION, bits[19:16]

Architecture revision. Returns the revision of the architecture that the ARCHID field specifies.

ARCHID, bits[15:0]

Architecture ID. Returns a value that identifies the architecture of the component.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-8 lists the ARCHID values for some example components where ARM is the architect.

Table B2-8 Example ARCHID values

ARCHID	Description
`0x0A00`	RAS architecture
`0x1A01`	*Instrumentation Trace Macrocell* (ITM) architecture
`0x1A02`	DWT architecture
`0x1A03`	*Flash Patch and Breakpoint unit* (FPB) architecture
`0x2A04`	Processor debug architecture (ARMv8-M)
`0x6A05`	Processor debug architecture (ARMv8-R)
`0x0A10`	PC sample-based profiling
`0x4A13`	*Embedded Trace Macrocell* (ETM) architecture.
`0x1A14`	*Cross Trigger Interface* (CTI) architecture
`0x6A15`	Processor debug architecture (v8.0-A)
`0x7A15`	Processor debug architecture (v8.1-A)
`0x8A15`	Processor debug architecture (v8.2-A)
`0x2A16`	Processor *Performance Monitor* (PMU) architecture
`0x0A17`	Memory Access Port v2 architecture
`0x0A27`	JTAG Access Port v2 architecture
`0x0A31`	Basic trace router
`0x0A37`	Power requestor
`0x0A47`	Unknown Access Port v2 architecture
`0x0A50`	HSSTP architecture
`0x0A63`	*System Trace Macrocell* (STM) architecture
`0x0A75`	CoreSight ELA architecture
`0x0AF7`	CoreSight ROM architecture

Accessing DEVARCH

DEVARCH can be accessed at the following address:

Offset
`0xFBC`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.5 DEVID, Device Configuration Register

The DEVID characteristics are:

Purpose

Indicates the capabilities of the component.

Usage constraints

This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be RAZ.

If the component is configurable, ARM recommends that this register reflects any changes to a standard configuration.

DEVID is accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

DEVID is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID bit assignments are:

Bits[31:0]

IMPLEMENTATION DEFINED.

Accessing DEVID

DEVID can be accessed at the following address:

Offset
`0xFC8`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.6 DEVID1, Device Configuration Register 1

The DEVID1 characteristics are:

Purpose

Indicates the capabilities of the component.

Usage constraints

This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be RAZ.

If the component is configurable, ARM recommends that this register reflects any changes to a standard configuration.

DEVID1 is accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

DEVID1 is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID1 bit assignments are:

Bits[31:0]

IMPLEMENTATION DEFINED.

Accessing DEVID1

DEVID1 can be accessed at the following address:

Offset
`0xFC4`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.7 DEVID2, Device Configuration Register 2

The DEVID2 characteristics are:

Purpose

Indicates the capabilities of the component.

Usage constraints

This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be RAZ.

If the component is configurable, ARM recommends that this register reflects any changes to a standard configuration.

DEVID2 is accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

DEVID2 is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID2 bit assignments are:

Bits[31:0]

IMPLEMENTATION DEFINED.

Accessing DEVID2

DEVID2 can be accessed at the following address:

Offset
`0xFC0`

B2.3.8 DEVTYPE, Device Type Identifier Register

The DEVTYPE characteristics are:

Purpose

If the part number field is not recognized, a debugger can report the information that is provided by DEVTYPE about the component instead.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints

DEVTYPE is accessible as follows:

Default
RO

Configurations

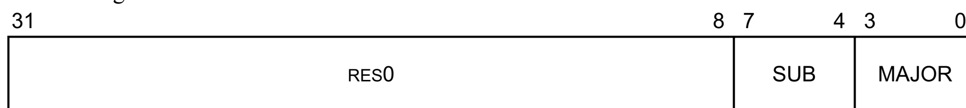
Included in all implementations.

Attributes

DEVTYPE is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVTYPE bit assignments are:



Bits[31:8]

RES0.

SUB, bits[7:4]

Sub type for the component device type, as described in Table B2-9.

MAJOR, bits[3:0]

Major type for the component device type, as described in Table B2-9.

Table B2-9 Device type encoding

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
`0x0`	Miscellaneous	`0x0`	Other, undefined.
		`0x1-0x3`	Reserved.
		`0x4`	Validation component.
		`0x5-0xF`	Reserved.
`0x1`	Trace Sink	`0x0`	Other.
		`0x1`	Trace port, for example TPIU.
		`0x2`	Buffer, for example ETB.
		`0x3`	Basic trace router.
		`0x4-0xF`	Reserved.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-9 Device type encoding (continued)

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
`0x2`	Trace Link	`0x0`	Other.
		`0x1`	Trace funnel, Router.
		`0x2`	Filter.
		`0x3`	FIFO, Large Buffer.
		`0x4-0xF`	Reserved.
`0x3`	Trace Source	`0x0`	Other.
		`0x1`	Associated with a processor core.
		`0x2`	Associated with a DSP.
		`0x3`	Associated with a Data Engine or coprocessor.
		`0x4`	Associated with a Bus, stimulus-derived from bus activity.
		`0x5`	Reserved.
		`0x6`	Associated with software, stimulus-derived from software activity.
		`0x7-0xF`	Reserved.
`0x4`	Debug Control	`0x0`	Other.
		`0x1`	Trigger Matrix, for example ECT.
		`0x2`	Debug Authentication Module. See *Control of authentication interfaces* on page D2-115
		`0x3`	Power requestor.
		`0x4-0xF`	Reserved.
`0x5`	Debug Logic	`0x0`	Other.
		`0x1`	Processor core.
		`0x2`	DSP.
		`0x3`	Data Engine or coprocessor.
		`0x4`	Bus, stimulus-derived from bus activity.
		`0x5`	Memory, tightly coupled device such as *Built In Self-Test* (BIST).
		`0x6-0xF`	Reserved.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-9 Device type encoding (continued)

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
`0x6`	Performance Monitor	`0x0`	Other.
		`0x1`	Associated with a processor.
		`0x2`	Associated with a DSP.
		`0x3`	Associated with a Data Engine or coprocessor.
		`0x4`	Associated with a bus, stimulus-derived from bus activity.
		`0x5`	Associated with a Memory Management Unit that conforms to the ARM System MMU Architecture.
		`0x6-0xF`	Reserved.
`0x7-0xF`	Reserved	-	-

Accessing DEVTYPE

DEVTYPE can be accessed at the following address:

Offset
`0xFCC`

B2.3.9 ITCTRL, Integration Mode Control Register

The ITCTRL characteristics are:

Purpose

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

In integration mode, topology detection is enabled. For more information, see Chapter B3 *Topology Detection*.

Usage constraints

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.

ITCTRL is accessible as follows:

Default
RW

Configurations

This register is not required. If no integration functionality is implemented, this register must be RAZ.

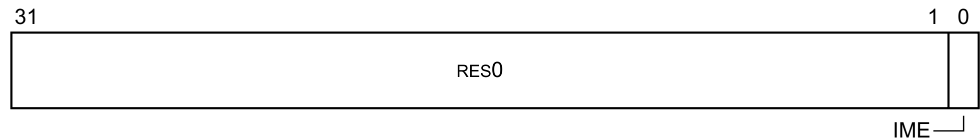
Attributes

ITCTRL is a 32-bit register.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The ITCTRL bit assignments are:



Bits[31:1]

RES0.

IME, bits[0] Permitted values of IME are:

`0` The component must enter functional mode.

`1` The component must enter integration mode, and enable support for topology detection and integration testing.

Accessing ITCTRL

ITCTRL can be accessed at the following address:

Offset
`0xF00`

B2.3.10 LSR and LAR, Software Lock Status Register and Software Lock Access Register

The characteristics of the Software lock registers are:

Purpose

The Software lock mechanism prevents accidental access to the registers of CoreSight components. Software that is being debugged might accidentally write to memory used by CoreSight components. Accidental accesses might disable those components, making the software impossible to debug. The CoreSight programmers' model includes a Lock Status Register LSR, and a Lock Access Register, LAR, to control software access to CoreSight components to ensure that the likelihood of accidental access to CoreSight components is small.

Note

From CoreSight version 3.0 onwards, implementation of the Software lock mechanism that is controlled by LAR and LSR is deprecated.

To ensure that the software being debugged can never access an unlocked CoreSight component, a software monitor that accesses debug registers must unlock the component before accessing any registers, and lock the component again before exiting the monitor.

ARM recommends that external accesses from a debugger are not subject to the Software lock, and therefore that external reads of the LSR return zero. For information on how CoreSight components can distinguish between external and internal accesses, see *Debug APB interface memory map* on page D2-116.

A system can include several bus masters capable of accessing the same CoreSight component, for example in systems that include several processors. In this case, it is possible for software running on one processor, processor A, to accidentally access the component while it is being programmed by a debug monitor running on another processor, processor B. Because the component that is being accessed cannot distinguish between the two processors, processor A might disable the component and cause problems for processor B. The probability of this occurring is low, but must be considered if there are special circumstances that make this scenario more likely.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The claim tag cannot be used to manage accesses to the Software lock registers, because access to the claim tag is subject to the Software lock mechanism.

Usage constraints

LSR and LAR are accessible as follows:

Default	
LSR	**LAR**
RO	WO

Configurations

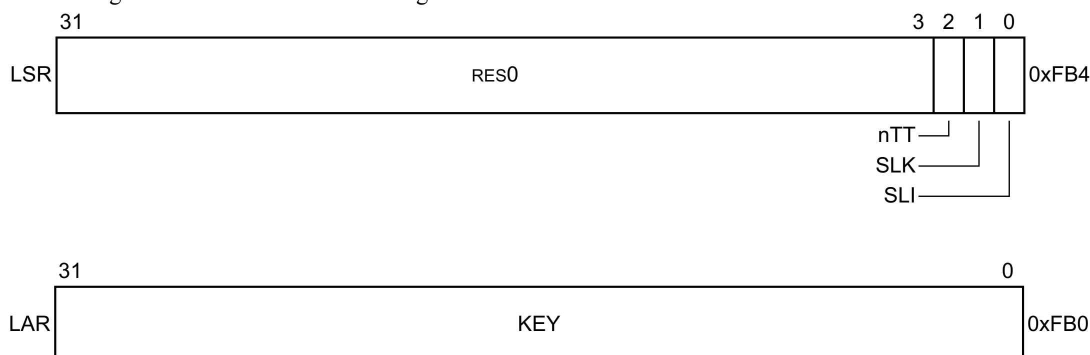
LSR is included in all implementations, and LSR.SLI indicates whether LAR is implemented.

Attributes

The Software lock registers are 32-bit registers.

Field Descriptions

The bit assignments of the Software lock registers are:



LSR, bits[31:3]

RES0.

nTT, LSR bits[2]

This bit is always zero, which indicates that the component implements a 32-bit LAR.

SLK, LSR bits[1]

This field is used to return the current software lock status.

Permitted values of SLK are:

`0` Writing to the other registers in the component is permitted.

`1` Writing to the other registers in the component is blocked.

Note

When present, the reset value of this bit is 1.

SLI, LSR bits[0]

This field indicates whether a Software lock mechanism is implemented.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Permitted values of SLI are:

`0` Software lock mechanism is not implemented.

`1` Software lock mechanism is implemented.

Note

Some components have two programmable views, one only visible from external tools and the other visible from software running on-chip. In this case:

- For accesses from external tools, the Software lock mechanism is not required and LSR.SLI and LSR.SLK both return a value of zero.
- For accesses from software running on-chip the Software lock is optional, and, when implemented, LSR.SLI has the value `0b1` and LSR.SLK returns the status of the Software lock.

KEY, LAR bits[31:0]

Writing a value to this field controls write access to the other registers in the component.

Permitted values of KEY are:

Write `0xC5ACCE55`

Signals that LSR must permit writing to the other registers in the component.

Write any other value

Signals that LSR must block writing to the other registers in the component.

Accessing LSR and LAR

LSR and LAR can be accessed at the following address:

Offset	
LAR	**LSR**
`0xFB0`	`0xFB4`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B2.4 Component-specific registers for Class 0xF CoreLink, PrimeCell, and system components

Components that have the value `0xF` assigned to the CIDR1.CLASS field in the Component Identification Register are CoreLink, PrimeCell, or system components. For details, see *CIDR0-CIDR3, Component Identification Registers* on page B2-38.

CoreLink, PrimeCell, and system components are not related to the CoreSight system.

No component-specific registers are specified for this component class.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B3
Topology Detection

This chapter describes the CoreSight topology detection registers. It contains the following sections:

- *About topology detection* on page B3-64.
- *Requirements for topology detection signals* on page B3-65.
- *Combination with integration registers* on page B3-66.
- *Interfaces that are not connected or implemented* on page B3-67.
- *Variant interfaces* on page B3-68.
- *Documentation requirements for topology detection registers* on page B3-69.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.1 About topology detection

CoreSight system components can have various interface types. A component specifies which interfaces are present, and whether they act as master or slave. Each interface type defines a set of control signals that enable a debugger to determine which other components are connected to it. These signals are referred to as topology detection signals. During topology detection, a debugger probes each interface to determine which other components are connected to it.

For the specification of the requirements for the topology detection signals for standard interfaces that are used by ARM CoreSight components, see Chapter C7 *Topology Detection at the Component Level*. Interface vendors must define the requirements for other interfaces, following the rules in Chapter D8 *Topology Detection at the System Level*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.2 Requirements for topology detection signals

Topology detection signals must observe the following requirements:

- For each topology detection input, it must be possible to read the state of that input.
- For each topology detection output, it must be possible to drive the state of that output without affecting other topology detection signals.

Note

- It is not necessary to implement topology detection registers on the programming interface for the component, because this connectivity is described by the ROM Table.
- Topology detection can be invasive. See Chapter D8 *Topology Detection at the System Level*.

B3.2.1 Recommended method

ARM recommends that topology detection registers are implemented as follows:

- Implement a topology detection mode that isolates the topology detection signals.
- For each topology detection output, provide a register that sets the value of that output in topology detection mode.
- For each topology detection input, provide a register that returns the value of that input in topology detection mode.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.3 Combination with integration registers

In addition to the registers that are required for topology detection, many components implement integration registers that provide the same control over most inputs and outputs. This technique enables rapid integration testing when validating a SoC built from these components, because a test bench can assess the connectivity between two components without knowledge of their underlying functionality.

For components that implement integration registers, ARM recommends reusing these registers for topology detection. Use the ITCTRL register to select both integration mode and topology detection mode. See also *ITCTRL, Integration Mode Control Register* on page B2-58.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.4 Interfaces that are not connected or implemented

Some components do not implement a fixed number of interfaces to allow for the possibility of interfaces not being connected. To the debugger, there is no difference between an interface that is not present and one that is not connected.

If the component requires that the interface is still usable when connected to a non-CoreSight component that is not capable of topology detection, the programmers' model must indicate whether the interface is connected or not.

If the component can only be connected to other CoreSight components, the tools can assume that the interface does not exist if they fail to find any connected interfaces during topology detection. In this case, the programmers' model does not need to indicate whether the interface is connected, but if it does, some time can be saved during topology detection.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.5 Variant interfaces

Usually, the connections between interfaces do not change after detection. However, sometimes it is necessary for some components to share a component. For example, a component tracing the operation of a processor might switch to tracing the operation of a different processor. It is important that the conditions under which a switch can occur are understood.

The connections between interfaces can only change if all the following conditions apply:

- An interface is defined as being variant between multiple connections.
- The programmers' model of the affected component controls the configuration by selecting between several alternative connections for that interface.
- The number of valid alternative connections that are indicated in the programmers' model, which is used to reduce the autodetection time, is less than 32, and remains constant during switching.

If these conditions are too stringent for your application, a separate CoreSight component that multiplexes the connections is required. Topology detection can then be performed between this new component and the components it is connected to.

When a switch has occurred, topology detection must be repeated to determine the new connections. Because topology detection can be invasive, ARM recommends performing topology detections for all configurations that are likely to occur in advance.

B3.5.1 External multiplexing

Figure B3-1 shows an example of how variable connections can be implemented using an external multiplexer. This example shows:

- A register that indicates that there are *n* inputs to select from. This register can be read by a debugger to determine which values of the selection register are valid. The register is tied to the value *n* outside the component.
- A selection register that selects the input to use.
- A variant connection receiving the selected input.

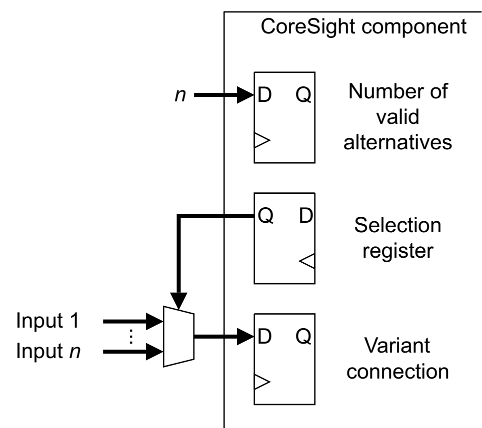


Figure B3-1 External multiplexing of connections

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

B3.6 Documentation requirements for topology detection registers

The component must have documentation that defines the interfaces present on that component. The definition of each interface must include:

- Its name, using the format that is listed in Chapter D8 *Topology Detection at the System Level*.
- If the interface supports variable connections:
  - How many connections are valid.
  - How to switch between connections.
- How to control the topology detection signals listed for that interface in Chapter D8 *Topology Detection at the System Level*.

B3.6.1 Interfaces where topology detection is not possible

If an interface can be connected to a non-CoreSight component, topology detection might not be possible. In this case, the documentation must define a method to determine from the programmers' model how such as component is connected.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part C
CoreSight Reusable Component Architecture

Chapter C1
About the Reusable Component Architecture

The CoreSight reusable component architecture specifies the rules that enable a component to be used with other components that use the CoreSight architecture, and defines the physical interfaces of the components so that they can be connected together easily.

Note

- Unlike the visible component architecture, implementing the reusable component architecture is not mandatory. Omitting the reusable component architecture from a self-contained system does not compromise its compatibility with debuggers, but prevents it from being used with other CoreSight components.
- If a component does not require the functionality that is provided by a particular interface, implementing the interface is optional. For example, a component with no programmable registers does not need to implement the AMBA APB interface.
- It is possible to create a component that performs several functions internally, while presenting only one set of reusable component interfaces. Doing so allows implementing pre-built platforms with an integrated CoreSight infrastructure, enabling the platform to be integrated into a larger system as if it were a single CoreSight component.

The reusable component architecture is described in the following chapters:

- Chapter C2 *AMBA APB and ATB Interfaces*.
- Chapter C3 *Event Interface*.
- Chapter C4 *Channel interface*.
- Chapter C5 *Authentication Interface*.
- Chapter C6 *Timestamp Interface*.
- Chapter C7 *Topology Detection at the Component Level*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C2
AMBA APB and ATB Interfaces

This chapter describes the following AMBA interfaces:

- The *AMBA APB interface* on page C2-76, which is used to program CoreSight components.
- The *AMBA ATB interface* on page C2-78, which transfers trace data.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C2.1 AMBA APB interface

The following sections describe the AMBA APB interface:

- *About the AMBA APB interface*.
- *AMBA APB interface signals*.
- *AMBA APB interface width* on page C2-77.
- *Alternative views of the register file* on page C2-77.

C2.1.1 About the AMBA APB interface

The AMBA APB interface is used to program CoreSight components.

The interface supports:

- Simple, non-pipelined operation.
- Implementation of 8-, 16-, or 32-bit slaves.
- Slave stalling.
- Slave error response.

For more information, see the *ARM® AMBA® APB Protocol Specification*.

Some legacy debug components implement a JTAG TAP Controller to access their functionality.

The bus that connects all CoreSight components is referred to as the Debug APB interface.

C2.1.2 AMBA APB interface signals

Table C2-1 shows the signals that comprise the AMBA APB interface.

Note

- The signal suffix **DBG** indicates that the Debug APB interface is used to access CoreSight components.
- The clamp value is the value that an output must be clamped to when the component is powered down or disabled.

For more information, see the *ARM® AMBA® APB Protocol Specification*.

Table C2-1 Signals on the Debug APB interface

Name	Direction		Clamp value	Description
	Master	Slave		
PCLKDBG	Input	Input	-	The rising edge of **PCLKDBG** synchronizes all transfers on the AMBA 3 APB interface.
PRESETDBGn	Input	Input	-	This signal resets the interface and is active LOW.
PADDRDBG[31:2]	Output	Input	0	This bus indicates the address of the transfer. It is not necessary to implement unused bits.[a]
PSELDBG	Output	Input	0	This signal indicates that the slave device is selected and a data transfer is required. There is a **PSELDBG** signal for each slave.
PENABLEDBG	Output	Input	0	This signal indicates the second and subsequent cycles of an AMBA APB interface transfer.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table C2-1 Signals on the Debug APB interface (continued)

Name	Direction		Clamp value	Description
	Master	Slave		
PWRITEDBG	Output	Input	0	When HIGH, PWRITEDBG indicates a write access. When LOW, it indicates a read access.
PWDATADBG[31:0]	Output	Input	0	**PWDATADBG[31:0]** is the write data bus. When **PWRITEDBG** is HIGH, it indicates that the write data bus is driven by the master during write cycles. The write data bus can be up to 32-bits wide.
PREADYDBG	Input	Output	1	This signal is used by the slave to extend an AMBA APB interface transfer.
PRDATADBG[31:0]	Input	Output	0	**PRDATADBG[31:0]** is the read data bus. When **PWRITEDBG** is LOW, it indicates that the read data bus is driven by the selected slave during read cycles. The read data bus can be up to 32-bits wide.
PSLVERRDBG	Input	Output	1	This signal is returned in the second cycle of the transfer, and indicates an error response. Only use this signal for indicating that a component is not available, for example because it is powered down.

a. The use of PADDRDBG[31] to split the memory map and indicate the difference between external and internal accesses is deprecated. For information on how components that require it can differentiate between external and internal access, see *Debug APB interface memory map* on page D2-116.

C2.1.3 AMBA APB interface width

The AMBA APB interface is 32-bits wide.

Chapter B2 *CoreSight programmers' model* describes the model compatible with a 32-bit AMBA APB interface.

C2.1.4 Alternative views of the register file

There might be several ways to access the registers of a component. It can be useful, for example, to provide special instructions to make debug registers in a processor visible as registers in a linked coprocessor. Provided the debug functionality of the component is also accessible through the AMBA APB interface, alternative methods to access registers are permitted.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C2.2 AMBA ATB interface

The AMBA ATB interface carries trace data around a SoC.

Every CoreSight component or platform with trace capabilities has an AMBA ATB interface, and is either a master or a slave on the AMBA ATB:

- A component or system that generates trace data is a master.
- A component or system that receives trace data is a slave.

The AMBA ATB interface supports the following features:

- Stalling of data, using valid and ready responses.
- Byte-sized packets, together with control signals to indicate the number of bytes that are valid in a cycle.
- Originating component marker, giving each data packet an associated ID.
- Any trace protocol or data agnostic requirements for the format of the data.
- Check-pointing of data from all originating components.

For more information, see the *ARM® AMBA® ATB Protocol Specification*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C3
Event Interface

A CoreSight system uses the event interface to transfer events between components. It is most commonly used for communicating cross-trigger events between debug components and a *Cross Trigger Interface* (CTI).

The event interface signals are:

EVENTCLK Clock. This signal is typically mapped onto an existing clock signal.

EVENTRESETn Reset. This signal is typically mapped onto an existing reset signal.

EVENT This signal indicates the event, and is typically mapped onto a signal of a different name that describes its purpose.

An event is indicated by a rising edge on **EVENT**. Therefore an event can be signaled at most once every two **EVENTCLK** cycles.

If the event has a duration, the falling edge on **EVENT** indicates its completion.

The interface defines no back-pressure mechanism, so events that are close together might merge. For example, if the event interface crosses an asynchronous boundary to a slower clock domain, events in close succession might merge into a single event.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C4
Channel interface

This chapter describes the channel interface. It contains the following sections:

- *About the channel interface* on page C4-82.
- *Channels* on page C4-84.
- *Channel interface signals* on page C4-85.
- *Channel connections* on page C4-86.
- *Synchronous and asynchronous conversions* on page C4-87.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1 About the channel interface

The channel interface is a special type of event interface that enables CoreSight components to communicate events, as described in Chapter C3 *Event Interface*. The channel interface supports:

- A variable number of event channels.
- Bidirectional communication.
- Synchronous or asynchronous communication.

Some examples of useful events are:

- If two or more processors are required to stop at the same time, they must signal to each other when they have stopped.
- To perform advanced profiling functions, profiling events from many different sources in the system must be shared.

Figure C4-1 shows a channel interface that connects multiple Cross Trigger Interfaces (CTIs), which are provided by CoreSight technology. The channel interface can be supported directly by a CoreSight component, if necessary.

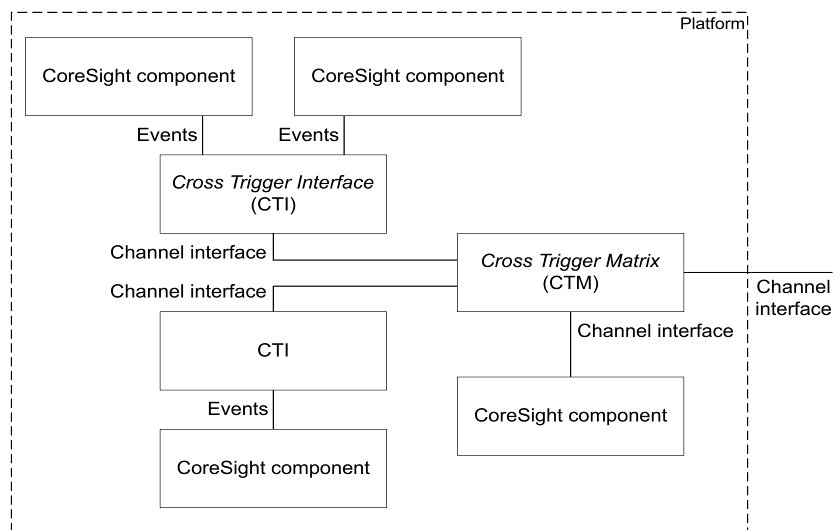


Figure C4-1 Implementation of a CTI-based channel interface

Note

When using the CTI:

- Some systems require more event signals than are supported by a CTI.
- In a platform-oriented system, it is necessary to connect event signals together within the platform, and export only a set of standard interfaces for extension at higher levels.

The channel interface is designed to enable components to communicate events with minimal overhead. However, if multiple events are presented to the channel interface in close succession, they might be interpreted as a single event. Figure C4-2 on page C4-83 illustrates this limitation for a situation where events that are generated in a fast clock domain, Clock A, are passed to a slower clock domain, Clock B. In Clock A, two separate events can be seen, but these events are too close together for Clock B, resulting in Clock B interpreting them as a single event.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

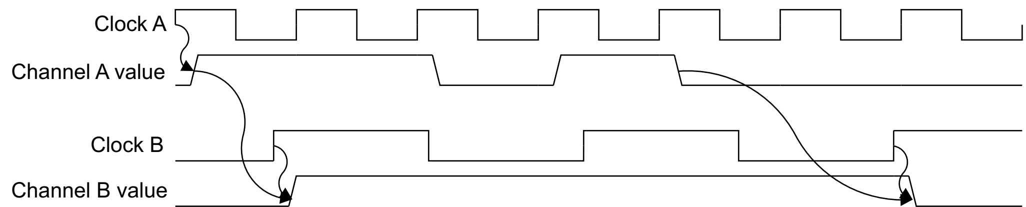


Figure C4-2 Event merging in the channel interface

This limitation makes the channel interface unsuitable for counting events that occur in rapid succession. An example of this type of operation is counting the number of instructions that are executed by a processor over time.

The channel interface is suitable for the following:

- Transmission of an event that happens only once, for example a trigger signal to an ETB or TPIU to end trace capture.
- Transmission of a low-speed signal level where precision is not important.
- Transmission of a signal subject to handshaking using another channel in the channel interface.
- Transmission of a signal subject to software handshaking, for example an interrupt request.
- Transmission of events to be counted that do not occur close together, for example the number of times a peripheral causes an interrupt.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C4.2 Channels

The channel interface comprises two types of signals:

- Channel outputs, which transmit events that are generated by a component.
- Channel inputs, which listen for events that are generated by other components.

A component uses its channel outputs to transmit events to the channel inputs of all components in the system, except its own.

The interface supports an IMPLEMENTATION DEFINED number of channels. ARM recommends that at least four channels are implemented.

Components must treat all channels identically. It must be possible for the debugger to control which channels are used for which purposes.

If a system consists of subsystems with different numbers of channels, and there is a requirement to pass events between these subsystems, the following rules must be observed:

- A subset of the channels from the subsystem with the greater number of channels is connected to all the channels in the other subsystem.
- The set of channels that is connected is always a contiguous set, starting from channel 0.

For example, in a system where subsystem A has eight channels and subsystem B has four channels, channels 0-3 from subsystem A are connected to channels 0-3 in subsystem B. Channels 4-7 are not connected to subsystem B.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C4.3 Channel interface signals

Table C4-1 shows the set of signals that are required by an asynchronous channel interface. The clamp value is the value that an output must be clamped to when the component is powered down or disabled.

Table C4-1 Asynchronous channel interface signals

Name	Direction	Clamp value	Description
CHIN[n-1:0]	Input	-	Channel input
CHINACK[n-1:0]	Output	1	Channel input acknowledge
CHOUT[n-1:0]	Output	0	Channel output
CHOUTACK[n-1:0]	Input	-	Channel output acknowledge

Figure C4-3 shows how the asynchronous interface uses a basic 4-phase handshaking protocol. The same protocol is used by **CHOUT** and **CHOUTACK**.

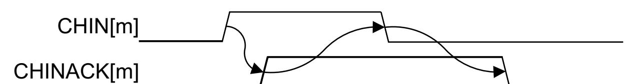


Figure C4-3 Channel interface handshaking

Table C4-2 shows the set of signals that are required by a synchronous channel interface.

Table C4-2 Synchronous channel interface signals

Name	Direction	Clamp value	Description
CHCLK	Input	-	Clock
CHIN[n-1:0]	Input	-	Channel input
CHOUT[n-1:0]	Output	0	Channel output

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C4.4 Channel connections

The channel interface is bidirectional. Take care to connect the correct signals together. Figure C4-4 shows how to connect two asynchronous channel interfaces together.

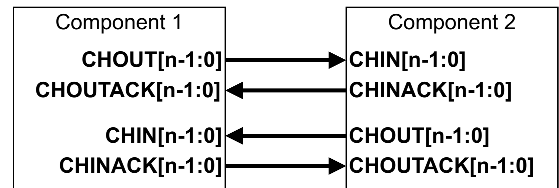


Figure C4-4 Asynchronous channel interface connection

Figure C4-5 shows how to connect two synchronous channel interfaces together.

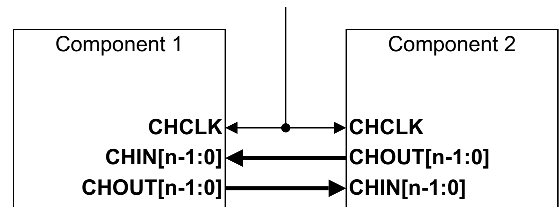


Figure C4-5 Synchronous channel interface connection

A component can support:

- Only the input channels, which is appropriate for components that do not generate events, but have to react to events from other components.
- Only the output channels, which is appropriate for components that generate events, but do not have to react to events from other components.
- Both the input and output channels.

If a component does not support both sets of channels, the unsupported outputs must be clamped as shown in Table C4-1 on page C4-85 and Table C4-2 on page C4-85.

If a component supports both input and output channels, the component must not reflect events on an input channel to the corresponding output channel.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C4.5 Synchronous and asynchronous conversions

Figure C4-6 shows a circuit that makes it possible to convert between synchronous and asynchronous versions of this interface.

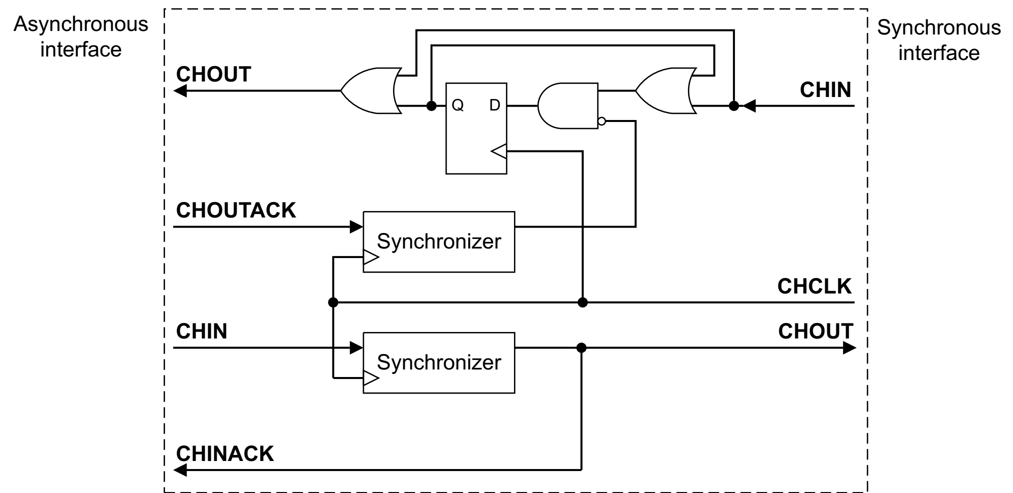


Figure C4-6 Asynchronous to synchronous converter

Implementing a synchronous to asynchronous converter increases the likelihood of events being merged as described in *About the channel interface* on page C4-82 on page C4-82.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5
Authentication Interface

This chapter defines the system requirements that control access to debug and trace peripherals, and how those requirements are met by devices that comply with the CoreSight architecture. It contains the following sections:

- *About the authentication interface* on page C5-90.
- *Definitions of Secure, hypervisor, and invasive debug* on page C5-91.
- *Authentication interface signals* on page C5-92.
- *Authentication rules* on page C5-93.
- *User mode debugging* on page C5-97.
- *Control of the authentication interface* on page C5-98.
- *Exemptions from implementing the authentication interface* on page C5-99.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1 About the authentication interface

Use the authentication interface for restricting access to debug and trace functionality in the following ways:

- To prevent unauthorized people from modifying the behavior of the system, for example to prevent a mobile phone from reporting a fake identification number to the network, which requires authenticated access to invasive debug functions such as traditional core debug, but permits non-invasive tracing and profiling functions.
- To prevent unauthorized people from reverse engineering a product or discovering secrets that are stored within it, for example to read encryption keys, which requires authenticated access to all debug and trace functions.

The authentication interface does not prevent accidental access of debug functionality by rogue code, making a system impossible to debug. This type of access is managed by the Software lock mechanism that is optional in all CoreSight components, see *LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2 Definitions of Secure, hypervisor, and invasive debug

This section defines Secure debug, invasive debug, and hypervisor debug.

C5.2.1 Definition of Secure debug

A Non-secure debug operation is any operation where instructions executing on-chip with Non-secure privileges have the same effect as external operations. Any other operation is a Secure debug operation.

Using this definition, debug operations that monitor the time that is taken by a Secure routine are Non-secure debug operations, because the time taken can be measured by combining off-chip timing information with Non-secure on-chip event generation information. Operations that affect the time that is taken by a Secure routine are considered Secure debug operations.

C5.2.2 Definition of hypervisor debug

The meaning of hypervisor debug is IMPLEMENTATION DEFINED. For processors based on ARM architectures, see the relevant ARM Architecture Reference Manual.

C5.2.3 Definition of invasive debug

Any operation that changes the defined behavior of the system is invasive.

Examples include any changes to the contents of memory and insertion of instructions into a processor pipeline, but not necessarily the act of changing the number of cycles that are taken to perform an operation, unless the number of cycles is defined architecturally.

An implementation can treat an IMPLEMENTATION DEFINED set of effects that change the observable, but not the defined behavior of the system, as invasive. Examples include most effects that change the number of cycles that are taken to perform an operation.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3 Authentication interface signals

Table C5-1 shows the authentication interface signals that a component might support. If a component uses a non-invasive enable signal, it must import the invasive equivalent. For example, using **SPNIDEN** requires importing **SPIDEN**, and using **HNIDEN** requires importing **HIDEN**.

Table C5-1 Authentication interface signals

AUTHCLK	Input	Clock (Not used for asynchronous authentication)[a]
AUTHRESETn	Input	Reset (Not used for asynchronous authentication)[a]
DBGEN	Input	Invasive debug enable
NIDEN	Input	Non-invasive debug enable
SPNIDEN	Input	Secure non-invasive debug enable
SPIDEN	Input	Secure invasive debug enable
HIDEN	Input	Hypervisor invasive debug enable
HNIDEN	Input	Hypervisor non-invasive debug enable

a. Use of the asynchronous authentication interface is deprecated.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4 Authentication rules

Authentication interface implementations must observe the following rules:

1. Synchronous interfaces must sample all signals synchronously, on the rising edge of **AUTHCLK**. Typically, **AUTHCLK** is mapped onto another clock signal. It is IMPLEMENTATION DEFINED when a change of any of the authentication signals takes effect. For example, a processor core might ignore changes to the authentication signals while in Debug state. By extension, it is possible that a component only observes the signals on reset, but it is recommended that more frequent changes are permitted.

   Asynchronous interfaces must sample all signals asynchronously.

   ARM recommends that processors implementing the authentication interface specify a sequence of instructions that, when executed, wait until changes to the authentication signals have taken effect before continuing.

2. If **DBGEN** is LOW, invasive debug is not permitted.

   Invasive debug is any debug operation that might cause the behavior of the system to be modified. Non-invasive debug, such as trace, is unaffected.

3. If **NIDEN** is LOW and **DBGEN** is LOW, neither invasive nor non-invasive debug is permitted.

4. If **NIDEN** is LOW and **DBGEN** is HIGH, both invasive and non-invasive debug are permitted. ARM recommends that these signals are not driven in this way.

   To ensure that a non-invasive component is correctly enabled, it must import both **DBGEN** and **NIDEN**, and internally OR the result.

5. If **SPIDEN** is LOW, Secure invasive debug is not permitted.

6. If **SPNIDEN** is LOW and **SPIDEN** is LOW, all Secure debug is not permitted.

7. If **SPNIDEN** is LOW and **SPIDEN** is HIGH both invasive and non-invasive Secure debug are permitted. ARM recommends that these signals are not driven in this way. To ensure that a non-invasive component is correctly enabled, it must import **SPIDEN** in addition to **SPNIDEN**, and internally OR the result.

Note

Rules 5 to 7 are the equivalent of rules 2 to 4, but used for Secure debug. They are useful for systems that separate Secure and Non-secure data, for example systems implementing ARM Security Extensions. Secure non-invasive debug is any debug operation that enables a debugger to read Secure data. Secure invasive debug is any debug operation that enables a debugger to change Secure data. If a debug component supports Secure non-invasive debug functions by implementing the signal **SPNIDEN**, it must also observe the Secure invasive signal, **SPIDEN**.

---

8. If **SPIDEN** is HIGH and **DBGEN** is LOW, invasive debug is not permitted. ARM recommends that these signals are not driven in this way. To ensure that a component that supports Secure invasive debug is correctly controlled, ARM recommends importing both **DBGEN** and **SPIDEN**, and using the result of an internal AND operation with the imported signals as operands for authentication.

9. If **SPNIDEN** is HIGH and **NIDEN** is LOW, debugging is not permitted. ARM recommends that these signals are not driven in this way. To ensure that a component that supports Secure non-invasive debug is correctly controlled, it must import **NIDEN** in addition to **SPNIDEN**, and internally AND the result.

10. If **HIDEN** is LOW, hypervisor invasive debug is not permitted.

11. If **HNIDEN** is LOW and **HIDEN** is LOW, all hypervisor debug is not permitted.

12. If **HNIDEN** is LOW and **HIDEN** is HIGH both invasive and non-invasive hypervisor debug are permitted. ARM recommends that these signals are not driven in this way. To ensure that a non-invasive component is correctly enabled, it must import **HIDEN** in addition to **HNIDEN**, and internally OR the result.

13. If a component supports **HNIDEN**, it must also support **HIDEN** and **NIDEN**.

14. If a component supports **HIDEN**, it must also support **DBGEN**.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

15. If **HIDEN** is HIGH and **DBGEN** is LOW, invasive debug is not permitted. ARM does not recommend that these signals are driven in this way. To ensure that a component that supports hypervisor invasive debug is correctly controlled, ARM recommends importing both **DBGEN** and **HIDEN**, and using the result of an internal AND operation with the imported signals as operands for authentication.
16. If **HNIDEN** is HIGH and **NIDEN** is LOW, debugging is not permitted. ARM does not recommend that these signals are driven in this way. To ensure that a component that supports Secure non-invasive debug is correctly controlled, it must import **NIDEN** in addition to **HNIDEN**, and internally AND the result.
17. If the value of any of the authentication signals changes, it is IMPLEMENTATION DEFINED when it takes effect.

    Pipeline effects mean that it is not possible for these signals to be precise. ARM recommends not to use them to enable and disable debugging around specific regions of code without a full understanding of the pipeline behavior of the system.

The authentication rules can be summarized as follows:

- **SPIDEN**, **DBGEN**, **SPNIDEN**, and **NIDEN** enable Secure invasive debug, Non-secure invasive debug, Secure non-invasive debug, and Non-secure non-invasive debug, respectively.
- Because invasive functionality requires non-invasive functionality to function correctly, if invasive debug is enabled, non-invasive debug must also be enabled.
- Secure functionality must be disabled if the corresponding Non-secure functionality is disabled.

Table C5-2 , Table C5-3, and Table C5-4 show the equations that define whether a particular level of debug functionality is permitted for a debug component that supports the authentication interface:

Table C5-2 Component without Secure debug capabilities

Debug functionality	Equation
Invasive debug	**DBGEN**
Non-invasive debug	**DBGEN** | **NIDEN**

Table C5-3 Component with Secure debug capabilities

Debug functionality	Equation
Non-secure invasive debug	**DBGEN**
Non-secure non-invasive debug	**DBGEN** | **NIDEN**
Secure invasive debug	**DBGEN** & **SPIDEN**
Secure non-invasive debug	(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**)

Table C5-4 Component with hypervisor debug capabilities

Debug functionality	Equation
Hypervisor invasive debug	**HIDEN** & **DBGEN**
Hypervisor non-invasive debug	(**HIDEN** | **HNIDEN**) & (**DBGEN** | **NIDEN**)

The **SPIDEN** and **SPNIDEN** signals have no dependencies on the **HIDEN** and **HNIDEN** signals, and conversely.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table C5-5 shows the restrictions for **SPIDEN** and **SPNIDEN** and their effects. Numbers in brackets indicate the rules that apply in each case, S indicates Secure, and NS indicates Non-secure.

Table C5-5 Authentication signal restrictions for SPIDEN and SPNIDEN

SPIDEN	DBGEN	SPNIDEN	NIDEN	Valid signal combination	Invasive debug permitted		Non-invasive debug permitted	
					S	NS	S	NS
0	0	0	0	Yes	No (2,5)	No (2)	No (3,6)	No (3)
0	0	0	1	Yes	No (2,5)	No (2)	No (6)	Yes
0	0	1	0	No[a] (9)	No (2,5)	No (2)	No (3,6)	No (3)
0	0	1	1	Yes	No (2,5)	No (2)	Yes	Yes
0	1	0	0	No (4)	No (5)	Yes (4)	No (6)	Yes (4)
0	1	0	1	Yes	No (5)	Yes	No (6)	Yes
0	1	1	0	No (4)	No (5)	Yes (4)	Yes (4)	Yes (4)
0	1	1	1	Yes	No (5)	Yes	Yes	Yes
1	0	0	0	No (7)	No (2)	No (2)	No (3)	No (3)
1	0	0	1	No (7)	No (2)	No (2)	Yes (7)	Yes
1	0	1	0	No[a] (8,9)	No (2)	No (2)	No (3)	No (3)
1	0	1	1	No[a] (8)	No (2)	No (2)	Yes	Yes
1	1	0	0	No (4,7)	Yes (7)	Yes (4)	Yes (4,7)	Yes (4)
1	1	0	1	No (7)	Yes (7)	Yes	Yes (7)	Yes
1	1	1	0	No (4)	Yes (4)	Yes (4)	Yes (4)	Yes (4)
1	1	1	1	Yes	Yes	Yes	Yes	Yes

a. These signal combinations were permitted in previous versions of the CoreSight architecture but are deprecated from v2.0 onwards.

Table C5-6 shows the restrictions for **HIDEN** and **HNIDEN** and their effects. Numbers in brackets indicate the rules that apply in each case, H indicates hypervisor, and NH indicates non-hypervisor.

Table C5-6 Authentication signal restrictions for HIDEN and HNIDEN

HIDEN	DBGEN	HNIDEN	NIDEN	Valid signal combination	Invasive debug permitted		Non-invasive debug permitted	
					H	NH	H	NH
0	0	0	0	Yes	No (2, 10)	No (2)	No (3)	No (3)
0	0	0	1	Yes	No (2, 10)	No (2)	No (3)	Yes
0	0	1	0	No (16)	No (2, 10)	No (2)	No (3)	No (3)
0	0	1	1	Yes	No (2, 10)	No (2)	Yes	Yes
0	1	0	0	No (4)	No (10)	Yes (4)	No (11)	Yes (4)

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table C5-6 Authentication signal restrictions for HIDEN and HNIDEN (continued)

HIDEN	DBGEN	HNIDEN	NIDEN	Valid signal combination	Invasive debug permitted		Non-invasive debug permitted	
					H	NH	H	NH
0	1	0	1	Yes	No (10)	Yes	No (11)	Yes
0	1	1	0	No (4)	No (10)	Yes (4)	Yes (4)	Yes (4)
0	1	1	1	Yes	No (10)	Yes	Yes	Yes
1	0	0	0	No (7)	No (2)	No (2)	No (3)	No (3)
1	0	0	1	No (7)	No (2)	No (2)	Yes (12)	Yes
1	0	1	0	No (15, 16)	No (2)	No (2)	No (3)	No (3)
1	0	1	1	No (15)	No (2)	No (2)	Yes	Yes
1	1	0	0	No (4, 12)	Yes (12)	Yes (4)	Yes (4, 12)	Yes (4)
1	1	0	1	No (12)	Yes (12)	Yes	Yes (12)	Yes
1	1	1	0	No (4)	Yes (4)	Yes (4)	Yes (4)	Yes (4)
1	1	1	1	Yes	Yes	Yes	Yes	Yes

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5 User mode debugging

Individual components can offer greater control over the permitted level of debugging. For example, some processors implementing ARM Security Extensions can grant permission to debug-specific Secure processes by permitting debugging of Secure User mode without permitting debugging of Secure privileged modes. This level of control is extended to the *Embedded Trace Macrocell* (ETM). For more information, see the *ARM® Embedded Trace Macrocell Architecture Specification*.

Figure C5-1 shows how the signals of the CoreSight authentication interface interact with the two registers that are controlled by the Secure *Operating System* (OS), **SUIDEN** and **SUNIDEN**:

- If **DBGEN** is asserted, **NIDEN** is ignored and assumed asserted.
- If **SPIDEN** is asserted, **SPNIDEN** is ignored and assumed asserted.
- In all other cases, the permissions that are represented by all the boxes bounding each level of debug functionality must be granted before that level of debug functionality is enabled.

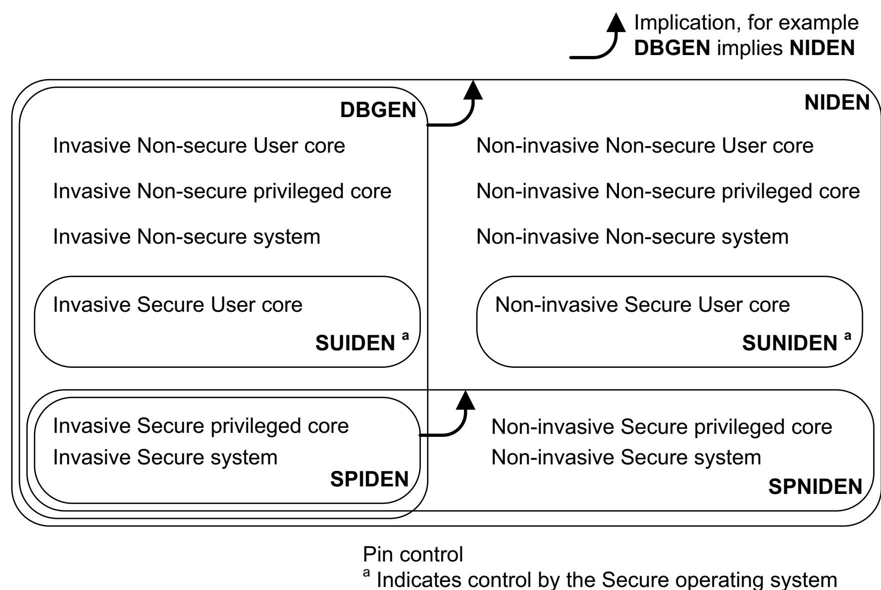


Figure C5-1 Interaction between CoreSight and ARM Security Extensions

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6 Control of the authentication interface

The authentication interface is controlled at the system level. For more information, see *Control of authentication interfaces* on page D2-115.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7 Exemptions from implementing the authentication interface

It is not necessary to implement the authentication interface to control debug functions that are only software-accessible. For these functions, it is sufficient to use standard mechanisms to control software access to privileged and Secure resources.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C6
Timestamp Interface

A CoreSight system uses the wide timestamp interface to distribute a time value to debug components. Typically this time value is included in a trace stream to permit correlation of events in multiple trace streams.

Table C6-1 shows the signals that are defined for the timestamp interface.

Table C6-1 Timestamp interface signals

Name	Description
TSCLK	Interface clock
TSRESETn	Interface reset
TSVALUEB[63:0]	Timestamp value, encoded as a natural binary number. A value of 0 indicates that the timestamp is UNKNOWN, which occurs when the timestamp value source is disabled or when the timestamp value is reset.

If a system with multiple components implements the timestamp interface, the same timestamp is used by all components. If systems use different clocks or different timestamp distribution mechanisms, there might be skew between the timestamp values that are observed by the components.

Some components might not implement a full 64-bit timestamp. These components use an IMPLEMENTATION DEFINED subset of the 64-bit timestamp value. ARM recommends using the subset that provides the largest set of unique observable timestamp values. This subset might depend on the clock speed of the component.

ARM recommends that the timestamp resolution is at least 10% of the fastest processor in the system.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C7
Topology Detection at the Component Level

This chapter describes how to detect components that are connected to the AMBA ATB interface and where they are logically located in any corresponding hierarchical connection. It contains the following sections:

- *About topology detection at the component level* on page C7-104.
- *Interface types for topology detection* on page C7-105.
- *Interface requirements for topology detection* on page C7-107.
- *Signals for topology detection* on page C7-108.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C7.1 About topology detection at the component level

This chapter describes how to perform topology detection on each interface type. Chapter B3 *Topology Detection* describes the topology detection requirements of CoreSight components. Chapter D8 *Topology Detection at the System Level* describes how debuggers can use this information to detect the topology of a target system.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2 Interface types for topology detection

A component has several interfaces that contain one or more signals. Each interface is defined in terms of the following parameters:

- A name, for example channel interface.
- A direction:
  - Master, always connected to one or more slaves of the same type.
  - Slave, always connected to one or more masters of the same type.
  - Bidirectional, always connected to one or more identical bidirectional interfaces.
  - Probe, read-only interface to trace the activity of a bus without affecting the behavior of that bus.

The following conditions apply to the interfaces used for topology detection:

- The list of interfaces must be defined for each block for the purposes of topology detection.
- Not all signals that are used in the implementation must be exposed in an interface, provided the signals that are not exposed are irrelevant for topology detection.
- Signals that are exposed in the interface must be strictly defined.

C7.2.1 Interfaces on standard components

Table C7-1 shows the interfaces present on common CoreSight components. For specific interface details, see the appropriate Technical Reference Manual.

Table C7-1 Interfaces on some example components

Programmable component	Interfaces
CoreSight ETM[a] CoreSight PTM[b]	• AMBA ATB interface, master. • One or more events, master. **EXTOUT[n-1:0]**. • One or more events, slave. **EXTIN[n-1:0]**. • Event, master. **TRIGOUT**. • Variant: CoreETM, slave. The number of core interfaces can be read from the programmers' model of the ETM or PTM. The state of **DBGACK** can be driven directly in all ARM cores, including those cores that are not CoreSight compliant.
CoreSight ETB	• AMBA ATB interface, slave: • Event, master. **ACQCOMP**. • Event, master. **FULL**. • Event, slave. **TRIGIN**. • Event, slave. **FLUSHIN**.
TPIU	• AMBA ATB interface, slave: • Event, slave. **TRIGIN**. • Event, slave. **FLUSHIN**.
Debug Ports and Access Ports	No topology detection interfaces.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table C7-1 Interfaces on some example components (continued)

Programmable component	Interfaces
HTM	• AMBA ATB interface, master: • 2x event, master. **HTMEXTOUT[1:0]**. • 2x event, master. **HTMEXTIN[1:0]**. • Event, master. **HTMTRIGGER**. • Variant: AHB, probe. The number of AHB interfaces can be read from the programmers' model of the HTM. The method to perform topology detection of this interface is not defined.
CoreSight Funnel	• One or more AMBA ATB interfaces, slave. • AMBA ATB interface, master.
CTI	• One or more events, slave. **TRIGIN[7:0]**. • One or more events, master. **TRIGOUT[7:0]**. • Channel, bidirectional.
VIC (PL190/192)	32x event, master: **VICINTSOURCE[n]**.

a. An ETM that implements the ETMv3 or ETMv4 architecture.
b. A PTM that implements the PFTv1 architecture.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C7.3 Interface requirements for topology detection

For all controllable signals, each interface type specifies:

- The signals on the master interface that must be controllable or observable.
- The signals on the slave interface that must be controllable or observable.
- The transitions on the interface that must be performed to trigger the following actions:
  - To initialize topology detection.
  - To assert the master interface.
  - To check whether the slave interface is asserted.
  - To deassert the master interface.
  - To check whether the slave interface is deasserted.

If the interface is bidirectional, each interface to be tested must in turn be treated as a master while the other interfaces of that type are treated as slaves. See Chapter D8 *Topology Detection at the System Level*.

For signals that must be controllable, it must be possible to independently control the value of outputs, and read the value of inputs. See Chapter B3 *Topology Detection*.

Usually each master interface specifies one output, and the slave interface specifies the corresponding input. When choosing a signal, observe the conditions that are described in the following section:

- *Intermediate non-programmable components*.
- *Multi-way connections*.

C7.3.1 Intermediate non-programmable components

Sufficient control signals must be available to enable the interface to be driven to an active state so that it passes through any intermediate non-programmable components. For example, in AMBA ATB interfaces, **ATVALID** must be controllable, because if it is LOW, an intermediate bridge does not pass any control signals through it.

C7.3.2 Multi-way connections

In a multi-way connection:

- Asserting and deasserting a master signal might cause an effect to be seen on multiple slaves.
- Asserting and deasserting a slave signal might cause an effect to be seen on multiple masters.

Sufficient signals must be controllable to cause the arbitration logic to route between the master and slave.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C7.4 Signals for topology detection

Table C7-2 shows the controllable signals for each interface type that is listed in Table C7-1 on page C7-105.

Table C7-2 Controllable signals for each interface type

Interface	Master wires	Slave wires
AMBA ATB interface	**ATVALID**	**ATVALID**, **ATREADY**
CoreETM	**DBGACK**[a]	**DBGACK**[a]
Event[b]	**EVENT**	**EVENT**, **EVENTACK**, if present
Channel, bidirectional	**CHOUT[0]**	**CHIN[0]**, **CHINACK[0]**, if asynchronous

a. Using **DBGACK** for topology detection is restricted by the fact that in some ARM processors it cannot be controlled, or it can only be controlled from a JTAG debugger. To perform topology detection on a processor that has this restriction, use a different IMPLEMENTATION DEFINED controllable signal, or the Device Affinity registers, DEVAFF0-DEVAFF1, which indicate the association of a processor with the ETM.

b. The event interface is defined for miscellaneous point-to-point connections that carry a one-bit signal. An event interface might implement an acknowledge signal, that, if implemented, must be controllable. To implement the event interface, substitute **EVENT** and **EVENTACK** for the equivalent signals in the appropriate interface.

Table C7-3 lists the signals that an interface must implement to support topology detection between masters and slaves of key interface types. Use the table with the algorithm given in *Detection algorithm* on page D8-215. For the full specification of a signal, see the relevant interface specification.

Table C7-3 Topology detection sequences

Signal	AMBA ATB	Core ETM	Event interface	Channel interface
Master preamble	**ATVALID** ← 0	**DBGACK** ← 0	**EVENT** ← 0	**CHOUT[0]** ← 0
Slave preamble	**ATREADY** ← 0	None	**EVENTACK** ← 0, if present	**CHINACK[0]** ← 0, if present
Master assert	**ATVALID** ← 1	**DBGACK** ← 1	**EVENT** ← 1	**CHOUT[0]** ← 1
Slave check asserted	**ATVALID** == 1	**DBGACK** == 1	**EVENT** == 1	**CHIN[0]** == 1
Slave post-assert	**ATREADY** ← 1	None	**EVENTACK** ← 1, if present	**CHINACK[0]** ← 1, if present
Master deassert	**ATVALID** ← 0	**DBGACK** ← 0	**EVENT** ← 0	**CHOUT[0]** ← 0
Slave check deasserted	**ATVALID** == 0	**DBGACK** == 0	**EVENT** == 0	**CHIN[0]** == 0
Slave post-deassert	**ATREADY** ← 0	None	**EVENTACK** ← 0, if present	**CHINACK[0]** ← 0, if present

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part D
CoreSight System Architecture

Chapter D1
About the System Architecture

The system architecture specifies:

- Rules that must be followed by all systems that implement CoreSight components.
- Additional information that is required by debuggers to use a CoreSight system.

The system architecture is described in the following chapters:

- Chapter D2 *System Considerations*.
- Chapter D3 *Physical Interface*.
- Chapter D4 *Trace Formatter*.
- Chapter D5 *About ROM Tables*.
- Chapter D6 *Class 0x1 ROM Tables*.
- Chapter D7 *Class 0x9 ROM Tables*.
- Chapter D8 *Topology Detection at the System Level*.
- Chapter D9 *Compliance Requirements*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D2
System Considerations

This chapter describes system aspects that must be considered when integrating CoreSight components into a system. It has the following sections:

- *Clock and power domains* on page D2-114 describes the requirements for the clock and power domain structure that is exposed to debuggers.
- *Control of authentication interfaces* on page D2-115 describes the requirements for the signals in the authentication interface.
- *Memory system design* on page D2-116 describes how to expose CoreSight registers to system software.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D2.1 Clock and power domains

CoreSight can be used in systems with many clock and power domains. CoreSight systems themselves, however, always define the following clock and power domains:

System domain This domain comprises most non-debug functionality. The clock frequencies in this domain can be asynchronous to the other domains and can vary over time in response to varying performance requirements. The clocks can be stopped, and the power can be removed, leading to the loss of all state information.

Debug domain This domain comprises most debug functionality. When debug functionality is not required, the power can be removed or the clocks can be stopped to reduce power consumption.

Always-on domain This domain comprises the power controller and the interface to the debugger. The power is never removed, even when the device is dormant, which enables the debugger to connect to the device even when it is powered down.

When deviating from the default clock and power domains, observe the following rules:

- When implementing extra clock and power domains by subdividing one of the default clock and power domains, make sure that the clock and power domains respond appropriately to requests made using the debug interface. For example, an implementation can have two system clock domains, as long as both domains are permanently accessible whenever a System Power Up request is made.
- When implementing fewer clock and power domains by combining two or more of the default clock and power domains, make sure that all requests made using the debug interface are operational. For example, when combining the system and debug power domains, the combined domain must always be powered up whenever a System Power Up request or a Debug Power Up request is made.

The debugger can use the debug interface to make the following requests to the system:

- Power up everything in the system domain. When this request is made, all logic in the system domain must be kept permanently powered up, and be continuously accessible to the debugger.
- Power up everything in the debug domain. When this request is made, all logic in the debug domain must be kept permanently powered up, and be continuously accessible to the debugger.
- Reset everything in the debug domain. When this request is made, all logic in the debug domain must be reset to its initial state.

The debugger interface is managed by an implementation of the *ARM Debug Interface* (ADI). For more information, see the appropriate CoreSight Technical Reference Manual.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D2.2 Control of authentication interfaces

A CoreSight system prevents unauthorized debugging by disabling debug functionality, rather than by preventing access to the debug registers. This mechanism is controlled by the authentication interface. For more information about the authentication interface, see Chapter C5 *Authentication Interface*.

Each signal can be driven in one of the following ways:

- Tied LOW. This method is most appropriate for production systems where the specified debug functionality is not required, and prevents in-the-field debugging. There is usually an alternative development chip with the same functionality enabled.
- Tied HIGH. This method is most appropriate for prototype or development systems where authentication is not required.
- Connected to a fuse that is blown in production parts to disable debug functionality, which prevents in-the-field debugging.
- Driven by a custom authentication module, that unlocks debug functionality after a successful authentication sequence. This method provides the most flexibility. In systems where high security is required, ARM recommends using a challenge-response mechanism that is based on an on-chip random number generator or a hardware key unique to that device.

When secure debugging is enabled, secure operations are visible to the outside world, and sometimes to software running in the Non-secure world.

ARM recommends that devices are split into development and production devices:

- Development devices can have secure debugging enabled by authorized developers. All secure data must be replaced by test data suitable for development purposes, where losses are minimal if the test data is disclosed.
- Production devices can never have secure debugging enabled. These devices are loaded with the real secure data.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D2.3 Memory system design

This section describes how to expose CoreSight registers to system software.

D2.3.1 Debug APB interface memory map

Components and the interconnect are not required to differentiate between external and internal accesses unless one of the following applies:

- The component implements the Software lock mechanism that is described in the programmers' model, see *LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.

  Note

  From v3.0 onwards, implementation of the Software lock is deprecated. If a component implements the Software lock, but is accessed using an interconnect that does not support indicating the difference between external and internal accesses, ARM strongly recommends that the component is configured as follows:

  — LSR is RAZ to indicate that the Software lock mechanism is not implemented. See also *LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.

  — PADDRDBG[31] is tied to HIGH to force all accesses to be interpreted as external accesses. See also *AMBA APB interface signals* on page C2-76.

- The component implements an OS lock mechanism, for example an Embedded Trace Macrocell (ETM).

For the limited set of components that must differentiate between external and internal accesses, ARM recommends that two views of the component are provided in the memory system, one for internal accesses, and one for external accesses:

- The two views can be located anywhere in the Debug APB interface address space.
- ARM strongly recommends that a ROM Table provides a pointer only to the external view.
- ARM recommends that one of the address bits is used to differentiate the views.

  An example of this configuration is shown in Figure D2-1, where a 4KB component of a PE at address `0xB00000000` uses two adjacent views at addresses `0x00002000` and `0x00003000` in the APB memory map. Accesses to addresses between `0x00002000` and `0x00002FFF`, for which address bit[12] has a value of `0b0`, are external accesses, and accesses to addresses between `0x00003000` and `0x00003FFF`, for which address bit[12] has a value of `0b1`, are internal accesses.

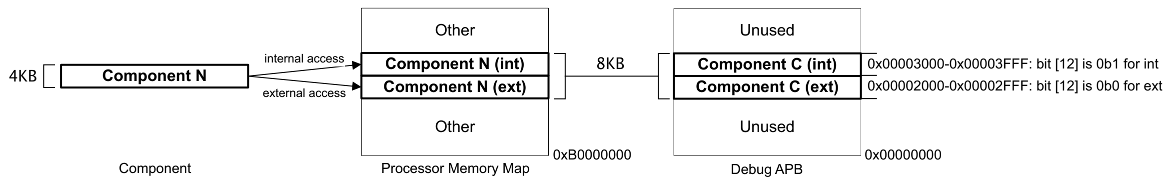


Figure D2-1 Example of an AMBA APB interface memory map for a component with external and internal views

Note

From v3.0 onwards, the use of PADDRDBG[31] to split the memory map into two 2GB segments to indicate the difference between external and internal accesses is deprecated. An older implementation having its external and internal views 2GB apart can be regarded as a special case of a v3.0 implementation: one that uses address bit[31] to differentiate between external and internal accesses.

If a component requires more than one view, and those views are not related to the external and internal views described in this section, the arrangement of those views in the memory system is IMPLEMENTATION DEFINED.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Examples of implementations of a system with three components are shown in Figure D2-2 and Figure D2-3 on page D2-118:

- In the example that is shown in Figure D2-2, one of the components, Component C, requires differentiation between external and internal accesses. The component provides two adjacent views allowing external accesses at addresses for which bit[12] equals `0b0`, and internal accesses at addresses for which bit[12] equals `0b1`.
- In the example that is shown in Figure D2-3 on page D2-118, all three components require differentiation between external and internal accesses. As in the example that is shown in Figure D2-2, address bit[12] is used to differentiate between external and internal accesses, leading to a configuration where the external views are interleaved with the internal views.

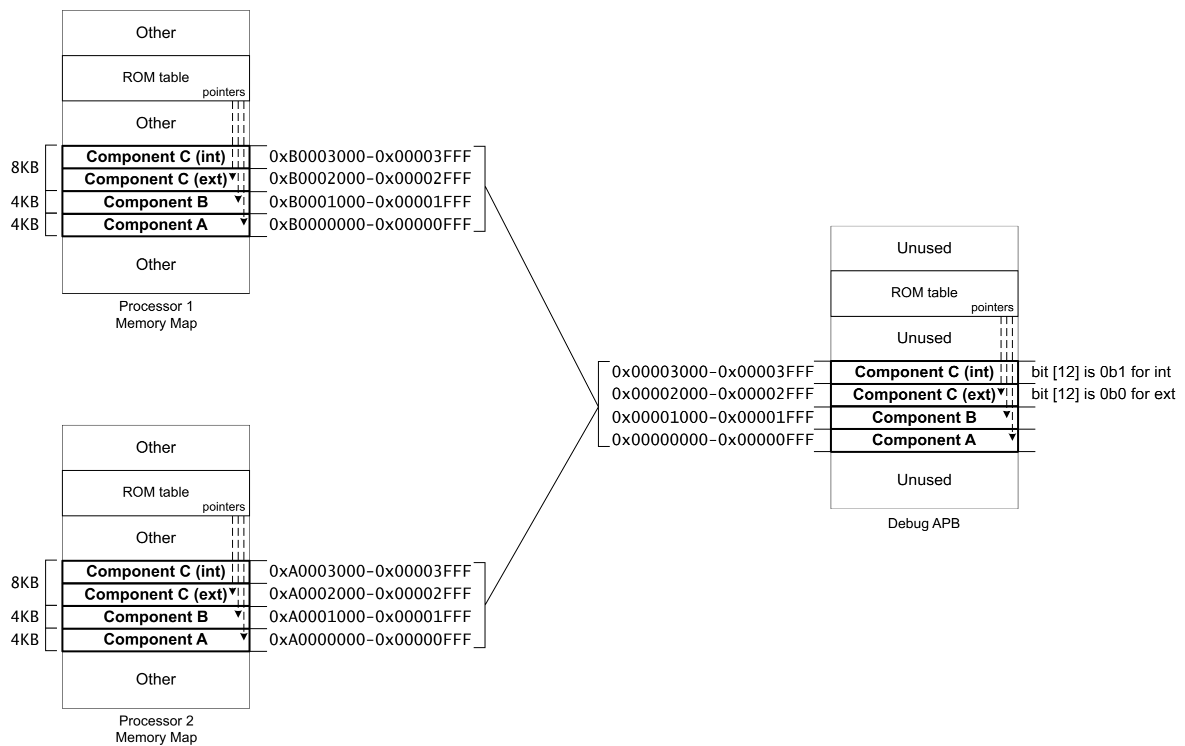


Figure D2-2 Example that includes one component with external and internal views

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

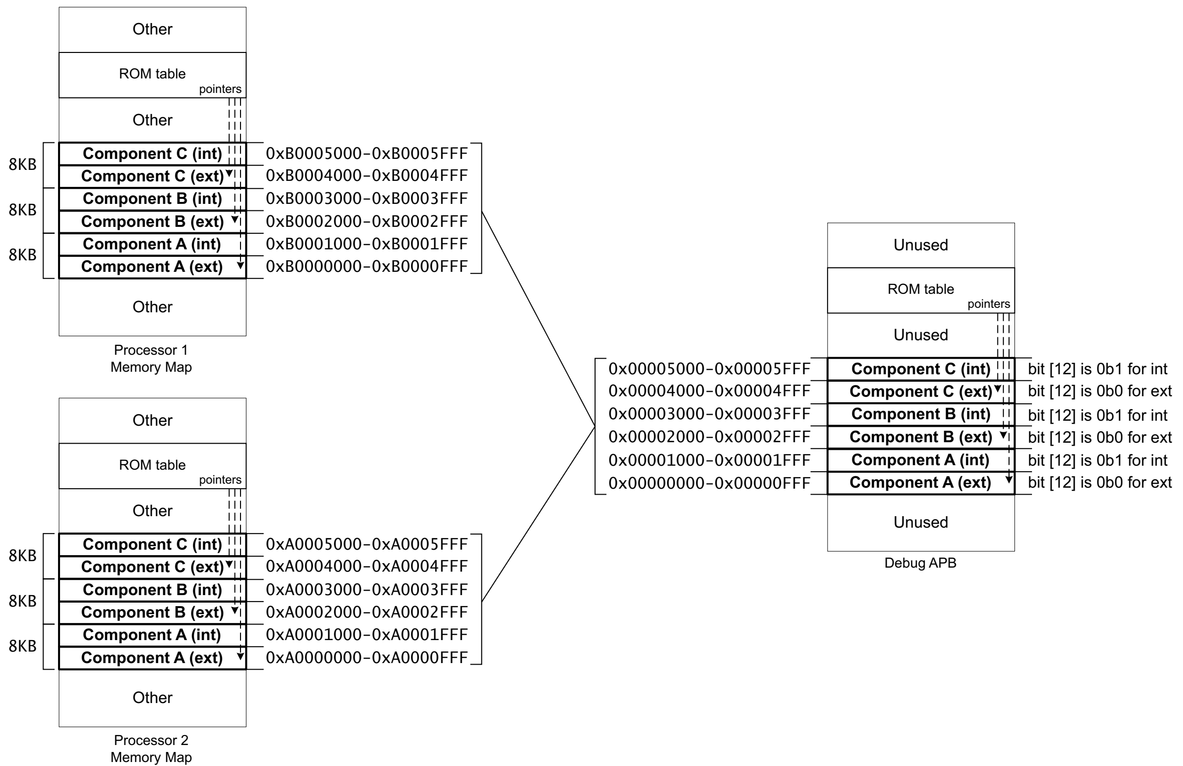


Figure D2-3 Example that includes three components with external and internal views

D2.3.2 Access to the Debug APB interface

When designing a CoreSight system, ensure that the registers of CoreSight components are visible to privileged software.

Note

Do not prevent Non-secure software from accessing the registers of CoreSight components, even if those components can debug secure software. Doing so seriously restricts debugging of Non-secure software.

For system masters that support privileged and unprivileged modes, ARM recommends the following:

- If the master does not have a Memory Management Unit (MMU) or Memory Protection Unit (MPU), the system prevents access to the CoreSight components by unprivileged software.
- If the master does have an MMU or MPU, the MMU or MPU is used to prevent access to the CoreSight components by unprivileged software.
- CoreSight components are marked as Device memory.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D3
Physical Interface

This chapter describes the external pin interface, timing, and connector type that is required for the trace port on a target system. It contains the following sections:

- *ARM JTAG 20* on page D3-120.
- *CoreSight 10 and CoreSight 20 connectors* on page D3-122.
- *ARM MICTOR* on page D3-126.
- *Target Connector Signal details* on page D3-131.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.1 ARM JTAG 20

The ARM JTAG 20 connector is a 20-way 2.54mm pitch connector. It supports the following interfaces:

- JTAG interface, which is based on IEEE 1149.1-1990 and includes the ARM **RTCK** signal.
- *Serial wire debug* (SWD) interface.
- *Serial Wire Output* (SWO) interface.

Figure D3-1 shows the ARM JTAG 20 connector pinout.

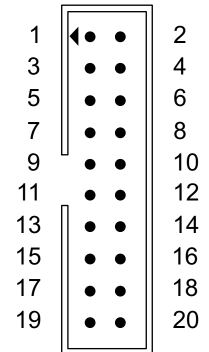


Figure D3-1 ARM JTAG 20 connector pinout

Table D3-1 shows the ARM JTAG 20 pinout as used on the target board.

Table D3-1 ARM JTAG 20 interface pinout table

Pin	Signal name
1	VTREF
2	NC
3	**nTRST**
4	GND
5	**TDI**
6	GND
7	**TMS/SWDIO**
8	GND
9	**TCK/SWCLK**
10	GND
11	**RTCK**
12	GND
13	**TDO/SWO**
14	GND
15	**nSRST**
16	GND
17	**DBGRQ/TRIGIN**

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D3-1 ARM JTAG 20 interface pinout table (continued)

Pin	Signal name
18	GND
19	**DBGACK/TRIGOUT**
20	GND

See *Target Connector Signal details* on page D3-131 for a description of the signals in Table D3-1 on page D3-120.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.2 CoreSight 10 and CoreSight 20 connectors

The CoreSight 10 and CoreSight 20 connectors are used for debug targets requiring JTAG, SWD, SWO, and low-bandwidth trace connectivity.

This section describes 10-way and 20-way connectors that are mounted on debug target boards. These connectors are specified as 0.050 inch pitch two-row pin headers, Samtec FTSH or equivalent. For more information, see the Samtec website, `www.samtec.com`.

The connectors are grouped into compatible sets according to the functions they support. Some targets support communication by both SWD and JTAG using the SWJ-DP block to switch between protocols.

Note

Some of the tables in this section have a column that is named MICTOR, which lists the equivalent pin numbers on a CoreSight-compatible *Matched Impedance Connector* (MICTOR) connector for the named CoreSight pin. A target can feature both CoreSight and MICTOR connectors, where the signals are connected in parallel as suggested by the pinout tables. This configuration enables the target to be debugged using either physical connector.

The connector pin layouts are described in:

- *Combined CoreSight 10 and CoreSight 20 pin names*.
- *CoreSight 10 pinouts* on page D3-123.
- *CoreSight 20 pinouts including trace* on page D3-124.

D3.2.1 Combined CoreSight 10 and CoreSight 20 pin names

Table D3-2 summarizes the combined pin names for the CoreSight 10 and CoreSight 20 connectors.

Table D3-2 Summary of pin names

Pin	Combined pin name	Pin	Combined pin name
1	VTref	11	Gnd/TgtPwr+Cap
2	**TMS/SWDIO**	12	**TraceClk/RTCK**
3	GND	13	Gnd/TgtPwr+Cap
4	**TCK/SWCLK**	14	**TraceD0/SWO**
5	GND	15	GND
6	**TDO/SWO/EXTa/TraceCtl**	16	**TraceD1/nTRST**
7	Key	17	GND
8	**TDI/EXTb**	18	**TraceD2/TrigIn**
9	GNDDetect	19	GND
10	**nSRST**	20	**TraceD3/TrigOut**

See *Target Connector Signal details* on page D3-131 for a description of the signals in Table D3-2.

The following sections describe the use of pins 6, 8, 11, and 13:

- *EXTa and EXTb pins* on page D3-123.
- *GND/TgtPwr+Cap pins* on page D3-123.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

EXTa and EXTb pins

Some pins on the connector have functions that are only used for certain connection layouts. Where a pin is not required for a particular debug communication protocol, it can be reused for a user-defined target function. These pins are labeled **EXTa** and **EXTb** in the tables in this chapter. Select any alternate functions carefully, and consider the effects of connecting debug equipment that is capable of communicating using multiple protocols.

GND/TgtPwr+Cap pins

There are two usage options for these pins:

Target boards

Standard target boards can connect these two pins directly to signal ground, GND.

Some special target boards, for example, boards that are used for evaluation or demonstration purposes, can use these pins to supply power to the target board. In this case, the target board must include capacitors between each of the pins and signal ground. The capacitors must be situated extremely close to the connector so that they maintain an effective AC ground, that is, high frequency signal return path. Typical values for the capacitors are 10nF.

Debug equipment

Debug communication equipment that is designed to work with special valuation or demonstration target boards provides an operating current, typically up to 100mA, at a target-specific supply voltage, for example, 3.3V, 5V, or 9V. ARM recommends that the debug equipment includes some protection when it is connected to standard target boards that have connected these pins directly to GND, for example, a current limiting circuit. This debug equipment must include capacitors between each of these power pins and signal ground. The capacitors must be situated extremely close to the connector so that they maintain an effective AC ground, ensuring a high frequency signal return path. Typical values for the capacitors are 10nF.

Standard debug communication equipment can connect these pins directly to GND. It is also possible for these pins to have only a high frequency signal return path to ground, using 10nF capacitors. This option is also compatible in the unlikely case where a target board has a connection between the debug connector **TgtPwr** pins, but is powered from another source.

D3.2.2 CoreSight 10 pinouts

There are two pinouts for a 10-pin connector:

- Pinout that supports communication using SWD.
- Pinout for JTAG.

The pinouts are arranged to facilitate dynamic switching between the protocols.

Note

SWD is the preferred protocol for debugging because it provides more data bandwidth over fewer pins, which increases the bandwidth that is available to application functions. JTAG can be used where the target is communicating with a tool chain that does not support SWD, or with test tools performing board-level boundary scan testing, where it might be acceptable to sacrifice the functional pins multiplexed with JTAG.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D3-3 shows the CoreSight 10 for targets using SWD or JTAG for debug communication, and includes an optional *Serial Wire Output* (SWO) signal for conveying application and instrumentation trace.

Table D3-3 CoreSight 10 for SWD or JTAG systems

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	10-way	MICTOR		10-way	MICTOR
VTref	1	12	VTref	1	12
SWDIO	2	17	**TMS**	2	17
GND	3	-	GND	3	-
SWCLK	4	15	**TCK**	4	15
GND	5	-	GND	5	-
SWO	6	11	**TDO**	6	11
Key	7	-	Key	7	-
NC/EXTb	8	19	**TDI**	8	19
GNDDetect	9	-	GNDDetect	9	-
nSRST	10	9	**nSRST**	10	9

The SWD layout is typically used in a CoreSight system that uses an SWJ-DP operating in SWD mode.

The JTAG layout is typically used in a CoreSight system that includes a JTAG-DP, or a system with an SWJ-DP operating in JTAG mode, possibly because it is cascaded with other JTAG TAPs.

Note

- A target board can use this connector for performing board-level boundary scans but then switch its SWJ-DP into SWD mode for debugging according to the layout shown in Table D3-3. This method frees up pins 6 and 8 for either application functions or SWO.
- It is not necessary to choose the switching mode at the time of chip or board development. The connector can be switched and the target board can be operated in either SWD or JTAG mode.

D3.2.3 CoreSight 20 pinouts including trace

20-way connectors include support for a narrow trace port of up to four data bits, operating at moderate speeds of up to 100MSamples/sec.

Table D3-4 on page D3-125 shows the CoreSight 20 for targets using SWD or JTAG for debug communication, and includes an optional **SWO** signal for conveying application or instrumentation trace. Alternatively, a target trace port operating in CoreSight normal or bypass modes might convey the **TraceCtl** signal on pin 6.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Both pin 6 and pin 8 in the SWD layout are shown with alternative extra signals, **EXTa** and **EXTb**, which provides flexibility to communicate other signals on these pins. For example, future target systems and trace equipment might convey two more trace data signals on these pins.

Table D3-4 CoreSight 20 for future SWD or JTAG systems

Pin name for SWD	Pin number		Pin name for JTAG	Pin number	
	20-way	MICTOR		20-way	MICTOR
VTref	1	12	VTref	1	12
SWDIO	2	17	**TMS**	2	17
GND	3	-	GND	3	-
SWCLK	4	15	**TCK**	4	15
GND	5	-	GND	5	-
SWO/EXTa/TraceCtl	6	11	**TDO**	6	11
Key	7	-	Key	7	-
NC/EXTb	8	(19)	**TDI**	8	19
GNDDetect	9	-	GNDDetect	9	-
nSRST	10	9	**nSRST**	10	9
Gnd/TgtPwr+Cap	11	-	Gnd/TgtPwr+Cap	11	-
TraceClk	12	6	**TraceClk**	12	6
Gnd/TgtPwr+Cap	13	-	Gnd/TgtPwr+Cap	13	-
TraceD0	14	38	**TraceD0**	14	38
GND	15	-	GND	15	-
TraceD1	16	28	**TraceD1**	16	28
GND	17	-	GND	17	-
TraceD2	18	26	**TraceD2**	18	26
GND	19	-	GND	19	-
TraceD3	20	24	**TraceD3**	20	24

The SWD layout is typically used in a CoreSight system that uses an SWJ-DP operating in SWD mode.

The JTAG layout is typically used in a CoreSight system that includes a JTAG-DP, or a system with an SWJ-DP operating JTAG mode, possibly because it is cascaded with other JTAG TAPs. This layout is the recommended debug connection for a processor that is built with support for instruction trace, for example, a processor that includes an ETM.

Note

- A target board can use this layout for performing board-level boundary scans but then switch its SWJ-DP into SWD mode for debugging according to the layout shown in Table D3-4. This method frees up pins 6 and 8 for either application functions or SWO.
- It is not necessary to choose the switching mode at the time of chip or board development. The connector can be switched and the target board can be operated in either SWD or JTAG mode.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.3 ARM MICTOR

The following sections describe:

- *Target system connector*.
- *Target connector description*.
- *Decoding requirements for Trace Capture Devices* on page D3-129.
- *Electrical characteristics* on page D3-130.

D3.3.1 Target system connector

The specified target system connector is the AMP MICTOR. This connector supports:

- JTAG interface, which is based on IEEE 1149.1-1990 and includes the ARM **RTCK** signal.
- Trace port interface, with up to 16 data pins.
- *Serial Wire Debug* (SWD) interface.
- *Serial Wire Output* (SWO) interface.
- Optional power supply pin.
- Reference voltage pin to enable support of a range of target voltages.
- Optional system reset request pin.
- Optional extra trigger pins for communicating with the target.

For tracing with large port widths that have more than 16 data pins, two connectors are required. See *Single target connector pinout* on page D3-127 and *Dual target connector pinout* on page D3-128.

The AMP MICTOR connector is a high-density matched-impedance connector. This connector has several important attributes:

- Direct connection to a logic analyzer probe using a high-density adapter cable with termination, for example HPE5346A from Agilent.
- Matching impedance characteristics, enabling the connector to be used at high speeds.
- Many ground fingers to ensure good signal integrity.
- Inclusion of one or both of the JTAG and SWD runtime control signals, enabling a single debug connection to the target.

Table D3-5 lists the AMP part numbers for the four possible connectors.

Table D3-5 Connector part numbers

AMP part number	Description
2-767004-2	Vertical, surface mount, board to board or cable connectors
767054-1	
767061-1	
767044-1	Right angle, straddle mount, board to board or cable connector

D3.3.2 Target connector description

This section contains details of the physical layout of the connector and recommended board orientation as follows:

- *Single target connector pinout* on page D3-127.
- *Dual target connector pinout* on page D3-128.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Single target connector pinout

Figure D3-2 shows how the connector and PCB can be oriented on the target system, as seen from above the PCB. The trace connector is mounted near the edge of the board to minimize the intrusiveness of the TPA when the interconnect is a direct PCB to PCB link.

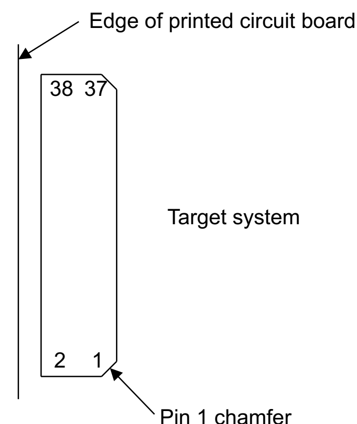


Figure D3-2 Recommended orientation for a single connector

Table D3-6 shows the connections on a single MICTOR connector, and indicates the differences with the connections specified in the *ARM® Embedded Trace Macrocell Architecture Specification*. If a signal is not implemented on the target system, it must be replaced with a logic 0 connection.

Table D3-6 Single target connector pinout

Pin	Signal	Pin	Signal
38	**TRACEDATA[0]**	37	**TRACEDATA[8]**
36	**TRACECTL**	35	**TRACEDATA[9]**
34	Logic 1	33	**TRACEDATA[10]**
32	Logic 0	31	**TRACEDATA[11]**
30	Logic 0	29	**TRACEDATA[12]**
28	**TRACEDATA[1]**	27	**TRACEDATA[13]**
26	**TRACEDATA[2]**	25	**TRACEDATA[14]**
24	**TRACEDATA[3]**	23	**TRACEDATA[15]**
22	**TRACEDATA[4]**	21	**nTRST**
20	**TRACEDATA[5]**	19	**TDI**
18	**TRACEDATA[6]**	17	**TMS**/**SWDIO**
16	**TRACEDATA[7]**	15	**TCK**/**SWCLK**
14	VSupply	13	**RTCK**
12	VTRef	11	**TDO**/**SWO**
10	No connection, was **EXTTRIG**	9	**nSRST**
8	**TRIGOUT**/**DBGACK**	7	**TRIGIN**/**DBGRQ**

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D3-6 Single target connector pinout (continued)

Pin	Signal	Pin	Signal
6	**TRACECLK**	5	GND
4	No connection	3	No connection
2	No connection	1	No connection

Dual target connector pinout

Figure D3-3 shows the arrangement for two connectors. ARM recommends that they are placed in line, with pins 1 separated by 1.35inches.

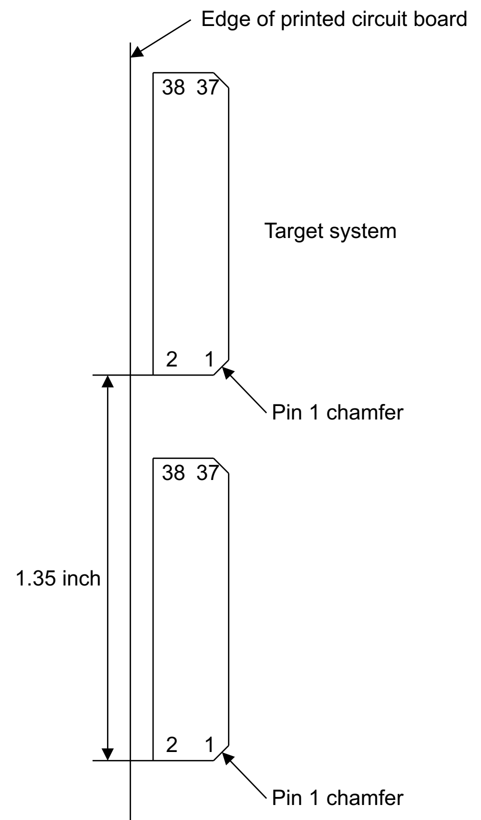


Figure D3-3 Recommended orientation for two connectors

Table D3-7 shows the connections for the secondary MICTOR connector. For the primary connector, see Table D3-6 on page D3-127. If a signal is not implemented on the target system, it must be replaced with a logic 0 connection.

Table D3-7 Dual target connector pinout

Pin	Signal	Pin	Signal
38	**TRACEDATA[16]**	37	**TRACEDATA[24]**
36	Logic 0	35	**TRACEDATA[25]**
34	Logic 1	33	**TRACEDATA[26]**
32	Logic 0	31	**TRACEDATA[27]**

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D3-7 Dual target connector pinout (continued)

Pin	Signal	Pin	Signal
30	Logic 0	29	**TRACEDATA[28]**
28	**TRACEDATA[17]**	27	**TRACEDATA[29]**
26	**TRACEDATA[18]**	25	**TRACEDATA[30]**
24	**TRACEDATA[19]**	23	**TRACEDATA[31]**
22	**TRACEDATA[20]**	21	No connection
20	**TRACEDATA[21]**	19	No connection
18	**TRACEDATA[22]**	17	No connection
16	**TRACEDATA[23]**	15	No connection
14	No connection	13	No connection
12	VTRef	11	No connection
10	No connection	9	No connection
8	No connection	7	No connection
6	**TRACECLK**	5	GND
4	No connection	3	No connection
2	No connection	1	No connection

D3.3.3 Decoding requirements for Trace Capture Devices

Table D3-8 shows the conditions that must be decoded by *Trace Capture Devices* (TCDs), for example a TPA or a logic analyzer.

Table D3-8 Trace Capture Device decoding

TRACECTL	TRACEDATA[0]	TRACEDATA[1]	Trigger	Capture	Description
0	x	x	No	Yes	Normal trace data
1	0	0	Yes	Yes	Trigger packet
1	0	1	Yes	No	Trigger
1	1	x	No	No	Trace disable

Normal trace data

When trace data is indicated, only the full field of **TRACEDATA[n:0]** has to be stored. **TRACECTL** can be discarded to permit more efficient packing of data within the TCD.

Trigger packet

Although CoreSight does not use the encoding for a trigger packet, it must be implemented to maintain cross compatibility with ETMv3.x trace ports. **TRACEDATA** signals must be stored because there is further information that is emitted on this cycle, but the **TRACECTL** signal can be discarded. For more information, see the *ARM® Embedded Trace Macrocell Architecture Specification*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Trigger

A trigger is used as a marker to enable the TCD to stop capture after a predetermined number of cycles. No data is output on this cycle, and **TRACEDATA[n:0]** and **TRACECTL** must not be captured.

Trace disable

This signal indicates that the current cycle must not be captured because it contains no useful information.

D3.3.4 Electrical characteristics

Debug equipment must be able to deal with a wide range of signal voltage levels. Typical ASIC operating voltages can range from 1V to 5V, although 1.8V to 3.3V is common.

It is important to keep the track length differences as small as possible to minimize skew between signals. Crosstalk on the trace port must be kept to a minimum because it can cause erroneous trace results. To minimize the chance of unpredictable responses, avoid stubs on the trace lines, especially at high frequencies. If stubs are necessary, make them as small as possible.

The trace port clock line, **TRACECLK** must be series-terminated as close as possible to the pins of the driving ASIC.

The maximum capacitance that is presented by the trace connector, cabling and interfacing logic must be less than 15pF.

There are no inherent restrictions on operating frequency, other than ASIC pad technology and TPA limitations. It is mandatory, however, to observe the following guidelines for maximizing the speed at which trace capture is possible.

Figure D3-4 shows the timing for **TRACECLK**.

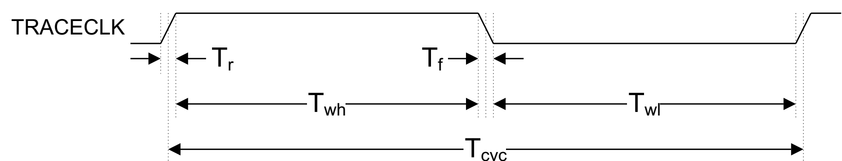


Figure D3-4 TRACECLK specification

ARM recommends that trace ports provide a **TRACECLK** that is as symmetrical as possible, because both edges are used to capture trace. Figure D3-5 shows the setup and hold requirements for the trace data pins, **TRACEDATA[n:0]** and **TRACECTL**, in relation to **TRACECLK**.

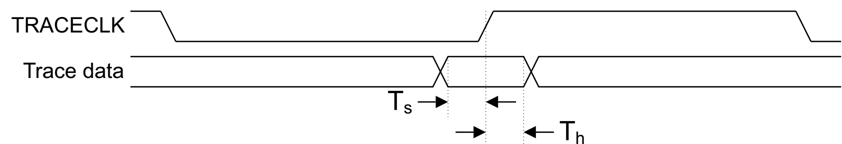


Figure D3-5 Trace data specification

ARM recommends observing the following principles:

- Make sure that the setup time T_s and the hold time T_h are as large as possible, and make sure that both are positive, which is required by some TPAs.
- Allow the TPAs to delay each trace data signal individually by up to a whole clock period, to compensate for trace ports where T_s and T_h are not balanced or vary between data signals.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4 Target Connector Signal details

The signals on the target connector pins are:

- *VTRef output*.
- *TRACECLK output*.
- *TRACECTL output*.
- *TRACEDATA[n:0] output*.
- *Logic one input* on page D3-132.
- *Logic 0 input* on page D3-132.
- *TRIGIN/DBGRQ input* on page D3-132.
- *TRIGOUT/DBGACK output* on page D3-132.
- *nSRST input* on page D3-132.
- *nTRST input* on page D3-132.
- *TDI input* on page D3-132.
- *TMS input* on page D3-132.
- *SWDIO input/output* on page D3-132.
- *TCK/SWCLK input* on page D3-132.
- *RTCK output* on page D3-132.
- *TDO output* on page D3-133.
- *SWO output* on page D3-133.
- *VSupply output* on page D3-133.
- *GND* on page D3-133.
- *No connection* on page D3-133.

D3.4.1 VTRef output

The VTRef signal is intended to supply a logic-level reference voltage to enable debug equipment to adapt to the signaling levels of the target board.

Note

VTRef does not supply operating current to the debug equipment.

Target boards must supply a voltage of 1-5V ±10%, implying a minimum of 0.9V, and a maximum of 5.5V. The DC output impedance of the target board must be low enough to ensure that the output voltage does not change by more than 1% when supplying a nominal signal current of 0.4mA. Debug equipment that connects to this signal must use it as a signal rather than a power supply pin and not load it more heavily than a signal pin. The recommended maximum source or sink current is 0.4mA.

D3.4.2 TRACECLK output

The trace port must be sampled on both edges of the TRACECLK clock signal. There is no requirement for TRACECLK to be linked to a core clock.

D3.4.3 TRACECTL output

This signal, together with **TRACEDATA[1:0]**, indicates whether trace information can be stored this cycle. It is not necessary to store **TRACECTL**.

D3.4.4 TRACEDATA[n:0] output

This signal can be any size and represents the data that is generated from the trace system. To decompress the data, an understanding of the data stream is required, because the data can be wrapped up within the Formatter protocol or consist of direct data from a single trace source. See also Chapter D4 *Trace Formatter*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4.5 Logic one input

This signal pin is at a voltage level that represents logic 1, typically a resistor pull-up to VTRef.

D3.4.6 Logic 0 input

This signal pin is at a voltage level that represents logic 0, typically a resistor pull-down to GND.

D3.4.7 TRIGIN/DBGRQ input

This signal is used to change the behavior of on-chip logic, for example by connecting it to a Cross Trigger Interface. ARM recommends that this pin is pulled to a defined state, LOW, to avoid unintentional requests to any connected on-chip logic.

D3.4.8 TRIGOUT/DBGACK output

This signal can be connected to on-chip trigger generation logic such as a Cross Trigger Interface to enable events to be propagated to external devices.

D3.4.9 nSRST input

This signal is an open-collector output from the run control unit to the target system reset, or an input to the run control unit so that a reset initiated on the target can be reported to the debugger.

On the target, pull up this pin to HIGH to avoid unintentional resets when there is no connection.

For more details on the use of **nSRST**, see the *ARM® Debug Interface Architecture Specification*.

D3.4.10 nTRST input

The **nTRST** signal is an open-collector input from the run control unit to the **Reset** signal on the target JTAG port. On the target, pull up this pin to HIGH to avoid unintentional resets when there is no connection.

D3.4.11 TDI input

TDI is the Test Data In signal from the run control unit to the target JTAG port. ARM recommends that this pin is set to a defined state.

D3.4.12 TMS input

TMS is the Test Mode Select signal from the run control unit to the target JTAG port. On the target, this pin must be pulled up to HIGH to ensure that when there is no connection, the effect of any spurious **TCK**s is benign. For connectors that share JTAG and SWD signals, this pin is shared with the **SWDIO** signal.

D3.4.13 SWDIO input/output

The Serial Wire Data I/O pin sends and receives serial data to and from the target during debugging. For connectors that share JTAG and SWD signals, this pin is shared with the **TMS** signal.

D3.4.14 TCK/SWCLK input

TCK/**SWCLK** is the Test Clock signal from the run control unit to the target JTAG or SWD port. ARM recommends that this pin is pulled to a defined state.

D3.4.15 RTCK output

RTCK is the Return Test Clock signal from the target JTAG port to the run control unit. Some targets, such as ARM7TDMI-S™, must synchronize the JTAG port to internal clocks. To facilitate meeting this requirement, use a returned, and retimed, **TCK** to dynamically control the **TCK** rate.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4.16 TDO output

TDO is the Test Data Out from the target JTAG port to the run control unit. This signal must be set to its inactive drive state, tristate, when the JTAG state machine is not in the Shift-IR or Shift-DR states. For connectors that share JTAG and SWD signals, this pin is shared with the **SWO** signal.

D3.4.17 SWO output

The Serial Wire Output pin can be used to provide trace data. For connectors that share JTAG and SWD signals, this pin is shared with the **TDO** signal.

D3.4.18 VSupply output

The VSupply pin enables the target board to supply operating current to debug equipment so that an external power supply is not required.

- This pin might not be used by all debug equipment.
- The V_{DD} power rail typically drives the pin on the target board.
- Target board documentation indicates the VSupply pin voltage and the current available. Target boards must supply a voltage that is nominally between 2V and 5V with a tolerance of ±10%, amounting to a minimum of 1.8V and a maximum of 5.5V. A target board that drives this pin must provide a minimum supply current of 250mA, where 400mA is recommended.
- To enable establishing the need for an external power supply to power the debug equipment, debug equipment must indicate the required supply voltage range and the current power consumption over that range.
- Target boards might have a limited amount of current available for external debug equipment, so a backup mechanism to power the debug equipment must be provided in case VSupply is not connected or insufficient.

For some hardware, this pin is unused.

D3.4.19 GND

This pin must be connected to 0V on the target board to provide a signal return and logic reference.

D3.4.20 No connection

No connection must be made to this pin.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D4
Trace Formatter

This chapter describes trace formatter requirements for devices that comply with the CoreSight architecture. It contains the following sections:

- *About trace formatters* on page D4-136.
- *Frame descriptions* on page D4-137.
- *Modes of operation* on page D4-142.
- *Flush of trace data at the end of operation* on page D4-143.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D4.1 About trace formatters

Formatters are methods for wrapping Trace Source IDs into the output Trace Data stream. This chapter specifies the format that is used by Trace Sinks to embed AMBA ATB interface source IDs into a single trace stream. For more information about the AMBA ATB protocol, see *AMBA ATB interface* on page C2-78.

The formatter protocol has the following features:

- It permits trace from several sources to be merged into a single stream and later separated.
- It does not place any requirements or constraints on the data that is emitted from trace sources.
- It is suitable for high-speed real-time decoding.
- It can be transmitted and stored as a bitstream without the need for separate alignment information.
- It can be decoded even if the start of the trace is lost.
- It indicates the position of the trigger signal around which trace capture is centered to the TPA, eliminating the need for a separate pin.
- It indicates when the trace port is inactive to the TPA, eliminating the need for a separate flow control pin.

If the embedded trigger and flow control information is not required by the TPA, and only a single trace source is used, it is possible to disable the formatting to achieve better data throughput.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D4.2 Frame descriptions

The formatter protocol outputs data in 16-byte frames. Each frame consists of:

- Seven bytes of data.
- Eight mixed-use bytes, each of which contains:
  - One bit to indicate the use of the remaining seven bits.
  - Seven bits that can be data or a change of trace source ID.
- One byte of auxiliary bits, where each bit corresponds to one of the eight mixed-use bytes:
  - If the corresponding byte was data, this bit indicates the remaining bit of that data.
  - If the corresponding byte was an ID change, this bit indicates when that ID change takes effect.

Figure D4-1 shows how these bytes are arranged in a formatter frame.

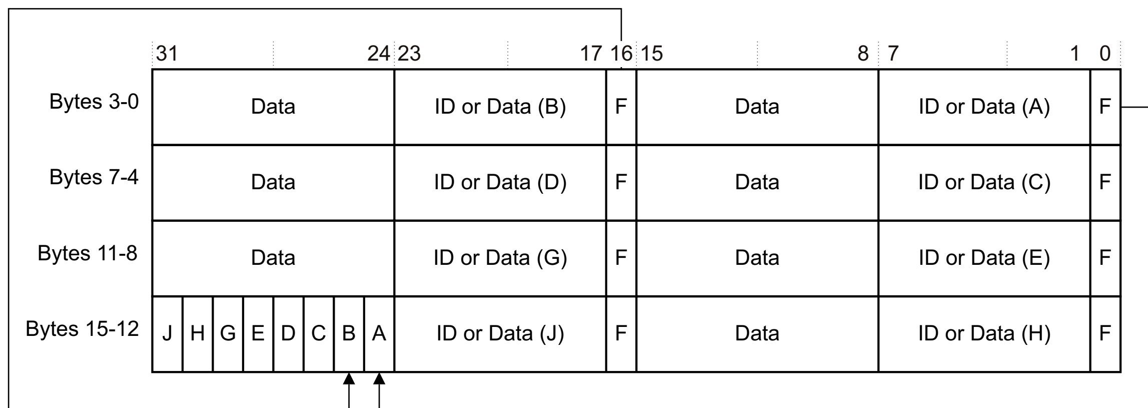


Figure D4-1 Formatter frame structure

Each time the ID changes, at least 1 byte of data must be output for that ID. Table D4-1 shows the meaning of each bit in a formatter frame. It is output least significant bit first, starting with bit 0.

Table D4-1 Meaning of bits in a formatter frame

Byte number	Bits	Description
0	0	ID or Data control for bits[7:1], marked F in Figure D4-1.
	7:1	Depends on bit 0: 0 = Data[7:1]. 1 = New ID.
1	7:0	Data[7:0].
2	7:0	ID or Data, see byte 0.
3	7:0	Data[7:0].
4	7:0	ID or Data, see byte 0.
5	7:0	Data[7:0].
6	7:0	ID or Data, see byte 0.
7	7:0	Data[7:0].
8	7:0	ID or Data, see byte 0.
9	7:0	Data[7:0].

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-1 Meaning of bits in a formatter frame (continued)

Byte number	Bits	Description
10	7:0	ID or Data, see byte 0.
11	7:0	Data[7:0].
12	7:0	ID or Data, see byte 0.
13	7:0	Data[7:0].
14	7:0	ID or Data, see byte 0.
15	0	Auxiliary bit for byte 0, see bit in Figure D4-1 on page D4-137 marked A. The meaning of this bit depends on whether byte 0 contains data or a new ID: Data = Data[0]. New ID: 0 = Byte 1 corresponds to the new ID. 1 = Byte 1 corresponds to the old ID. The new ID takes effect from the next data byte after byte 1.
	1	Auxiliary bit for byte 2, marked B in Figure D4-1 on page D4-137. See bit 0. If byte 2 contains a new ID, this bit indicates whether the new ID takes effect from byte 3 or the following data byte.
	2	Auxiliary bit for byte 4, marked C in Figure D4-1 on page D4-137. See bit 0.
	3	Auxiliary bit for byte 6, marked D in Figure D4-1 on page D4-137. See bit 0.
	4	Auxiliary bit for byte 8, marked E in Figure D4-1 on page D4-137. See bit 0.
	5	Auxiliary bit for byte 10, marked G in Figure D4-1 on page D4-137. See bit 0.
	6	Auxiliary bit for byte 12, marked H in Figure D4-1 on page D4-137. See bit 0.
	7	Auxiliary bit for byte 14, marked J in Figure D4-1 on page D4-137. See bit 0. If byte 14 is a new ID, this bit is reserved. It must be zero, and must be ignored when decompressing the frame. The new ID takes effect from the first data byte of the next frame.

D4.2.1 Frame example

Two trace sources with IDs of `0x03` and `0x15` generate trace data and are interleaved on the trace bus, presenting one word of data at a time.

The following stream of bytes is output by the formatter:

`0x07, 0xAA, 0xA6, 0xA7, 0x2B, 0xA8, 0x54, 0x52, 0x52, 0x54, 0x07, 0xCA, 0xC6, 0xC7, 0xC8, 0x1C.`

Figure D4-2 on page D4-139 shows the corresponding frame.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

	31 … 24	23 … 17	16	15 … 8	7 … 1	0
Bytes 3-0	Data 0xA7	Data 0x53	0	Data 0xAA	ID 0x03	1
Bytes 7-4	Data 0x52	Data 0x2A	0	Data 0xA8	ID 0x15	1
Bytes 11-8	Data 0xCA	ID 0x03	1	Data 0x54	Data 0x29	0
Bytes 15-12	0 0 0 1 1 1 0 0	Data 0x64	0	Data 0xC7	Data 0x63	0

Figure D4-2 Example formatter frame

Table D4-2 shows how this frame is decoded.

Table D4-2 Decoding the example formatter frame

Byte	Observation	Interpretation	Data	ID
0	Bit[0] is set.	This byte represents the new ID `0x03`. Bit[0] of byte 15 is clear, so the new ID takes effect immediately.	-	`0x03`
1		Data byte for the trace with the new ID 0x03.	`0xAA`	`0x03`
2	Bit[0] is clear	This byte is a data byte for the trace with the current ID 0x03. Bit[0] of the data is taken from bit[1] of byte 15.	`0xA6`	`0x03`
3		Data byte.	`0xA7`	`0x03`
4	Bit[0] is set	This byte represents the new ID `0x15`. Bit[2] of byte 15 is set, so the next data byte continues to use the current ID 0x03.	-	`0x03`
5		Data byte for the trace with the current ID 0x03.	`0xA8`	`0x03`
6	Bit[0] is clear.	This byte is a data byte for the trace with the new ID 0x15. Bit[0] of the data is taken from bit[3] of byte 15.	`0x55`	`0x15`
7		Data byte.	`0x52`	`0x15`
8	Bit[0] is clear.	This byte is a data byte. Bit[0] of the data is taken from bit[4] of byte 15.	`0x53`	`0x15`
9		Data byte.	`0x54`	`0x15`
10	Bit[0] is set.	This byte represents the new ID `0x03`. Bit[5] of byte 15 is clear, so the new ID takes effect immediately.	-	`0x03`
11		Data byte for the trace with the new ID 0x15.	`0xCA`	`0x03`
12	Bit[0] is clear.	This byte is a data byte for the trace with the current ID 0x15. Bit[0] of the data is taken from bit[6] of byte 15.	`0xC6`	`0x03`
13		Data byte.	`0xC7`	`0x03`
14	Bit[0] is clear.	This byte is a data byte. Bit[0] of the data is taken from bit[7] of byte 15.	`0xC8`	`0x03`
15		Auxiliary bits.	-	-

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D4.2.2 Frame synchronization packet

The frame synchronization packet enables a TPA or trace decompressor to find the start of a frame. It is output periodically between frames. It is output least significant bit first, starting with bit[0]. In continuous mode, the TPA might discard all frame synchronization packets after the start of a frame is found. See *Modes of operation* on page D4-142 for more information about continuous mode.

Figure D4-3 shows a frame synchronization packet.

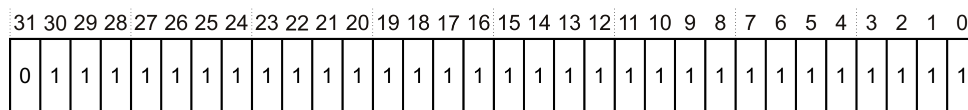


Figure D4-3 Full frame synchronization packet

This sequence cannot occur at any other time, if ID `0x7F` has not been used. See *Special trace source IDs* for more information on reserved source IDs.

Note

Frame synchronization packets and frame data are always multiples of 32-bits, but do not always start on a 32-bit boundary. Because halfword synchronization packets can occur within frames and between frames, they can also start on 16-bit boundaries. See also *Halfword synchronization packet*.

D4.2.3 Halfword synchronization packet

Halfword synchronization packets enable a TPA to detect when the trace port is idle and there is no trace to be captured. They observe the following rules:

- They are output between frames or within a frame.
- If they appear within a frame, they are always aligned to a 16-bit boundary.
- They are output least significant bit first, starting with bit[0].
- They are only generated in continuous mode. If a TPA detects a halfword synchronization packet, it must discard it, because it does not form part of a formatter frame. See *Modes of operation* on page D4-142 for more information about continuous mode.

Figure D4-4 shows a halfword synchronization packet.

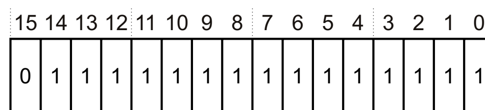


Figure D4-4 Halfword synchronization packet

Note

Frame synchronization packets and frame data are always multiples of 32-bits, but do not always start on a 32-bit boundary. Because halfword synchronization packets can occur within frames and between frames, they can also start on 16-bit boundaries.

D4.2.4 Special trace source IDs

The following IDs are for special purposes and must not be used under normal operation:

`0x00` This ID indicates a NULL trace source. Any data following this ID change must be ignored by the debugger, which is required if there is insufficient trace data available to complete a formatter frame.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`0x70-0x7A` Reserved.

`0x7B` This ID indicates a flush response. Trace that is output with the flush response ID signifies that all trace that was generated previous to a flush request has been output.

Each byte of trace that is output with ID `0x7B` constitutes a separate flush response, whereby the value of the byte can be one of the following:

`0x00` All active trace sources have indicated a flush response.

`0x01-0x6F` The trace source ID with this value has indicated a flush response.

`0x70-0xFF` Reserved.

Note

The use of trace source ID `0x7B` is also permitted on ATB, with the same payload semantics.

`0x7C` Reserved.

`0x7D` This ID indicates a trigger within the trace stream and is accompanied by one byte of data for each trigger. The value of each data byte indicates the ID of the trigger. A data byte with a value of zero indicates that the trigger ID is UNKNOWN.

Note

The use of trace source ID `0x7D` is also permitted on ATB, with the same payload semantics.

`0x7E` Reserved.

`0x7F` This ID must never be used because it conflicts with the synchronization packet encodings.

D4.2.5 Data overheads

The formatter protocol adds an overhead of 6% to the bandwidth requirement of a trace port. It also requires one byte of extra trace every time the source ID changes. ARM recommends that components that arbitrate between trace sources switch between different source IDs as infrequently as possible.

Under certain conditions, the formatter can be bypassed to eliminate this overhead. For more information, see *Bypass* on page D4-142.

D4.2.6 Repeating the trace source ID

If a large amount of consecutive trace is generated by a single source ID, the ID must be repeated periodically. This mechanism ensures that the debugger can determine the source of the trace even when the beginning of the trace has been lost.

ARM recommends repeating the source ID approximately every ten frames.

D4.2.7 Indication of alignment points

In most trace protocols, it is necessary to periodically indicate the beginning of a packet. This process is called alignment synchronization. Most protocols achieve alignment synchronization by periodically outputting a packet similar to the Frame Synchronization Packet.

When using a protocol that is unable to output such a packet, use the formatter protocol to indicate the position of synchronization points by using two source IDs. The trace source starts outputting trace using the first ID, then switches to the second ID at the first alignment point. It switches back to the first ID at the next alignment point, and continues switching at each subsequent alignment point.

A trace source that uses ID switching in this manner cannot use bypass mode. See *Bypass* on page D4-142.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3 Modes of operation

The formatter can operate in one of three modes. Not all modes are supported by all components that implement a formatter. For example, an ETB does not need to support continuous mode.

D4.3.1 Bypass

In this mode, the trace is output without modification. No formatting information is inserted into the trace stream. When using bypass mode, observe the following rules:

- Only one trace source ID is in use.
- If the trace is to be output over a trace port, the **TRACECTL** pin must be implemented, and the TPA must support this pin. This configuration enables the TPA to determine the position of the trigger and detect when no trace is available for capture. See *Decoding requirements for Trace Capture Devices* on page D3-129.
- The debugger does not need to report the position of the trigger as seen by the trace sink. In bypass mode, the trigger ID `0x7D` is not generated.

To ensure that all trace is output from a trace sink when stopping trace, extra data might be added to the end of the trace stream. See *Bypass mode* on page D4-143.

D4.3.2 Normal

The formatter is enabled, and the **TRACECTL** pin is used to determine the position of the trigger and detect when no trace is available for capture. Halfword synchronization packets are not generated. The TPA does not have to decode any part of the trace stream.

This mode is the easiest to support by TPAs designed for ETMs.

D4.3.3 Continuous

The formatter is enabled, but the **TRACECTL** pin is not used. The TPA must decode part of the formatter protocol to determine the position of the trigger and detect when no trace is available for capture.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4 Flush of trace data at the end of operation

To support AMBA ATB protocol flushes, the formatter must ensure that all trace is output, because the trace that remains after a flush might be insufficient to complete a frame or use all the pins in the trace port. This section describes how the remaining trace must be formatted.

Note

- When tracing is resumed, some leftover trace that is generated by the flush sequence might be output before any new trace is output. Look for the first synchronization packet in the protocol before starting to decompress the trace.
- Trace that is output with the flush response ID `0x7B` signifies that all trace that was generated previous to a flush request has been output. See also *Special trace source IDs* on page D4-140.

D4.4.1 Bypass mode

When running in bypass mode, if the formatter cannot guarantee that all trace has been output, it must output an extra sequence at the end of the trace. This mechanism ensures that all trace stored in the formatter is output, even if, for example, there is insufficient trace to use all the pins of a trace port.

The sequence consists of a single bit that is set, followed by a series of zeros. This sequence does not represent real trace data and must always be removed before decompression when the trace sink has been requested to stop trace output.

The following two examples show sequences that can be observed on a 32-bit trace port. Figure D4-5 shows an example of how the last AMBA ATB protocol transaction left three bytes within the formatter.

31 … 24	23 … 16	15 … 8	7 … 0
0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]
0x01	0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]
0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00

Figure D4-5 End of session example 1

Figure D4-6 shows an example of how the trace finishes on a 32-bit trace port boundary.

31 … 24	23 … 16	15 … 8	7 … 0
0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]
0x55 [Real Data]	0xAA [Real Data]	0x55 [Real Data]	0xAA [Real Data]
0x00	0x00	0x00	0x01
0x00	0x00	0x00	0x00

Figure D4-6 End of session example 2

D4.4.2 Normal and continuous mode

When running in normal or continuous mode, the formatter must complete the frame currently being output, using the null ID encoding, ID `0x00`. Any data that is associated with this ID can be ignored. More frames of data corresponding to the null ID can be generated to ensure that all trace has been output.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D5
About ROM Tables

The chapter describes ROM Tables. It includes the following sections:

- *ROM Tables Overview* on page D5-146.
- *ROM Table Types* on page D5-147.
- *Component and Peripheral ID Registers for ROM Tables* on page D5-148.
- *The component address* on page D5-149.
- *Location of the ROM Table* on page D5-150.
- *ROM Table hierarchies* on page D5-151.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.1 ROM Tables Overview

ROM Tables hold information about debug components.

- Systems with a single debug component do not require a ROM Table. However, a designer might choose to implement such a system to include a ROM Table.
- Systems with more than one debug component must include at least one ROM Table.

A ROM Table connects to a bus controlled by a Memory Access Port (MEM-AP). In other words, the ROM Table is part of the address space of the memory system that is connected to a MEM-AP. More than one ROM Table can be connected to a single bus.

A ROM Table always occupies 4KB of memory.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2 ROM Table Types

The following types of ROM Tables are permitted to be used with components that comply with CoreSight v3:

Class 0x1 ROM Tables

In a Class 0x1 ROM Table implementation:

- The Component class field, CIDR1.CLASS, is `0x1`, which identifies the component as a Class 0x1 ROM Table.
- The PIDR4.SIZE field must be 0.
- A ROM Table must occupy a single 4KB block of memory.
- A Class 0x1 ROM Table is a read-only device.

For a detailed description of the Class 0x1 ROM Table entries and registers, see Chapter D6 *Class 0x1 ROM Tables*.

Class 0x9 ROM Tables

In a Class 0x9 ROM Table implementation:

- The Component class field, CIDR1.CLASS, is `0x9`, which identifies the component as a CoreSight Component.
- The DEVTYPE and DEVID registers contain information about the ROM Table, as described in Chapter D7 *Class 0x9 ROM Tables*.
- The PIDR4.SIZE field must be 0.
- A ROM Table must occupy a single 4KB block of memory.
- Class 0x9 ROM Table entries are 32 or 64 bits wide.

For a detailed description of the Class 0x9 ROM Table entries and registers, see Chapter D7 *Class 0x9 ROM Tables*.

Note

Class 0x9 ROM Tables can be used alongside Class 0x1 ROM Tables, and both Class 0x9 and Class 0x1 ROM Tables might be present in systems that comply with CoreSight v3.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3 Component and Peripheral ID Registers for ROM Tables

Any ROM Table must implement a set of Component and Peripheral ID Registers, that start at offset `0xFD0` in the ROM Table. *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40 in Chapter B2 *CoreSight programmers' model* describes these registers. This section only describes particular features of the registers when they relate to a ROM Table.

D5.3.1 Identifying the debug SoC or platform

The top-level ROM Table Peripheral ID Registers uniquely identify the SoC or platform. If a system has more than one Memory Access Port with a connected ROM Table, the information from the Peripheral ID Registers of all the top-level ROM Tables, considered collectively, is required to uniquely identify the SoC or platform.

If there is any change in the set of components that are identified by the ROM Table, or any change in the connections to those components, then the Peripheral ID Registers must be updated to reflect the change.

Each possible configuration must be uniquely identifiable from the Peripheral ID Register values, because the Peripheral ID can be used by the debugger to name the description of the system.

If a debugger performs topology detection on the system that it connects to through an implementation of the *ARM Debug Interface* (ADI), it can save its description of the system with the Peripheral ID. If that system is connected to the debugger again, the debugger can retrieve that saved description, avoiding any requirement for topology detection.

If two different systems have the same Peripheral ID, a debugger might retrieve an incorrect description. If this situation occurs, you must force the debugger to perform topology detection again.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.4 The component address

Each debug component occupies one or more 4KB blocks of address space. The occupied address space is referred to as the Debug Register File for the component.

The Address offset field of a ROM Table entry, ROMENTRY<n>.OFFSET, points to the start of the 4KB block that contains the Peripheral ID and Component ID registers of component *n*. The base address of this 4KB block is calculated using the following equation:

```
Component_n_Address = ROM_Base_Address + (ROMENTRY<n>.OFFSET << 12)
```

The Component and Peripheral ID Registers for component *n* start at Component_n_Address + `0xFD0`.

For a component that occupies more than one 4KB block, the size of the component and the base address of the component are IMPLEMENTATION DEFINED, and might be determined by a combination of the values in the Peripheral ID registers and other IMPLEMENTATION DEFINED registers.

Note

For a component that occupies more than 4KB, the ROM Table entry always points to the 4KB block that contains the Peripheral ID and Component ID registers, and this 4KB might occupy any 4KB block in the Debug Register File of the component.

Previous versions of this specification used the PIDR4.SIZE field to define the size and base address of the component. The use of the PIDR4.SIZE field is deprecated, and ARM recommends that for all components:

- Debuggers ignore the value of PIDR4.SIZE.
- New components set PIDR4.SIZE to zero.

Previous versions of this specification required the Peripheral ID and Component ID registers to occupy the highest 4KB block of the Debug Register File. This requirement is removed.

In general, the ROM Table indicates all the valid addresses in the memory map of the connection to the system being debugged. For more information about accesses to addresses that are not pointed to by the ROM Table, see *MEMTYPE, Memory Type Register* on page D6-162.

ARM recommends that the debug component base address is aligned to the largest translation granule supported by any processor that can access the component, which is up to 64KB for an ARMv8 processor.

For more information about the Component and Peripheral ID Registers, see Chapter B2 *PIDR0-PIDR7*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.5 Location of the ROM Table

This section describes how to provide a pointer to the top-level ROM Table.

While entries in a ROM Table are always relative addresses, the top-level pointer to a ROM Table always takes the form of an absolute address.

From an Access Port or a Debug Port

Each *Memory Access Port* (MEM-AP) contains a BASE register that indicates one of the following:

- The base address of a ROM Table.
- The address of a debug component, which must be the only debug component that is accessible from that AP. The memory system that is accessed by this MEM-AP does not contain a ROM Table.
- No debug components are accessible from this AP, which is indicated by BASE.P having the value `0b0`.

Each *Debug Port* (DP) contains the BASEPTR0-BASEPTR1 registers that indicate one of the following:

- The base address of a ROM Table.
- The address of a debug component, which is the only debug component that is directly accessible from this DP. The debug component might be an AP that provides indirect access to more debug components.
- No debug components are accessible from this DP, which is indicated by BASEPTR0.VALID having the value `0b0`.

From processor cores

The operating system or debug monitor must be aware of the memory map of the system to find the ROM Table and debug components.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D5.6 ROM Table hierarchies

Normally, each ROM Table entry, ROMENTRY<n>, points to the memory space of a debug component. The Component and Peripheral ID Registers for that component start at offset `0xFD0` in a 4KB section of the memory space of the component. The Component class field CIDR1.CLASS, bits[7:4] of the Component ID1 Register, identifies the type of the component. This field is described in *CIDR0-CIDR3, Component Identification Registers* on page B2-38.

ROMENTRY<n> can point to another ROM Table, which is referred to as a lower-level ROM Table.

A ROM Table can include more than one entry that points to lower-level ROM Tables, and a hierarchy of ROM Tables can exist. All ROM Tables within that hierarchy must be scanned to discover all the debug components in the system.

A system that is compliant with CoreSight v3.0 or later can contain both Class `0x1` and Class `0x9` ROM Tables in a single implementation.

When identifying Class 0x1 ROM Tables, DEVARCH and DEVTYPE are treated as having a value of 0.

If more than the maximum number of ROM Table entries are required, the ROM Table must be expanded by creating a ROM Table hierarchy that contains as many ROM Table entries as necessary.

The MEM-AP BASE register must point to the top-level ROM Table in the hierarchy.

Figure D5-1 shows an example of a ROM Table hierarchy.

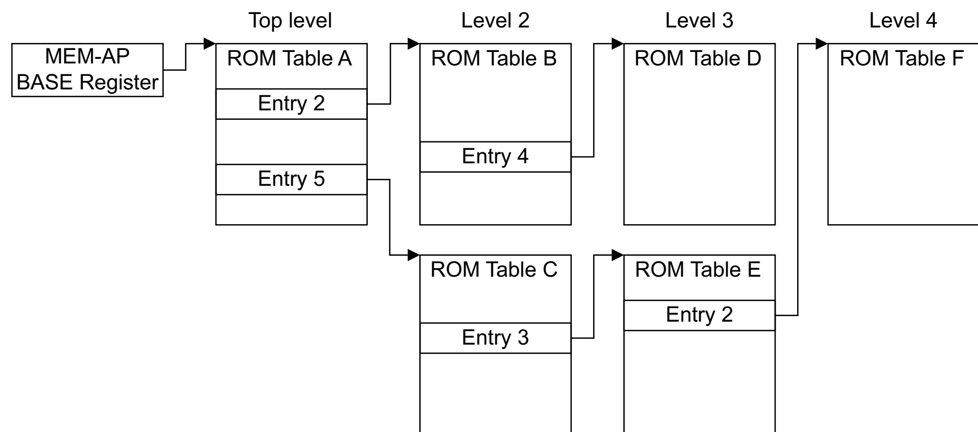


Figure D5-1 ROM Table hierarchy example

A hierarchy of ROM Tables might increase the total number of ROM Table entries in the system.

A hierarchy might be implemented for some other reason, for example, to reflect the logical organization of the debug components of the system. There might be only a few entries in each ROM Table within a hierarchy.

D5.6.1 Peripheral ID Registers in lower-level ROM Tables

The Peripheral ID value that is obtained from the Peripheral ID Registers of any ROM Table that is not a top-level ROM Table is used to identify the subsystem described by the ROM Table. It is not used to identify the SoC or platform.

D5.6.2 Component Revision Numbers

When a component is changed, the revision number that is contained in the Unique Component Identifier of that component must be changed to ensure that debug tools can differentiate the versions of the component, which usually involves changing one or more of PIDR2.REVISION and PIDR3.REVAND. For details about the Unique Component Identifier, see *The Unique Component Identifier* on page B2-33.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

In systems that are designed as multiple subsystems of components, each subsystem has a ROM Table that indicates the locations of the components in the subsystem. When changing the revision of a component in a subsystem:

- The revision of the subsystem that the component is part of may or may not change.
- The revision number of the ROM Table describing that subsystem may or may not change.

As a result, debug tools must inspect the revision of each component within a subsystem to uniquely identify the revision of those components and must not rely on the revision of the ROM Table to uniquely identify the revision of all the components within the subsystem.

For example, if the revision number of a trace macrocell that is part of a subsystem with a ROM Table which describes the layout of the subsystem changes, the revision of the ROM Table may not change, and multiple instances of the subsystem with the same revision number could exist in the ROM Table, even though the components making up the subsystems have different revision numbers for each subsystem.

D5.6.3 Prohibited ROM Table references

Every debug component within a system must appear only once in the ROM Table, or ROM Table hierarchy, that is visible to an external debugger. Figure D5-2 shows a prohibited case, where entries in ROM Tables B and C both point to ROM Table D.

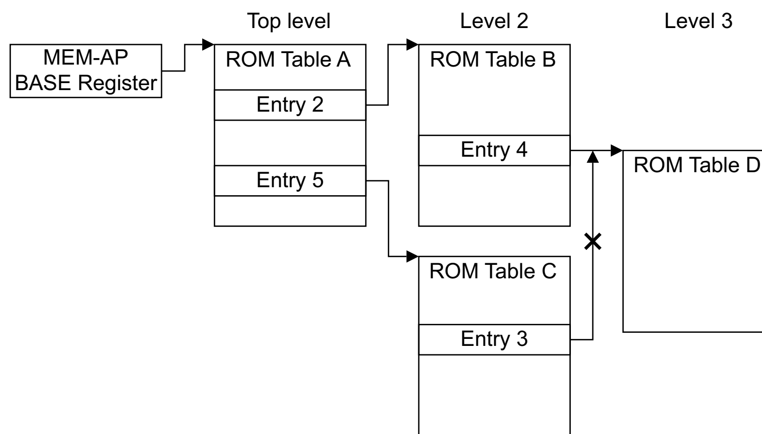


Figure D5-2 Prohibited duplicate ROM Table reference from ROM Table

Figure D5-3 shows a similar prohibited case, where entries in ROM Table A and MEM-AP 2 both point to ROM Table B.

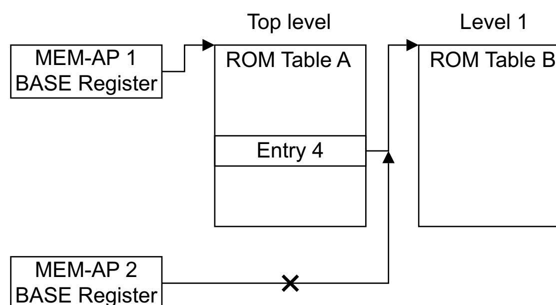


Figure D5-3 Prohibited duplicate ROM Table reference from MEM-AP

In addition, circular ROM Table references are prohibited. A ROM Table entry must not point to a ROM Table that directly or indirectly points to itself. In particular, ROM Table entries must not point back to the top-level ROM Table, as is shown in Figure D5-4 on page D5-153, where both ROM Table B and ROM Table C have prohibited links back to ROM Table A.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

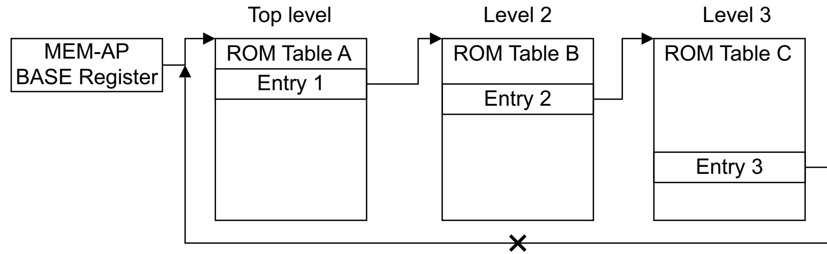


Figure D5-4 Prohibited circular ROM Table references

There is no requirement that components in separate ROM Table hierarchies must be in separate systems, which includes multiple APs in a single ADI implementation, and multiple ADI implementations. For example, if MEM-AP 1 in ADI implementation 1 points to a hierarchy of ROM Tables that includes a pointer to trace macrocell A, and MEM-AP 2 in ADI implementation 2 points to a hierarchy of ROM Tables that includes a pointer to trace sink B, then trace from trace macrocell A can be collected by trace sink B, as shown in Figure D5-5.

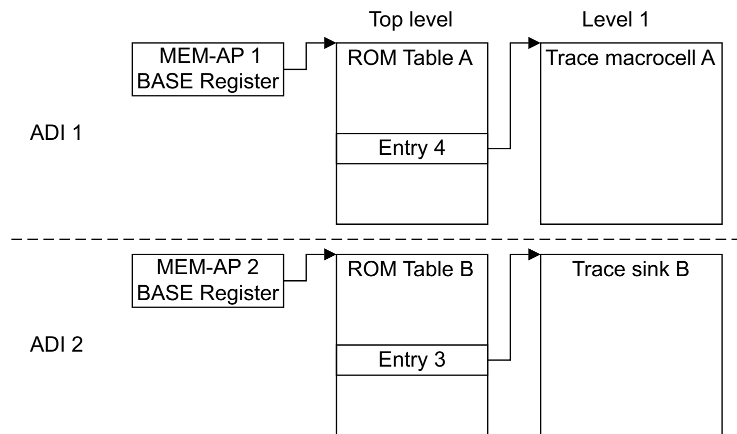


Figure D5-5 Components in separate ROM Table hierarchies

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D6
Class 0x1 ROM Tables

The chapter describes Class 0x1 ROM Tables. It includes the following sections:

- *About Class 0x1 ROM Tables* on page D6-156.
- *Class 0x1 ROM Table summary* on page D6-157.
- *Use of power domain IDs* on page D6-159.
- *Register Descriptions* on page D6-161.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.1 About Class 0x1 ROM Tables

In a Class 0x1 ROM Table implementation:

- The Component class field, CIDR1.CLASS, is 0x1, which identifies the component as a Class 0x1 ROM Table.
- The PIDR4.SIZE field must be 0, because a ROM Table must occupy a single 4KB block of memory.

Class 0x1 ROM Tables can be used alongside Class 0x9 ROM Tables, and both Class 0x1 and Class 0x9 ROM Tables might be present in systems that comply with CoreSight v3.

See also:

- For general information about ROM Tables, see Chapter D5 *About ROM Tables*.
- For information about the Component and Peripheral ID Registers, see Chapter B2 *CoreSight programmers' model*. The class configuration is described in the field description of the CIDR1.CLASS field, in section *CIDR0-CIDR3, Component Identification Registers* on page B2-38.
- For information about Class 0x9 ROM Tables, see Chapter D7 *Class 0x9 ROM Tables*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.2 Class 0x1 ROM Table summary

This section summarizes the characteristics of Class 0x1 ROM Tables.

D6.2.1 Class 0x1 ROM Table Layout

A Class 0x1 ROM Table:

- Occupies a single 4KB block of memory, and starts at offset `0x000` in this block.
- Has a series of entries, each of which is a register.
- Has a final entry that is one of the following:
  - A marker with the value 0x00000000, which signals the end of the ROM Table.
  - A regular entry at offset `0xEFC`. A ROM Table entry at this offset is always the final entry, even if it does not have the value `0x00000000`.
- Almost always has an unused area between the entry that marks the end of the ROM Table entries and the start of the reserved area at offset `0xF00`. This unused area is reserved, RES0. If a ROM Table contains 960 entries, there is no unused area.

Table D6-1 shows the Class 0x1 ROM Table registers, in order of their address offset in the 4KB block where the programmers' model resides. For detailed descriptions of each register, see *Register Descriptions* on page D6-161.

Table D6-1 ROM Table register summary

Offset	Type	Name	Description
ROM Entries, including any unused area			
`0x000` to (*N*–1)×4	RO	ROMENTRY<n>	Up to 960 ROM Table entries[a]. The end of the area that contains ROM Table entries is IMPLEMENTATION DEFINED, and depends on the number of ROM Table entries, which is denoted *N*.
N×4	RO	ROMENTRY<n> with value `0x00000000`	Marker that indicates the final entry in a ROM Table with fewer than 960 entries. The offset of this entry depends on the number of ROM Table entries, which is denoted *N*. See also *ROM Table entries that are marked not present* on page D6-158.
(N+1)×4 to `0xEFC`	-	Unused area of the ROM Table.	RES0. The offset depends on the number of ROM Table entries, which is denoted *N*. This area is not present if *N* is 960.
Reserved area			
`0xF00 - 0xFC8`	-	Reserved area of the ROM Table.	RES0.
MEMTYPE and ID registers			
`0xFCC`	RW	MEMTYPE	Memory Type Register.
`0xFD0 - 0xFDC`	RO	PIDR4-PIDR7	Peripheral Identification Registers.
`0xFE0 - 0xFEC`	RO	PIDR0-PIDR3	
`0xFF0 - 0xFFC`	RO	CIDR0-CIDR3	Component Identification Registers.

a. An implementation is unlikely to require more than the maximum number of entries in a ROM Table. However, *ROM Table hierarchies* on page D5-151 describes how larger ROM Tables can be constructed.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.2.2 ROM Table entries that are marked *not present*

The descriptions of the debug components are stored in sequential locations in the ROM Table, starting at the ROM Table base address. However, a ROM Table entry can be marked *not present* by setting the PRESENT field of the entry to a value that indicates that the entry is not present.

When scanning the ROM Table, an entry that is marked as *not present* must be skipped. Unless the entry has the value `0x00000000`, however, you must not assume that an entry that is marked *not present* represents the end of the ROM Table. For example, a ROM Table might be generated using static configuration tie-offs that indicate the presence or absence of particular devices, giving *not present* entries in the ROM Table.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3 Use of power domain IDs

If the following conditions are met, a Class 0x1 ROM Table entry, ROMENTRY<n>, has a valid power domain ID *m*:

- ROMENTRY<n>.PRESENT is `0b1`, indicating that the ROM Table entry is present.
- ROMENTRY<n>.POWERIDVALID is `0b1`, indicating that the power ID is valid.
- ROMENTRY<n>.POWERID is *m*, the power domain ID.

If any of the ROMENTRY<n> has a valid power domain ID, the ROM Table must include a valid entry that points to a power requestor that enables a debugger to request power to the power domains that are specified in the ROM Table. The power requestor must comply with the following rules:

- The power requestor must not have a valid power domain ID.
- The power requestor must be in the same power domain as the ROM Table.

For any component with a valid power domain ID, ARM recommends that, before accessing any register in a component, the debugger first accesses the power requestor to request that power is applied to the component, otherwise the component might not be powered, or it might be powered down at any time.

D6.3.1 Power domain entries

The power domain ID is specific to components identified by the Class 0x1 ROM Table, which enables hierarchies of power domains to be constructed with each level enabling access to a level below.

Figure D6-1 shows an example Class 0x1 ROM Table that indicates the locations of three components.

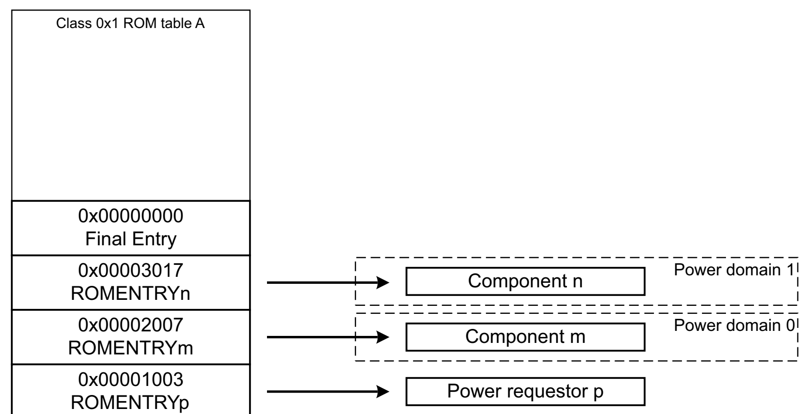


Figure D6-1 Single Class 0x1 ROM Table with power domain IDs

In Figure D6-1, the first component, *p*, has no valid power domain ID and is the power requestor. The other two components, *m* and *n*, have power domain IDs of 0 and 1 respectively. The debugger requests power for these components, by using the power requestor *p*.

Figure D6-2 on page D6-160 shows an example system with nested power domains.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

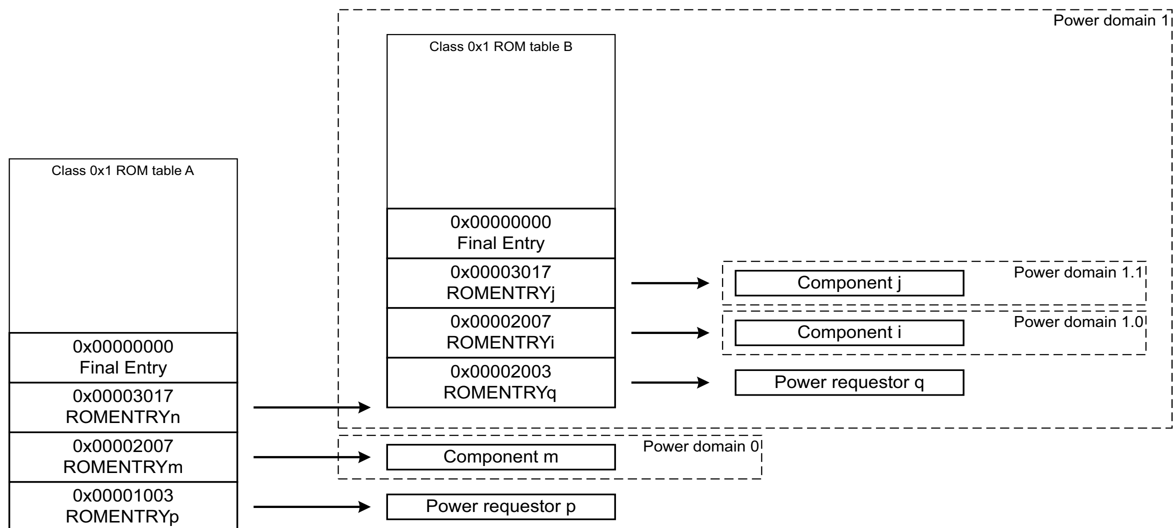


Figure D6-2 Multiple ROM Tables with nested power domain IDs

In Figure D6-2, ROM Table A is the top-level ROM Table and indicates the presence of:

- Power requestor *p*, which must be in the same power domain as the ROM.
- Component *m*, which is in power domain 0.
- ROM Table B, which is in power domain 1.

ROM Table B indicates the presence of power requestor *q* and components *i* and *j*, so power for components *i* and *j* is requested through power requestor *q*. Because power requestor *q* is in power domain 1, the power domains it controls are subdomains of power domain 1, and are labeled 1.0 and 1.1.

The power domain IDs indicated by ROM Table B are different from the power domain IDs indicated by ROM Table A and are nested within power domain 1.

D6.3.2 Algorithm to discover power domain IDs

Inspect each Class 0x1 ROM Table in the system, starting at the top-level ROM Table. For each valid Class 0x1 ROM Table entry ROMENTRY<n>:

1. If ROMENTRY<n>.POWERIDVALID is `0b1`, the power domain ID information is present. To request power for this entry, use the ROMENTRY<n>.POWERID field, bits[8:4], to program the power requestor.

2. If ROMENTRY<n>.POWERIDVALID is `0b0`, no power domain ID information is present.
   - Inspect the component and determine if it is a power requestor. Power requestor components have no power domain ID. Make a note of whether a power requestor is detected.
   - If the component is not a power requestor, the absence of a power domain ID value indicates that it is powered, and no power requests are required to power this component.

   Note

   If no power requestor is indicated in this ROM Table, all entries in this ROM Table must not have valid power domain IDs.

If there are Class 0x9 ROM Tables in the system, use the algorithm that is described in Chapter D7 *Class 0x9 ROM Tables*, section *Algorithm to discover power domain IDs* to discover power domains in them.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4 Register Descriptions

This section provides detailed descriptions of the registers in a Class 0x1 ROM Table, in alphabetical order.

D6.4.1 CIDR0-CIDR3, Component Identification Registers

This section describes the bit assignments for ROM Table components. For a full description of the CIDR, see *CIDR0-CIDR3, Component Identification Registers*.

The CIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

CIDR0-CIDR3 are accessible as follows:

Default
RO

Configurations

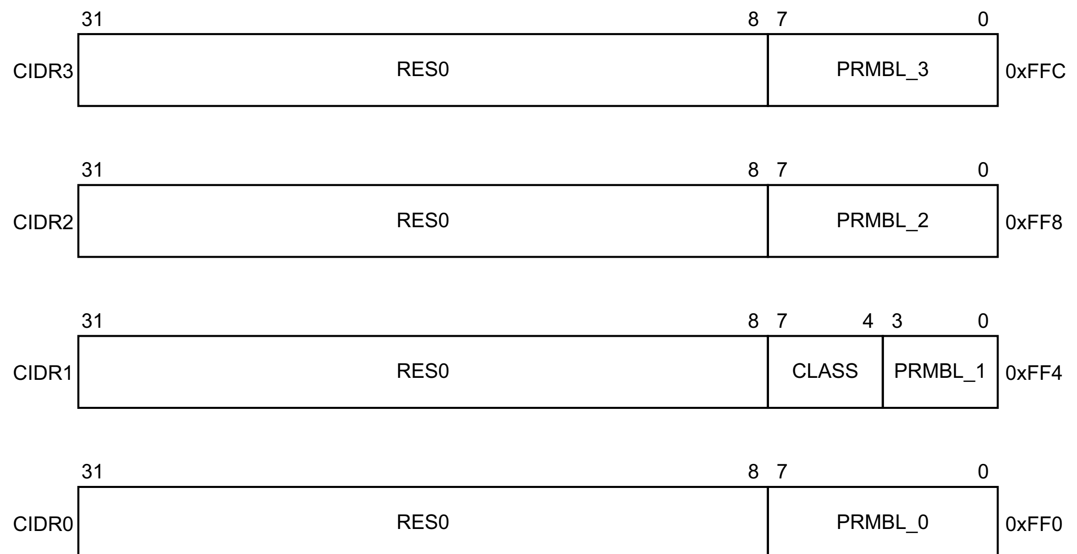
Included in all implementations.

Attributes

CIDR0-CIDR3 are four 32-bit management registers.

Field Descriptions

The CIDR bit assignments are:



CIDR3 bits[31:8]

RES0.

PRMBL_3, CIDR3 bits[7:0]

`0xB1`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

CIDR2 bits[31:8]

RES0.

PRMBL_2, CIDR2 bits[7:0]

`0x05`.

CIDR1 bits[31:8]

RES0.

CLASS, CIDR1 bits[7:4]

`0x1`.

PRMBL_1, CIDR1 bits[3:0]

`0x0`.

CIDR0 bits[31:8]

RES0.

PRMBL_0, CIDR0 bits[7:0]

`0x0D`.

Accessing the CIDR

CIDR0-CIDR3 can be accessed at the following address:

Component	Offset			
	CIDR0	CIDR1	CIDR2	CIDR3
ROM Table	`0xFF0`	`0xFF4`	`0xFF8`	`0xFFC`

D6.4.2 MEMTYPE, Memory Type Register

The MEMTYPE characteristics are:

Purpose

Identifies the type of memory present on the bus that connects to the ROM Table. In particular, it identifies whether system memory is connected to the bus.

Configurations

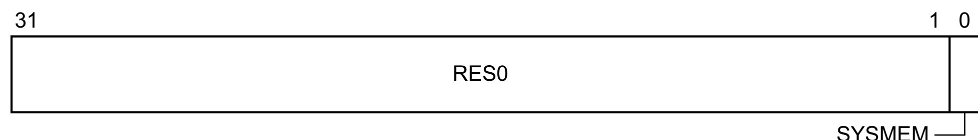
Included in all implementations.

Attributes

A 32-bit register.

Field Descriptions

The MEMTYPE register bit assignments are:



Bits[31:1]

Reserved, RES0.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

SYSMEM, bit[0]

System memory present. Indicates whether system memory is present on the bus that connects to the ROM Table. The possible values are:

`0b0` System memory not present on bus. This value indicates that the bus is a dedicated debug bus.

`0b1` System memory is also present on this bus.

MEMTYPE.SYSMEM indicates the memory accesses that the ADI can make:

When SYSMEM is `0b0`

The ROM Table indicates all the valid addresses in the memory system that the ADI is connected to, and the result of accessing any other address is UNPREDICTABLE. For more information, see *The component address* on page D5-149.

When SYSMEM is `0b1`

There might be other valid addresses in the memory system that the ADI is connected to. The result of accessing these addresses is IMPLEMENTATION DEFINED, and:

- The ADI specification does not include any mechanism that the debugger can use to discover what addresses it can access, other than the addresses that are listed in the ROM Table.
- If the ADI accesses addresses that are not in the ROM Table, there can be side effects on the system that the ADI is connected to.

Accessing MEMTYPE

MEMTYPE can be accessed at the following address:

Component	Offset
ROM Table	`0xFCC`

D6.4.3 PIDR0-PIDR7, Peripheral Identification Registers

This section describes the bit assignments for ROM Table components. For a full description of the PIDR, see *PIDR0-PIDR7, Peripheral Identification Registers*.

The PIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

PIDR0-PIDR7 are accessible as follows:

Default
RO

Configurations

Included in all implementations.

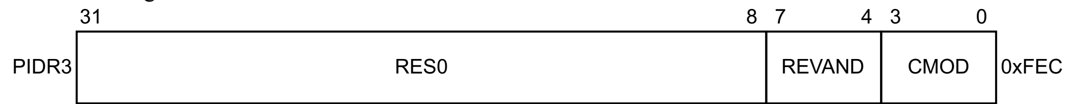
Attributes

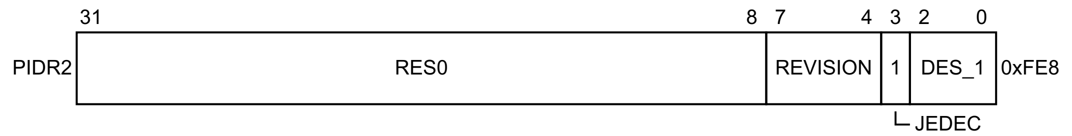
PIDR0-PIDR7 are eight 32-bit management registers.

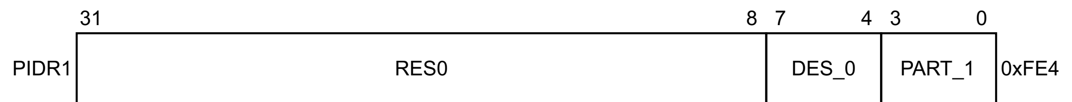
Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

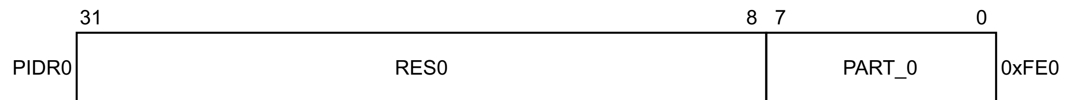
Field Descriptions

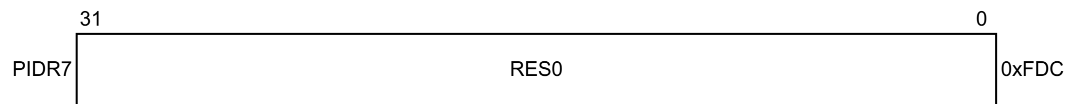
The PIDR bit assignments are:

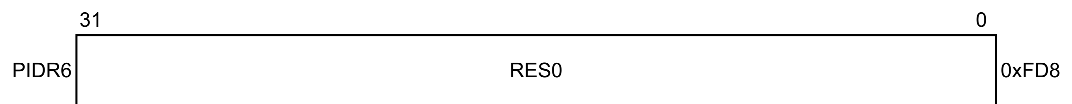


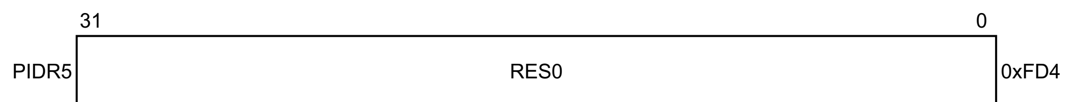


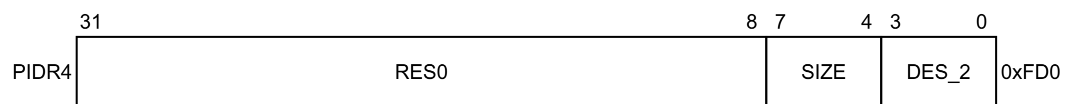












PIDR3 bits[31:8]

RES0.

REVAND, PIDR3 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

CMOD, PIDR3 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR2 bits[31:8]

RES0.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION, PIDR2 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

JEDEC, PIDR2 bits[3]

`0b1`

DES_1, PIDR2 bits[2:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR1 bits[31:8]

RES0.

DES_0, PIDR1 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PART_1, PIDR1 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR0 bits[31:8]

RES0.

PART_0, PIDR0 bits[7:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR7 bits[31:0]

RES0.

PIDR6 bits[31:0]

RES0.

PIDR5 bits[31:0]

RES0.

PIDR4 bits[31:8]

RES0.

SIZE, PIDR4 bits[7:4]

`0x0` A ROM Table occupies a single 4KB block of memory.

DES_2, PIDR4 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

Accessing the PIDR

PIDR0-PIDR7 can be accessed at the following address:

Component	Offset							
	PIDR0	PIDR1	PIDR2	PIDR3	PIDR4	PIDR5	PIDR6	PIDR7
ROM Table	0xFE0	0xFE4	0xFE8	0xFEC	0xFD0	0xFD4	0xFD8	0xFDC

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4.4 ROMENTRY<n>, Class 0x1 ROM Table entries

A Class 0x1 ROM Table contains up to 960 ROM Table entries. Each entry that is present, ROMENTRY<n>, describes a single component, component *n*.

The series of ROM Table entries start at the base address of the ROM Table. The value of a ROMENTRY<n> depends on the subsystem that is implemented.

The ROMENTRY<n> characteristics are:

Purpose

Describes a single debug component within the system.

Usage constraints

The ROMENTRY<n> are accessible as follows:

Default
RO

Configurations

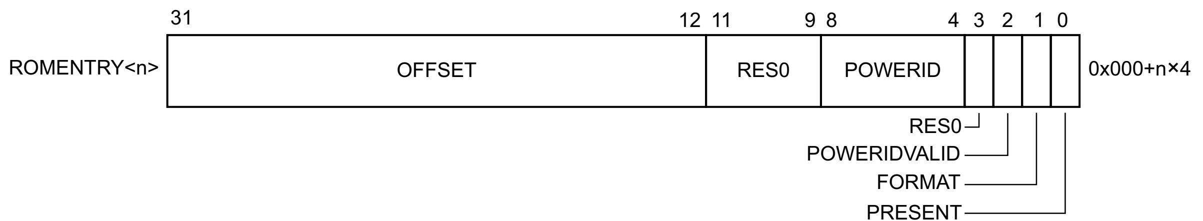
Included in all implementations.

Attributes

ROMENTRY<n> are up to 512 32-bit registers.

Field Descriptions

The ROMENTRY<n> bit assignments are:



OFFSET, bits[31:12]

The address of the component, relative to the base address of this ROM Table.

Negative values are permitted, using two's complement.

Note

If FORMAT and PRESENT both have a value that is not equal to `0b0`, OFFSET must not be zero, because a zero address offset points back to the ROM Table that contains this ROMENTRY<n>.

For more information, see *The component address* on page D5-149.

Bits[11:9]

RES0.

POWERID, bits[8:4]

The Power domain ID of the component. This field:

- Supports up to 32 Power domains using values `0x00` to `0x1F`.
- Is only valid if the POWERIDVALID field, which consists of bit[2] of the same ROM Table entry, is `0b1`, otherwise this field must be RAZ.

For more information about Power domains, see the *CoreSight Architecture Specification*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Bit[3]

RES0.

POWERIDVALID, bit[2]

Indicates if the Power domain ID field contains a Power domain ID:

`0b0` A Power domain ID is not provided.

`0b1` The POWERID field, which consists of bits[8:4] of the same ROM Table entry, provides a Power domain ID.

FORMAT, bit[1]

Indicates the format of the ROM Table. This field has the following value:

RAO 32-bit ROM Table format.

PRESENT, bit[0]

Indicates whether an entry is present at this location in the ROM Table. This field can have one of the following values:

`0b0` The ROM entry is not present.

`0b1` The ROM entry is present.

For more information, see *ROM Table entries that are marked not present* on page D6-158.

Accessing the ROMENTRY<n>

The ROMENTRY<n> for component *n* can be accessed at the following address:

Component	Offset
ROM Table	`0x000` + $n \times 4$

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D7
Class 0x9 ROM Tables

The chapter describes Class 0x9 ROM Tables.

It includes the following sections:

- *About Class 0x9 ROM Tables* on page D7-170.
- *Class 0x9 ROM Table summary* on page D7-171.
- *Use of power domain IDs* on page D7-174
- *Reset control* on page D7-180
- *Register descriptions* on page D7-182.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.1 About Class 0x9 ROM Tables

In a Class 0x9 ROM Table implementation:

- The Component class field, CIDR1.CLASS, is `0x9`, identifying the component as a CoreSight component.
- The Component Architecture Register, DEVARCH, identifies the component as a Class 0x9 ROM Table.

Class 0x9 ROM Tables can be used alongside Class 0x1 ROM Tables, and both Class 0x9 and Class 0x1 ROM Tables might be present in systems with components that comply with CoreSight v3 or later.

See also:

- For general information about ROM Tables, see Chapter D5 *About ROM Tables*.
- For information about the Component and Peripheral ID Registers, see *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40. The class configuration is described in the field description of the CIDR1.CLASS field, in section *CIDR0-CIDR3, Component Identification Registers* on page B2-38.
- For information about Class 0x1 ROM Tables, see Chapter D6 *Class 0x1 ROM Tables*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.2 Class 0x9 ROM Table summary

This section summarizes the characteristics of Class 0x9 ROM Tables.

D7.2.1 Class 0x9 ROM Table Layout

Table D7-1 shows the Class 0x9 ROM Table registers, in order of their address offset in the 4KB block where the programmers' model resides. For detailed descriptions of each register, see *Register descriptions* on page D7-182.

A Class 0x9 ROM Table:

- Occupies a single 4KB block of memory, and starts at offset `0x000` in this block.
- Has a series of entries, each of which is a register.
- Has a final entry that is one of the following:
  - — A marker that signals the end of the ROM Table, which is an entry that is all zeroes.
  - — A regular entry at offset `0x7FC`. A ROM Table entry at this offset is always the final entry, even if its PRESENT field does not have the value `0b00`.
- Almost always has an unused area between the final entry of the ROM Table and the start of the reserved area at offset `0x800`. This unused area is RES0. If a ROM Table contains the maximum number of entries, there is no unused area.

Table D7-1 ROM Table register summary

Offset	Type	Name	Description
ROM Entries, including any unused area			The number of ROM entries is denoted *N*. The number of bytes making up a ROM entry, *w*, and the maximum number of ROM entries, N_{max}, depend on the configuration of the DEVID Register: If DEVID.FORMAT == `0b0`, *w* = 4 and N_{max} = 512. If DEVID.FORMAT == `0b1`, *w* = 8 and N_{max} = 256.
`0x000` to (*N*–1)×*w*	RO	ROMENTRY<n>	Up to N ROM entries[a]. The end of the area that contains ROM entries is IMPLEMENTATION DEFINED, and depends on *N*.
N×*w*	RO	ROMENTRY<n> that is all zeroes.	Marker that indicates the final entry in a ROM Table with fewer than N entries. The offset of this entry depends on *N*. See also *ROM Table entries that are marked not present* on page D7-173.
(*N+1*)×*w* to `0x7FC`	-	Unused area of the ROM Table.	RES0. The offset depends on *N*. This area is not present if *N* is equal to the maximum number of ROM entries, N_{max}. See also *ROM Table entries that are marked not present* on page D7-173.
Reserved area			
`0x800 - 0x9FC`	-	Reserved.	RES0.
Power and reset registers			
`0xA00 - 0xA7C`	RW	DBGPCR<n>	Debug Power Control Registers.
`0xA80 - 0xAFC`	RW	DBGPSR<n>	Debug Power Status Registers.
`0xB00 -0xB7C`	RW	SYSPCR<n>	System Power Control Registers.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table D7-1 ROM Table register summary (continued)

Offset	Type	Name	Description
0xB80 - 0xBFC	RW	SYSPSR<n>	System Power Status Registers.
0xC00	RO	PRIDR0	Power Request Identification Register 0.
0xC04 - 0xC0C	-	-	RES0.
0xC10	RW	DBGRSTRR	Debug Reset Request Register.
0xC14	RO	DBGRSTAR	Debug Reset Acknowledge Register.
0xC18	RW	SYSRSTRR	System Reset Request Register.
0xC1C	RO	SYSRSTAR	System Reset Acknowledge Register.
0xC20-0xCFC	-	-	RES0.
Reserved area			
0xD00-0xEFC	-	-	RES0.
CoreSight management registers			
0xF00	RW	ITCTRL	Integration Mode Control Register.
0xF04-0xF9C	-	-	RES0.
0xFA0	RW	CLAIMSET	Claim Tag Registers.
0xFA4	RW	CLAIMCLR	
0xFA8	RO	DEVAFF0	Device Affinity Registers.
0xFAC	RO	DEVAFF1	
0xFB0	WO	LAR	Lock Access and Lock Status Registers.
0xFB4	RO	LSR	
0xFB8	RO	AUTHSTATUS	Authentication Status Register.
0xFBC	RO	DEVARCH	Device Architecture Register.
0xFC0	RO	DEVID2	Device ID Registers.
0xFC4	RO	DEVID1	
0xFC8	RO	DEVID	Device ID Register.
0xFCC	RO	DEVTYPE	Device Type Register.
0xFD0-0xFDC	RO	PIDR4-PIDR7	Peripheral Identification Registers.
0xFE0-0xFEC	RO	PIDR0-PIDR3	
0xFF0-0xFFC	RO	CIDR0-CIDR3	Component Identification Registers.

a. An implementation is unlikely to require more than the maximum number of entries in a ROM Table. However, *ROM Table hierarchies* on page D5-151 describes how larger ROM Tables can be constructed.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.2.2 ROM Table entries that are marked *not present*

The descriptions of the debug components are stored in sequential locations in the ROM Table, starting at the ROM Table base address. However, a ROM Table entry can be marked *not present* by setting the PRESENT field of the entry to a value that indicates that the entry is not present:

- If ROMENTRY<n>.PRESENT has the value `0b10`, it is *not present* and must be skipped. Do not assume that the entry represents the end of the ROM Table when scanning the ROM Table. For example, a ROM Table might be generated using static configuration tie-offs that indicate the presence or absence of particular devices, giving *not present* entries in the ROM Table.
- If ROMENTRY<n>.PRESENT has the value `0b00`, it is *not present* and indicates the end of the ROM Table.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.3 Use of power domain IDs

If the following conditions are met, a Class 0x9 ROM Table entry, ROMENTRY<n>, has a valid power domain ID m:

- ROMENTRY<n>.PRESENT is `0b1`, indicating that the ROM Table entry is present.
- ROMENTRY<n>.POWERIDVALID is `0b1`, indicating that the power ID is valid.
- ROMENTRY<n>.POWERID is *m*, the power domain ID.

The mechanism to power up power domain *m* can be one of the following:

- If DBGPCR<m>.PRESENT reads as `0b1`, the power request mechanism for DBGPCR<m>.PR is implemented, and can be used to request power for power domain *m*, as described in the field descriptions for the DBGPCR<n>.
- If DBGPCR<m>.PRESENT reads as `0b0`, the power request mechanism for power domain *m* is IMPLEMENTATION DEFINED.

ARM recommends that debug tools do not attempt accesses to any component with a valid power domain ID without first powering up the component.

D7.3.1 Power domain entries

The power domain ID is specific to components identified by the Class 0x9 ROM Table, which enables hierarchies of power domains to be constructed with each level enabling access to a level below.

Figure D7-1 shows an example Class 0x9 ROM Table that indicates the locations of two components, *m* and *n*. Components *m* and *n* are in debug power domain 0 and 1, respectively. The debugger requests power for these power domains through the power and reset registers in the programmers' model of ROM Table A.

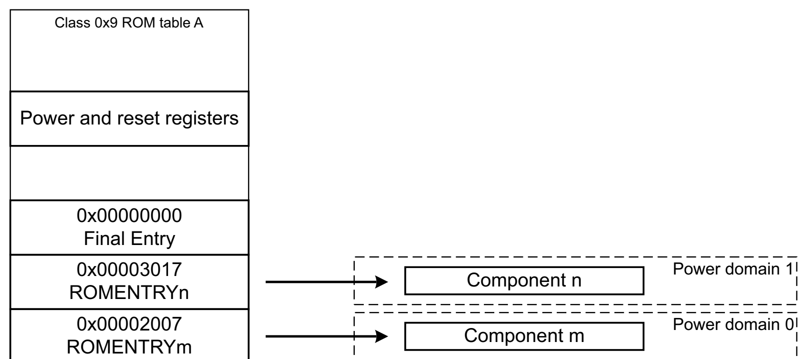


Figure D7-1 Single ROM Table with power domain IDs

Figure D7-2 on page D7-175 shows an example system with nested power domains.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

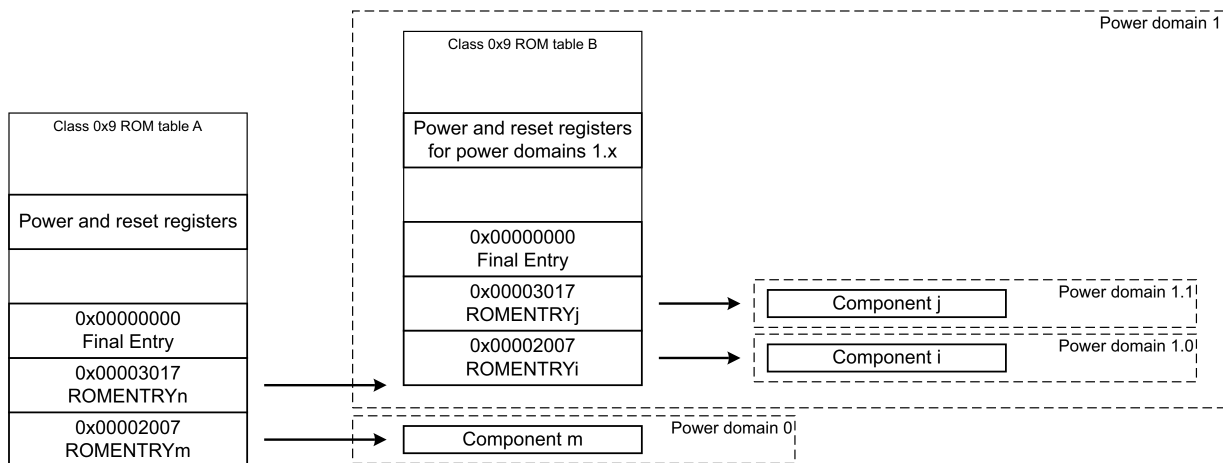


Figure D7-2 Multiple ROM Tables with nested power domain IDs

In Figure D7-2, ROM Table A is the top-level ROM Table and indicates the presence of:

- Component *m*, which is in debug power domain ID 0.
- Component *n*, which is a Class 0x9 ROM Table, ROM Table B, in debug power domain 1.

ROM Table B indicates the presence of components *i* and *j*, and power for these components is requested through the power and reset registers in the programmers' model of ROM Table B. Because ROM Table B is in power domain 1, these power domains are subdomains of power domain 1, and are labeled 1.0 and 1.1.

The power domain IDs indicated by ROM Table B are different from the power domain IDs indicated by ROM Table A and are nested within power domain 1.

D7.3.2 Algorithm to discover power domain IDs

Inspect each Class 0x9 ROM Table in the system, starting at the top-level ROM Table. For each valid Class 0x9 ROM Table entry ROMENTRY<n>:

1. If ROMENTRY<n>.POWERIDVALID is `0b1`, the power domain ID information is present. To request power for this entry, use the DBGPCR<n> in the programmers' model of the ROM Table, using the value of *n* that refers to ROMENTRY<n>.

2. If ROMENTRY<n>.POWERIDVALID is `0b0`, no power domain ID information is present, so the component is powered.

If there are Class 0x1 ROM Tables in the system, use the algorithm that is described in Chapter D6 *Class 0x1 ROM Tables*, section *Algorithm to discover power domain IDs* to discover power domains in them.

D7.3.3 Debug power requests

If access to a component is needed and the ROM entry for that component has a valid power domain ID, the debugger must request power to that component before attempting to access it.

The process for issuing a power request for a debug component is shown in Figure D7-3 on page D7-176.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

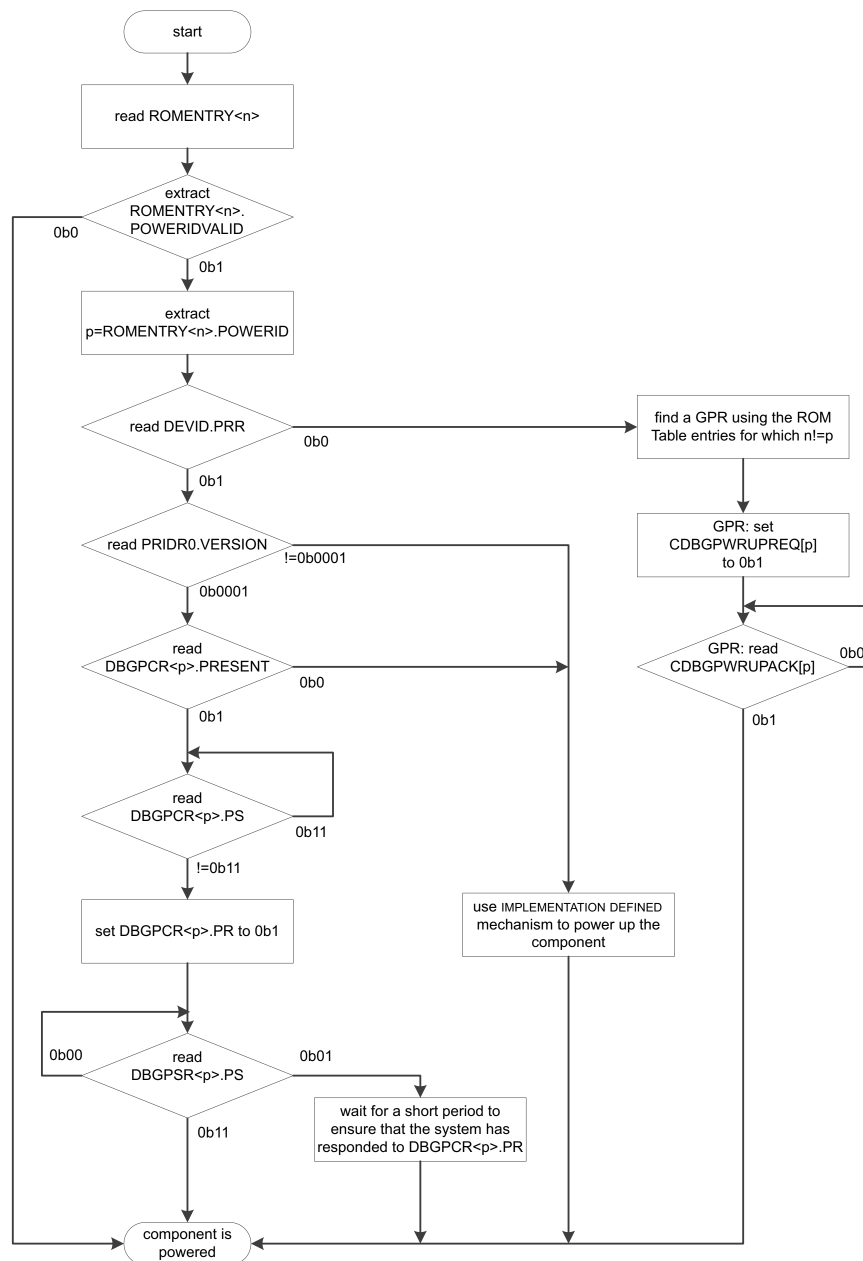


Figure D7-3 Debug power request process

A power request for a component must be removed if one of the following applies:

- The component is no longer in use. Removing the power request allows the system to consider removing power to the power domain that includes the component.
- A mechanism to preserve the component state after it is powered down is in place, for example an automated save and restore mechanism.

The process for removing a power request for a debug component is shown in Figure D7-4 on page D7-177.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

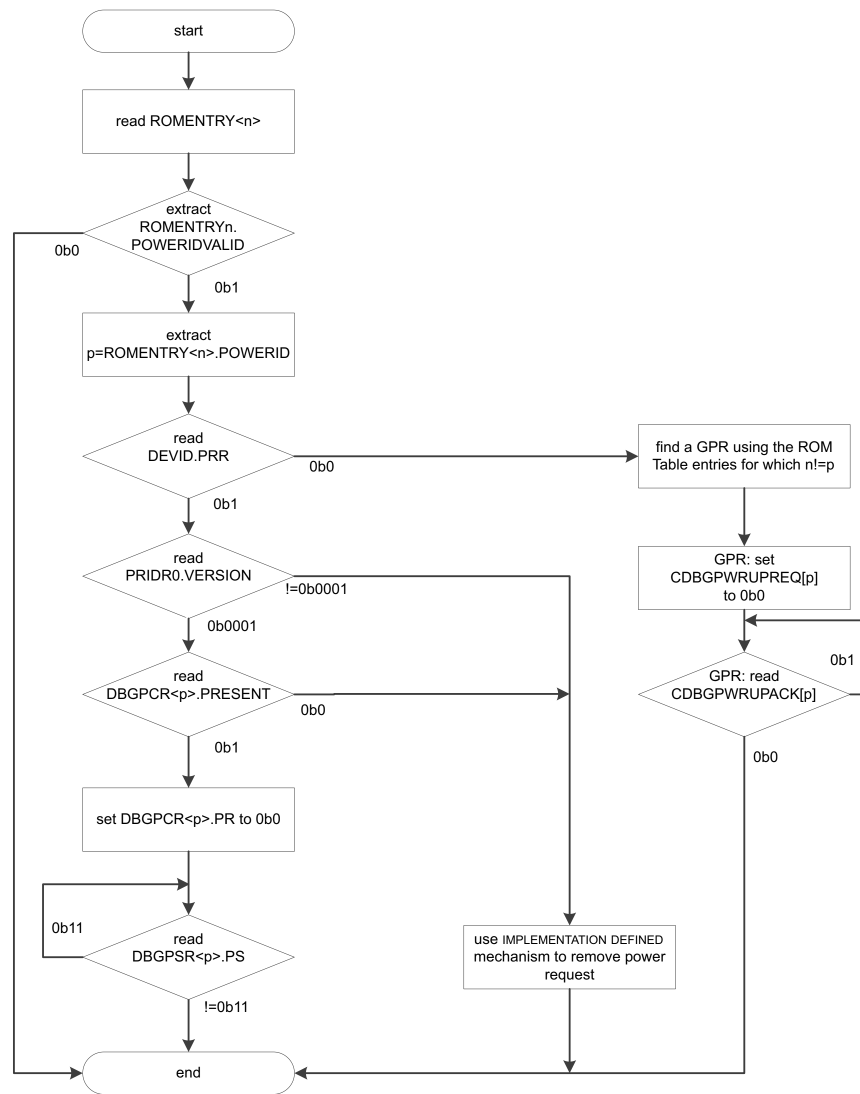


Figure D7-4 Debug power request removal process

D7.3.4 System power requests

A debugger might need to access power domains that are supported by the system, but for which no power domain IDs are defined in the ROM Table. Examples include power domains for the system interconnect and normal system memory. The power request functionality for these systems might include optional system power request controls.

It is IMPLEMENTATION DEFINED which system power domains are associated with a system power request.

The process for issuing a power request for a system component is shown in Figure D7-5 on page D7-178.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

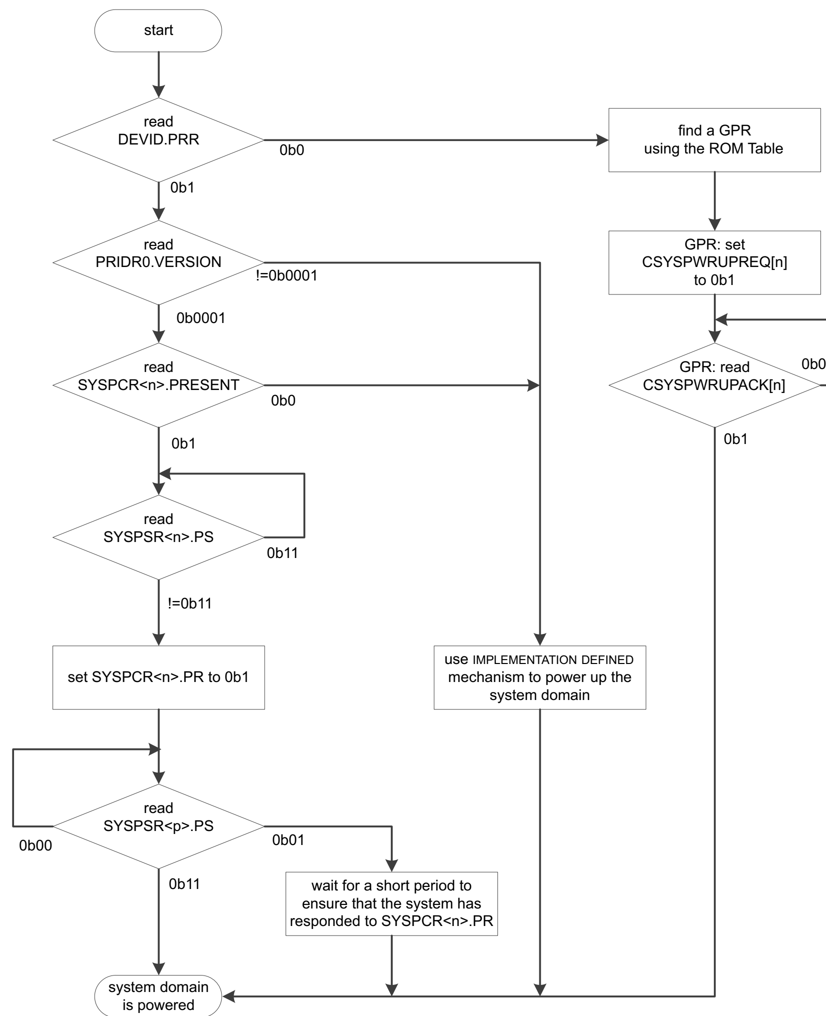


Figure D7-5 System power request process

The process for removing a power request for a system component is shown in Figure D7-6 on page D7-179.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

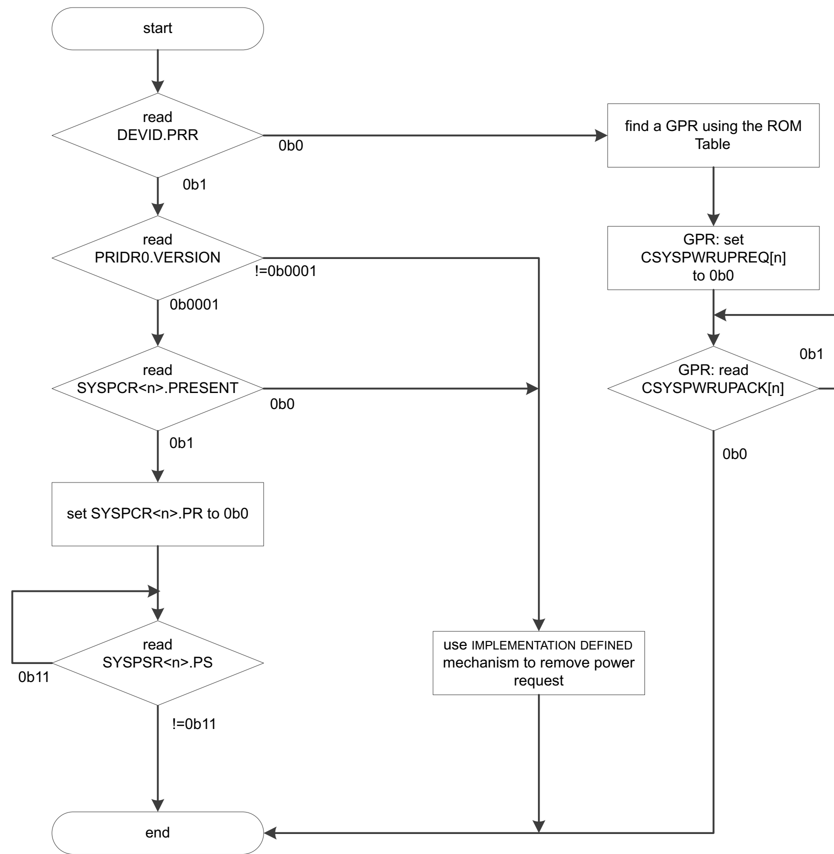


Figure D7-6 System power request removal process

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.4 Reset control

The reset control registers in the ROM Table can be used to issue reset requests:

- The debug power control registers, DBGRSTRR and DBGRSTAR, can be used to manage reset requests for up to 32 debug reset domains, as described in *Debug reset control*.
- The system power control registers, SYSRSTAR and DBGRSTAR, can be used to manage reset requests for up to 32 system reset domains, as described in *System reset control*.

For detailed descriptions of the ROM Table registers, see *Register descriptions* on page D7-182.

D7.4.1 Debug reset control

The DBGRSTR in a Class 0x9 ROM Table provides a debug logic reset request function.

The process for issuing a reset request for a debug component is shown in Figure D7-7. When performing a debug reset request, the debugger must proceed through each step of this process.

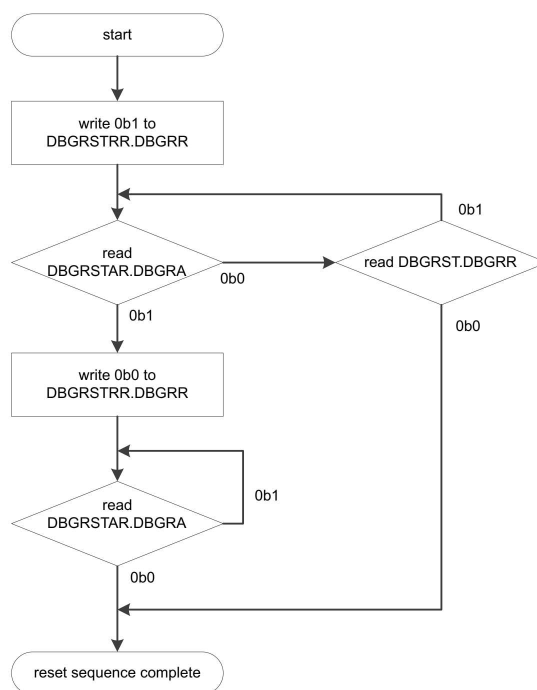


Figure D7-7 Debug reset request process

D7.4.2 System reset control

The SYSRSTRR and SYSRSTAR in a Class 0x9 ROM Table provide a system reset request function.

When SYSRSTR.SYSRR is `0b1`, the system must be held in reset. When SYSRSTR.SYSRR is `0b1`, a reset of any debug logic, for example the debug interface and logic on a processor, is permitted to happen, but the debug logic must be released from reset as soon as it has been reset. Releasing the debug logic allows a debugger to configure the debug logic while holding the system in reset, resulting in the debug logic being programmed and operating immediately after the system reset is released.

Note

If SYSRSTR.SYSRR is reset by either a system reset or a debug reset, the system is not held in reset.

This function is similar to the nSRST function provided on some physical debug ports.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The process for issuing a system reset request is shown in Figure D7-8. When performing a system reset request, the debugger must proceed through each step of this process.

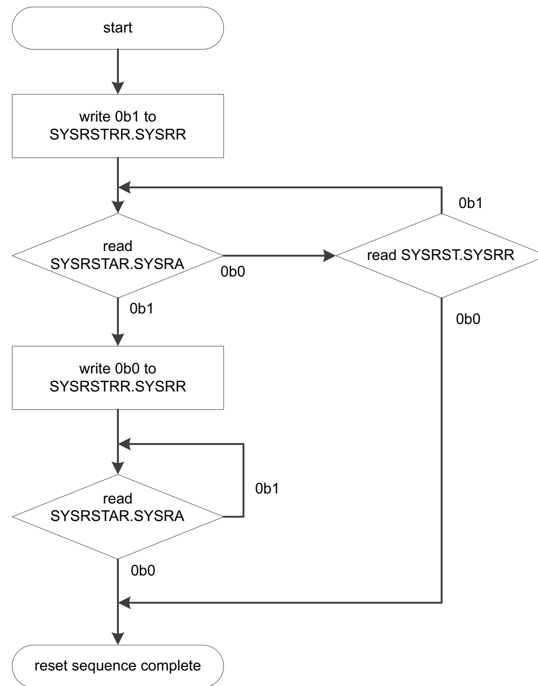


Figure D7-8 System reset request process

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5 Register descriptions

D7.5.1 AUTHSTATUS, Authentication Status Register

The AUTHSTATUS characteristics are:

Purpose

AUTHSTATUS indicates whether certain functions are enabled.

Usage constraints

Power requests using the DBGPCR<n> or SYSPCR<n> registers are considered non-invasive debug functions and are ignored when all debug functions are disabled.

If the system supports separate Secure and Non-secure functionality, reset requests using DBGRSTRR or SYSRSTRR registers are considered to be Secure invasive debug functions, and are ignored when Secure invasive debug is disabled.

If the system does not support separate Secure and Non-secure functionality, reset requests using DBGRSTRR or SYSRSTRR registers are considered to be invasive debug functions, and are ignored when invasive debug is disabled.

If the system does not support separate Secure and Non-secure functionality, then:

- AUTHSTATUS.SNID and AUTHSTATUS.NSNID are identical.
- AUTHSTATUS.SID and AUTHSTATUS.NSID are identical.

AUTHSTATUS is accessible as follows:

Default
RO

Configurations

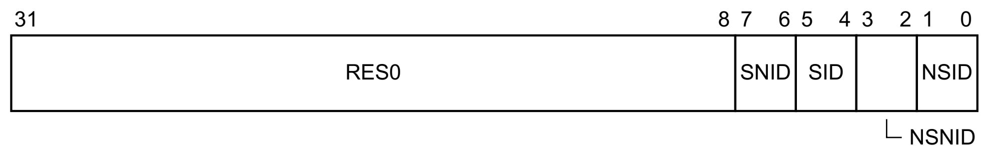
Included in all implementations.

Attributes

AUTHSTATUS is a 32-bit register.

Field Descriptions

The AUTHSTATUS bit assignments are:



Bits[31:8]

RES0.

SNID, bits[7:6]

Secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled.
(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == FALSE.

`0b11` Supported and enabled.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == TRUE.

All other values are reserved.

SID, bits[5:4]

Secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**SPIDEN** & **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**SPIDEN** & **DBGEN**) == TRUE.

All other values are reserved.

NSNID, bits[3:2]

Non-secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**NIDEN** | **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**NIDEN** | **DBGEN**) == TRUE.

All other values are reserved.

NSID, bits[1:0]

Non-secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. **DBGEN** == FALSE.

`0b11` Supported and enabled. **DBGEN** == TRUE.

All other values are reserved.

Accessing AUTHSTATUS

AUTHSTATUS can be accessed at the following address:

Component	Offset
ROM Table	`0xFB8`

D7.5.2 CIDR0-CIDR3, Component Identification Registers

This section describes the bit assignments for ROM Table components. For a full description of the CIDR, see *CIDR0-CIDR3, Component Identification Registers* on page B2-38.CIDR1

The CIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

CIDR0-CIDR3 are accessible as follows:

Default
RO

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Configurations

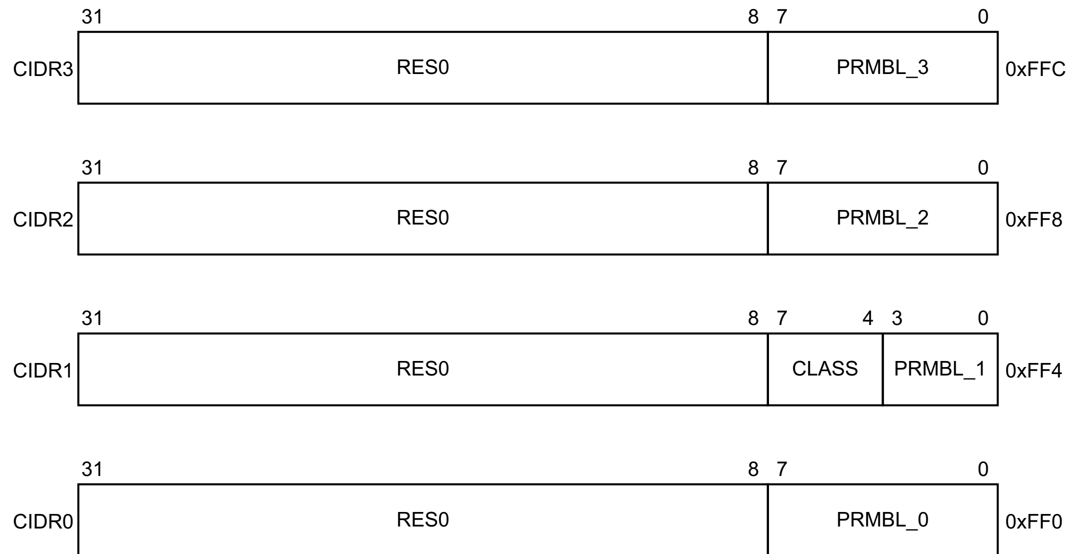
Included in all implementations.

Attributes

CIDR0-CIDR3 are four 32-bit management registers.

Field Descriptions

The CIDR bit assignments are:



CIDR3 bits[31:8]

RES0.

PRMBL_3, CIDR3 bits[7:0]

`0xB1`.

CIDR2 bits[31:8]

RES0.

PRMBL_2, CIDR2 bits[7:0]

`0x05`.

CIDR1 bits[31:8]

RES0.

CLASS, CIDR1 bits[7:4]

`0x9` CoreSight component.

PRMBL_1, CIDR1 bits[3:0]

`0x0`.

CIDR0 bits[31:8]

RES0.

PRMBL_0, CIDR0 bits[7:0]

`0x0D`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing the CIDR

CIDR0-CIDR3 can be accessed at the following address:

Component	Offset			
	CIDR0	CIDR1	CIDR2	CIDR3
ROM Table	`0xFF0`	`0xFF4`	`0xFF8`	`0xFFC`

D7.5.3 CLAIMSET and CLAIMCLR, Claim Tag Set Register and Claim Tag Clear Register

The characteristics of CLAIMSET and CLAIMCLR are:

Purpose

The Claim tag registers provide various bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For a Class 0x9 ROM Table, no claim tags are implemented.

Usage constraints

CLAIMSET and CLAIMCLR are accessible as follows:

CLAIMSET	CLAIMCLR
RW	RW

Configurations

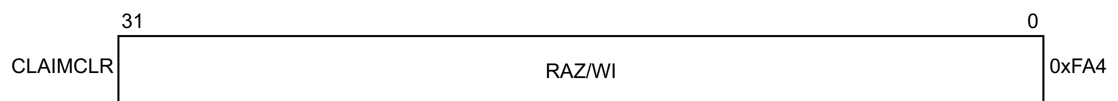
Included in all implementations.

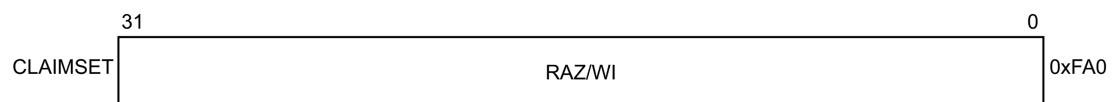
Attributes

CLAIMSET and CLAIMCLR are a set of 32-bit registers.

Field Descriptions

The CLAIMSET and CLAIMCLR bit assignments are:





CLAIMCLR, bits[31:0]

RAZ/WI, which corresponds to 0 claim tags being used.

CLAIMSET, bits[31:0]

RAZ/WI, which corresponds to 0 claim tags being used.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CLAIMSET and CLAIMCLR

CLAIMSET and CLAIMCLR can be accessed at the following addresses:

Component	Offset	
	CLAIMSET	CLAIMCLR
ROM Table	`0xFA0`	`0xFA4`

D7.5.4 DBGPCR<n>, Debug Power Control Registers

The DBGPCR<n> characteristics are:

Purpose

Power request version 0 supports up to 32 debug power domains, with separate registers for each debug domain that provide controls for power requests.

The register that controls power requests for debug power domain *n* is DBGPCR<n>. DBGPCR<n> is used for the following purposes:

- To indicate whether a power request mechanism is implemented for debug power domain *n*.
- To request power to debug power domain *n*.

Usage constraints

Debug power requests are ignored when all debug functionality is disabled. For details, see the description of AUTHSTATUS.

The DBGPCR<n> are accessible as follows:

Default
RW

Configurations

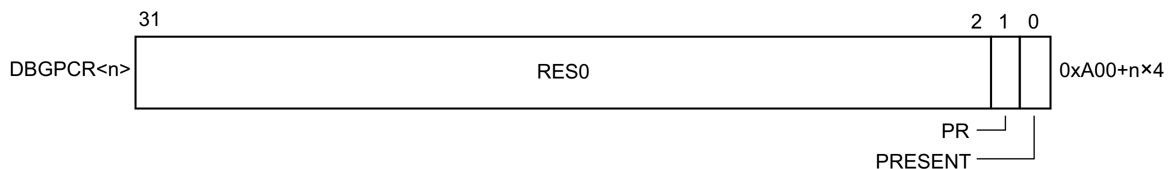
Included in all implementations.

Attributes

DBGPCR<n> are up to 32 32-bit registers.

Field Descriptions

The DBGPCR<n> bit assignments are:



Bits[31:2]

RES0.

PR, bit[1]

Power request. This field can have one of the following values:

`0b0` Power is not requested for debug power domain *n*.

`0b1` Power is requested for debug power domain *n*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

This field is reserved if DBGPCR<n>.PRESENT is `0b0`, which indicates that power requests are not implemented for debug power domain *n*.

PRESENT, bit[0]

This RO field is IMPLEMENTATION DEFINED, and indicates whether the power request for debug power domain *n* is implemented. PRESENT can have one of the following values:

`0b0` Power request for debug power domain *n* is not implemented.

`0b1` Power request for debug power domain *n* is implemented, and therefore DBGPCR<n>.PR and DBGPSR<n> are also implemented.

If, for a ROMENTRY<m> with a POWERIDVALID value of `0b1` and a POWERID value of *n*, a read of DBGPCR<n>.PRESENT returns a value of `0b0`, the mechanism to power up the debug power domain with the ID *n* is IMPLEMENTATION DEFINED.

ARM recommends that debug tools do not attempt accesses to components where ROMENTRY<m>.POWERIDVALID is `0b1` without first powering up the component using either the DBGPCR<n> register or an IMPLEMENTATION DEFINED method.

Accessing the DBGPCR<n>

The DBGPCR<n> for debug power domain *n* can be accessed at the following address:

Component	Offset
ROM Table	`0xA00` + *n*×4

D7.5.5 DBGPSR<n>, Debug Power Status Registers

The DBGPSR<n> characteristics are:

Purpose

Power request version 0 supports up to 32 debug power domains, with separate registers for each domain providing controls for power requests.

The register that indicates the current power status for debug power domain *n* is DBGPSR<n>.

Usage constraints

DBGPCR<n>.PRESENT indicates whether this register is implemented.

The DBGPSR<n> are accessible as follows:

Default
RO

Configurations

Included in all implementations.

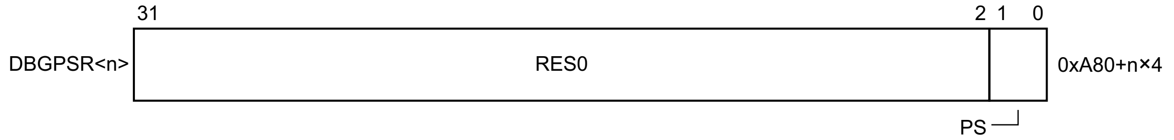
Attributes

DBGPSR<n> are up to 32 32-bit registers.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The DBGPSR<n> bit assignments are:



Bits[31:2]

RES0.

PS, bits[1:0]

Power status of debug power domain *n*. This field can have one of the following values:

`0b00` Debug power domain *n* might not be powered.

`0b01` Debug power domain *n* is powered.

`0b10` Reserved.

`0b11` Debug power domain *n* is powered and must remain powered until DBGPCR<n>.PR is set to `0b0`.

An implementation of the power request mechanism is not expected to be capable of indicating both the `0b01` and `0b11` values of DBGPSR<n>.PS.

- When DBGPSR<n>.PS reads as `0b01`, power is applied to debug power domain *n*. If DBPCRn.PR is `0b1`, ARM recommends that power to debug power domain *n* is continually maintained until DBGPCR<n>.PR is set to `0b0`, to ensure that debug-related programmed state is not lost during a debug session.
- When DBGPSR<n>.PS can read as `0b11`, the four-phase handshake with DBGPCR<n>.PR that is shown in Figure D7-9 guarantees that debug power domain *n* is powered when required:
  - At t0, DBGPCR<n>.PR is written with `0b1` to request power.
  - At t1, DBGPSR<n>.PS is set to `0b11` to indicate that the power request has been seen and power has been provided and is maintained.
  - At t2, DBGPCR<n>.PR is written with `0b0` to clear the power request.
  - At t3, DBGPSR<n>.PS is cleared to `0b00` to acknowledge that the power request has been removed.

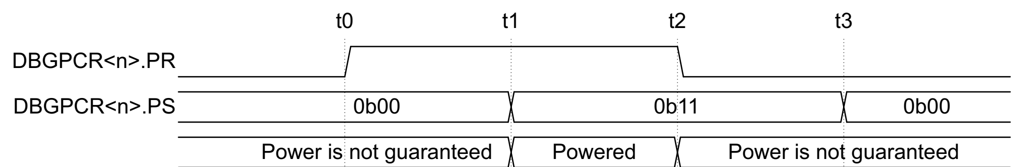


Figure D7-9 Power handshake

Between t1 and t2, power must be supplied to power domain `n`, and must not be removed.

If t1 is never reached because the system cannot power up the domain, DBGPSR<n>.PS must remain `0b00`. The debugger might choose to remove the power request by writing `0b0` to DBGPCR<n>.PR, although this practice is not recommended.

DBGPCR<n>.PR must not be used to initiate a power request when DBGPSR<n>.PS reads as `0b11`, because this value indicates that the handshake from a previous request is still completing.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing the DBGPSR<n>

The DBGPSR<n> for debug power domain *n* can be accessed at the following address:

Component	Offset
ROM Table	`0xA80` + *n*×4

D7.5.6 DBGRSTAR, Debug Reset Acknowledge Register

The DBGRSTAR characteristics are:

Purpose

Used to indicate that a debug reset has been completed, as an acknowledgement to a request that was made by using DBGRSTRR.

Usage constraints

PRIDR0.DBGRR indicates whether this register is implemented.

DBGRSTAR is accessible as follows:

Default
RO

Configurations

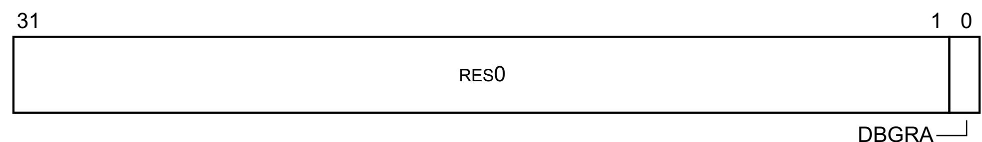
Included in all implementations.

Attributes

DBGRSTAR is a 32-bit register.

Field Descriptions

The DBGRSTAR bit assignments are:



Bits[31:1]

RES0.

DBGRA, bit[0]

Debug Reset Acknowledge:

`0b0` The debug reset request is not initiated or not completed.

`0b1` The debug reset request has been completed.

After a power-on reset, this field is set to `0b0`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGRSTAR

DBGRSTAR can be accessed at the following address:

Component	Offset
ROM Table	`0xC14`

D7.5.7 DBGRSTRR, Debug Reset Request Register

The DBGRSTRR characteristics are:

Purpose

DBGRSTRR is used to request a reset of debug functionality.

DBGRSTRR is used with DBGRSTAR in a handshake mechanism that indicates when a reset has been completed.

Usage constraints

PRIDR0.DBGRR indicates whether this register is implemented.

Debug reset requests are ignored when invasive debug functionality is disabled. For details, see the description of AUTHSTATUS.

DBGRSTRR is accessible as follows:

Default
RW

Configurations

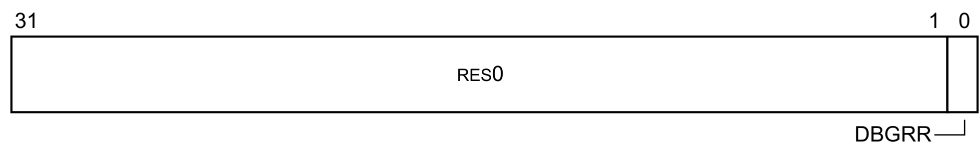
Included in all implementations.

Attributes

DBGRSTRR is a 32-bit register.

Field Descriptions

The DBGRSTRR bit assignments are:



Bits[31:1]

RES0.

DBGRR, bit[0]

Debug Reset Request:

`0b0` A debug reset is not requested.

`0b1` A debug reset is requested. The request remains asserted until this field is explicitly overwritten with the value `0b0`.

After a power-on reset, this field is set to `0b0`.

Whether a debug reset request also resets DBGRSTRR is IMPLEMENTATION DEFINED. If DBGRSTRR is reset by a debug reset, DBGRR is reset to `0b0`, and the reset process is complete.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Setting DBGRSTRR.DBGRR to `0b0` when DBGRSTAR.DBGRA is not `0b1` results in UNPREDICTABLE behavior, and a debug reset might or might not occur.

Accessing DBGRSTRR

DBGRSTRR can be accessed at the following address:

Component	Offset
ROM Table	`0xC10`

D7.5.8 DEVAFF0-DEVAFF1, Device Affinity Registers

The DEVAFF0-DEVAFF1 characteristics are:

Purpose

Enables a debugger to determine whether two components have an affinity with each other.

Usage constraints

DEVAFF0-DEVAFF1 are accessible as follows:

Default
RO

Configurations

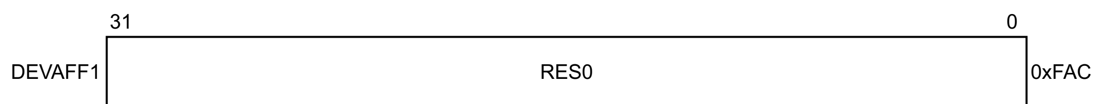
Included in all implementations.

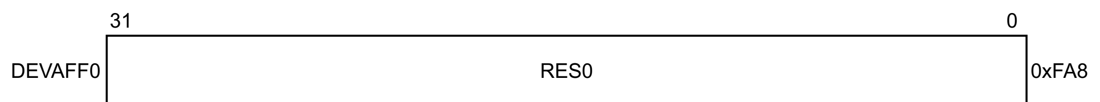
Attributes

DEVAFF0-DEVAFF1 are two 32-bit registers.

Field Descriptions

The DEVAFF0-DEVAFF1 bit assignments are:





DEVAFF0, bits[31:0]
DEVAFF1, bits[31:0]

RES0.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DEVAFF0-DEVAFF1

DEVAFF0-DEVAFF1 can be accessed at the following addresses:

Component	Offset	
	DEVAFF0	DEVAFF1
ROM Table	`0xFA8`	`0xFAC`

D7.5.9 DEVARCH, Device Architecture Register

The DEVARCH characteristics are:

Purpose

Identifies the architect and architecture of a CoreSight component.

Usage constraints

DEVARCH is accessible as follows:

Default
RO

Configurations

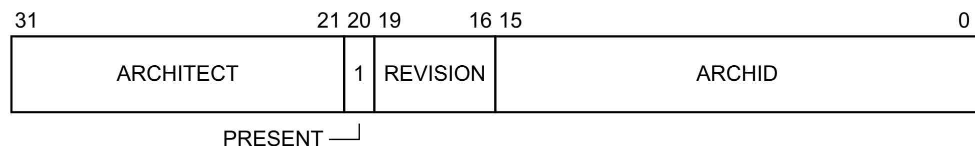
Included in all implementations.

Attributes

DEVARCH is a 32-bit register.

Field Descriptions

The DEVARCH bit assignments are:



ARCHITECT, bits[31:21]

`0x23B` ARM.

PRESENT, bit[20]

`0b1` Present.

REVISION, bits[19:16]

`0x0` Revision 0.

ARCHID, bits[15:0]

`0x0AF7` ROM Table v0. If this value of ARCHID is found, the debug tool must inspect DEVTYPE and DEVID to determine further information about the ROM Table.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DEVARCH

DEVARCH can be accessed at the following address:

Component	Offset
ROM Table	`0xFBC`

D7.5.10 DEVID, Device Configuration Register

The DEVID characteristics are:

Purpose

Indicates the capabilities of the component.

Usage constraints

DEVID is accessible as follows:

Default
RO

Configurations

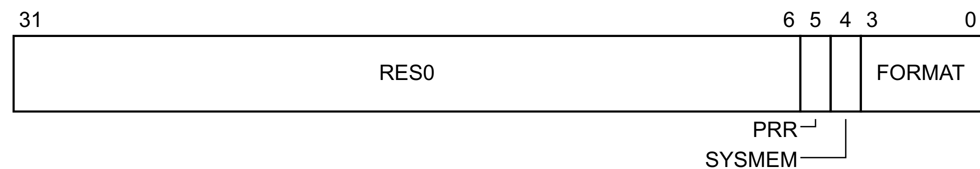
Included in all implementations.

Attributes

DEVID is a 32-bit register.

Field Descriptions

The DEVID bit assignments are:



Bits[31:6]

RES0.

PRR, bit[5]

Power Request functionality included. This field can have one of the following values:

`0b0` Power Request functionality not included.

If any ROM Table entries contain power domain IDs, a GPR must be present, and pointed to by the ROM Table. The GPR provides functionality to control the power domains.

PRIDR0 is not implemented.

`0b1` Power Request functionality included.

If any ROM Table entries contain power domain IDs, they are controlled by the Power Request functionality in the ROM Table.

PRIDR0 is implemented.

See also *PRIDR0, Power Request ID Register 0* on page D7-201.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

SYSMEM, bit[4]

System memory present. Indicates whether system memory is present on the bus that connects to the ROM Table. The possible values are:

`0b0` System memory is not present on the bus. This value indicates that the bus is a dedicated debug bus.

`0b1` System memory is also present on this.

SYSMEM indicates the memory accesses that the ADIcan make:

When SYSMEM is `0b0`

The ROM Table indicates all the valid addresses in the memory system that the ADI is connected to, and the result of accessing any other address is UNPREDICTABLE. For more information, see *The component address* on page D5-149.

When SYSMEM is `0b1`

There might be other valid addresses in the memory system that the ADI is connected to. The result of accessing these addresses is IMPLEMENTATION DEFINED, and:

- The CoreSight v3 specification does not include any mechanism that the debugger can use to discover what addresses it can access, other than the addresses that are listed in the ROM Table.
- If the ADI accesses addresses that are not in the ROM Table, there can be side effects on the system that the ADI is connected to.

FORMAT, bits[3:0]

ROM format. This field can have one of the following values:

`0x0` 32-bit format 0.

`0x1` 64-bit format 1.

Values `0x2-0xF` are reserved.

Accessing DEVID

DEVID can be accessed at the following address:

Component	Offset
ROM Table	`0xFCC`

D7.5.11 DEVID1-DEVID2, Device Configuration Registers

The DEVID1-DEVID2 characteristics are:

Purpose

Indicates the capabilities of the component.

Usage constraints

DEVID1-DEVID2 are accessible as follows:

Default
RO

Configurations

Included in all implementations.

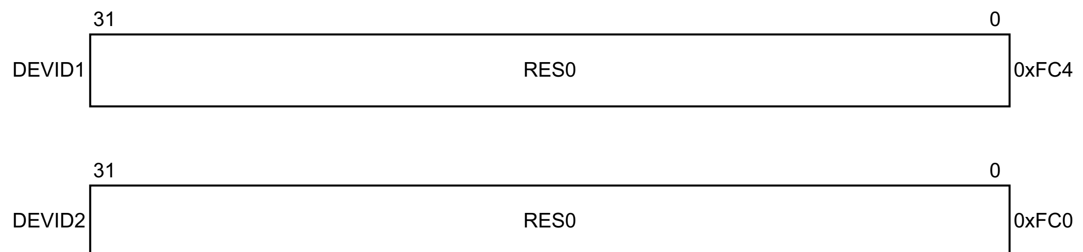
Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

DEVID1-DEVID2 are two 32-bit registers.

Field Descriptions

The DEVID1-DEVID2 bit assignments are:



DEVID1, bits[31:0]
DEVID2, bits[31:0]

RES0.

Accessing DEVID1-DEVID2

DEVID1-DEVID2 can be accessed at the following addresses:

Component	Offset	
	DEVID1	DEVID2
ROM Table	`0xFC4`	`0xFC0`

D7.5.12 DEVTYPE, Device Type Register

The DEVTYPE characteristics are:

Purpose

A debugger can use DEVTYPE to obtain information about a component that has an unrecognized part number.

Usage constraints

DEVTYPE is accessible as follows:

Default
RO

Configurations

Included in all implementations.

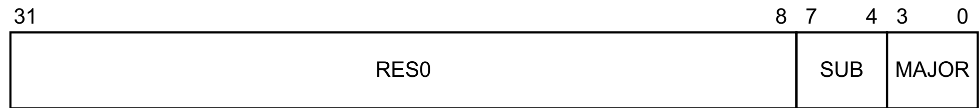
Attributes

DEVTYPE is a 32-bit register.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The DEVTYPE bit assignments are:



Bits[31:8]

RES0.

SUB, bits[7:4]

`0x0` Other, undefined.

MAJOR, bits[3:0]

`0x0` Miscellaneous.

Accessing DEVTYPE

DEVTYPE can be accessed at the following address:

Component	Offset
ROM Table	`0xFCC`

D7.5.13 ITCTRL, Integration Mode Control Register

The ITCTRL characteristics are:

Purpose

A component can use ITCTRL to dynamically switch between functional mode and integration mode.

For a Class 0x9 ROM Table, this mechanism is not implemented.

Usage constraints

ITCTRL is accessible as follows:

Default
RW

Configurations

Included in all implementations.

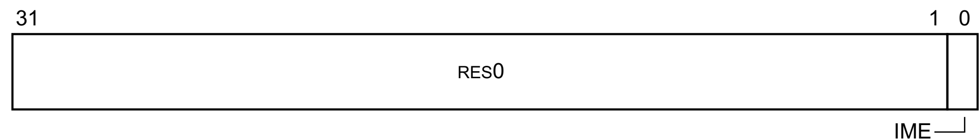
Attributes

ITCTRL is a 32-bit register.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The ITCTRL bit assignments are:



Bits[31:1]

RES0.

IME, bit[0]

RAZ/WI, which indicates that no integration functionality is implemented.

Accessing ITCTRL

ITCTRL can be accessed at the following address:

Component	Offset
ROM Table	`0xF00`

D7.5.14 LAR and LSR, Software Lock Access Register and Software Lock Status Register

The LAR and LSR characteristics are:

Purpose

The Software lock mechanism prevents accidental access to the registers of CoreSight components.

For a Class 0x9 ROM Table, the lock mechanism is not implemented.

Usage constraints

LAR and LSR are accessible as follows:

LAR	LSR
WO	RO

Configurations

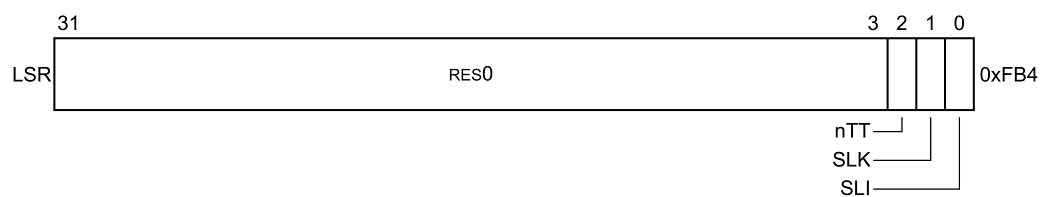
Included in all implementations.

Attributes

LAR and LSR are two 32-bit registers.

Field Descriptions

The LAR and LSR bit assignments are:



Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

LSR, bits[31:3]

RAZ.

nTT, LSR bit[2]

RAZ.

SLK, LSR bit[1]

RAZ.

SLI, LSR bit[0]

RAZ.

KEY, LAR bits[31:0]

WI.

Accessing LSR and LAR

LAR and LSR can be accessed at the following addresses:

Component	Offset	
	LAR	LSR
ROM Table	`0xFB0`	`0xFB4`

D7.5.15 PIDR0-PIDR7, Peripheral Identification Registers

This section describes the bit assignments for ROM Table components. For a full description of the PIDR, see *PIDR0-PIDR7, Peripheral Identification Registers*.

The PIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

PIDR0-PIDR7 are accessible as follows:

Default
RO

Configurations

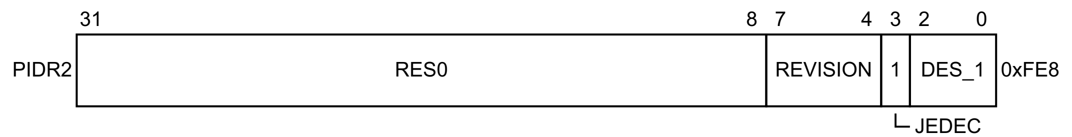
Included in all implementations.

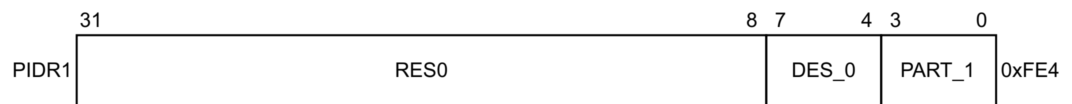
Attributes PIDR0-PIDR7 are eight 32-bit management registers.

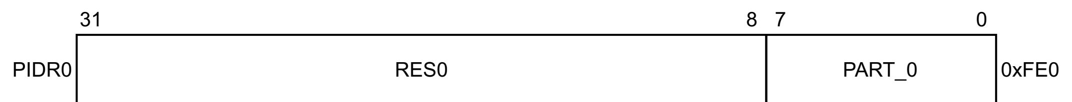
Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

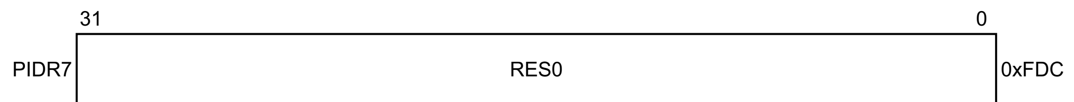
Field Descriptions

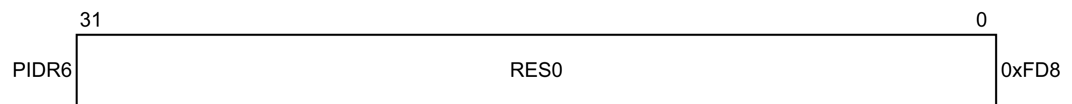
The PIDR bit assignments are:

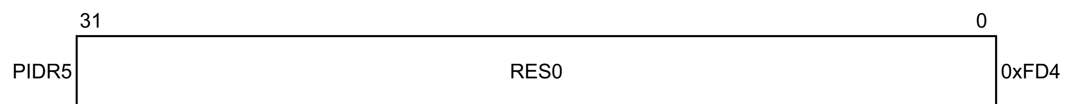


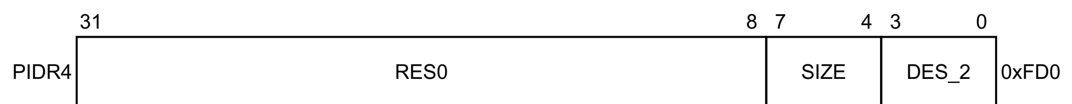












PIDR3 bits[31:8]

RES0.

REVAND, PIDR3 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

CMOD, PIDR3 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR2 bits[31:8]

RES0.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION, PIDR2 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

JEDEC, PIDR2 bits[3]

`0b1` A JEDEC value is used.

DES_1, PIDR2 bits[2:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR1 bits[31:8]

RES0.

DES_0, PIDR1 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PART_1, PIDR1 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR0 bits[31:8]

RES0.

PART_0, PIDR0 bits[7:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR7 bits[31:0]

RES0.

PIDR6 bits[31:0]

RES0.

PIDR5 bits[31:0]

RES0.

PIDR4 bits[31:8]

RES0.

SIZE, PIDR4 bits[7:4]

`0x0` A ROM Table occupies a single 4KB block of memory.

DES_2, PIDR4 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

Accessing the PIDR

PIDR0-PIDR7 can be accessed at the following address:

Component	Offset							
	PIDR0	PIDR1	PIDR2	PIDR3	PIDR4	PIDR5	PIDR6	PIDR7
ROM Table	0xFE0	0xFE4	0xFE8	0xFEC	0xFD0	0xFD4	0xFD8	0xFDC

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5.16 PRIDR0, Power Request ID Register 0

The PRIDR0 characteristics are:

Purpose

Indicates the features of the power request functionality.

Usage constraints

PRIDR0 is accessible as follows:

Default
RO

Configurations

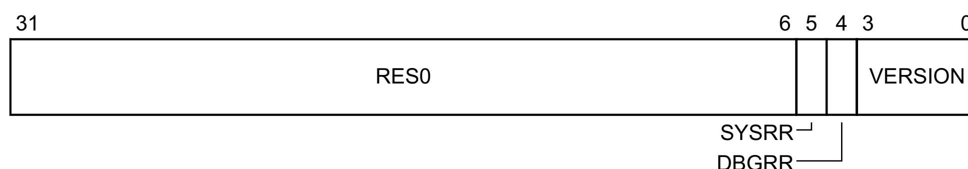
Included in all implementations.

Attributes

PRIDR0 is a 32-bit register.

Field Descriptions

The PRIDR0 bit assignments are:



Bits[31:6]

RES0.

SYSRR, bit[5]

Indicates whether the system reset request functionality is present. This field can have one of the following values:

`0b0` The system reset request functionality is not implemented.

`0b1` The system reset request functionality, and SYSRSTRR and SYSRSTAR, which provide status information for system resets, are implemented.

DBGRR, bit[4]

Indicates whether the debug reset request functionality is present. This field can have one of the following values:

`0b0` The debug reset request functionality is not implemented.

`0b1` The system reset request functionality, and DBGRSTRR and DBGRSTAR, which provide status information for debug resets, are implemented.

VERSION, bits[3:0]

Indicates the version of the reset request functionality. This field can have one of the following values:

`0b0000` The power request functionality is not implemented.

`0b0001` The power request functionality version 0, and the DBGPCR<n>, DBGPSR<n>, SYSPCR<n>, and SYSPSR<n>, which provide controls for power requests, are implemented.

Values `0b0010`-`0b1111` are reserved.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PRIDR0

PRIDR0 can be accessed at the following address:

Component	Offset
ROM Table	`0xC00`

D7.5.17 ROMENTRY<n>, Class 0x9 ROM Table entries

The ROMENTRY<n> characteristics are:

Purpose

A Class 0x9 ROM Table contains up to 512 ROM Table entries. Each entry that is present, ROMENTRY<n>, provides the address offset of the address space of one CoreSight component, component *n*, along with information about its power domain.

The series of ROM Table entries starts at the base address of the Class 0x9 ROM Table:

- The first entry, entry 0, has offset `0x000`.
- ROMENTRY<n> has the offset `0x000` + n×4, where $0 \leq n \leq 511$.
- If the number of components, *N*, is lower than the maximum supported number, 512, the offsets of the ROM Table entries are in the following range:
  - — ROM Table entries representing components have offsets from `0x000` to (N–1)×4.
  - — The ROMENTRY<n> at offset N×4, which has a PRESENT field with the value `0b00`, indicates the end of the ROM Table.
- If the number of components is equal to the maximum supported number, the ROM Table entries have offsets from `0x000` to `0x7FC`. If a ROM Table entry is present at offset `0x7FC`, its PRESENT field must have a value of either `0b00` or `0b11`, and it must be interpreted as the final entry of the ROM Table, even if its PRESENT field has the value `0b11`.

A component that requires differentiation between external and internal accesses may provide two views, one for internal and one for external accesses. For these components, ARM strongly recommends that ROM Tables provide a pointer only to the external view. See also *ROM Table hierarchies* on page D5-151.

Usage constraints

A ROM Table does not have to use power domain IDs. If none of the ROM Table entries provides a power domain ID, all the components that pointed to by the ROM Table are in the same power domain as the ROM Table. For more information, see *Use of power domain IDs* on page D7-174.

ARM recommends the power requestor that is described in Appendix E1 *Power Requestor*.

The ROMENTRY<n> are accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

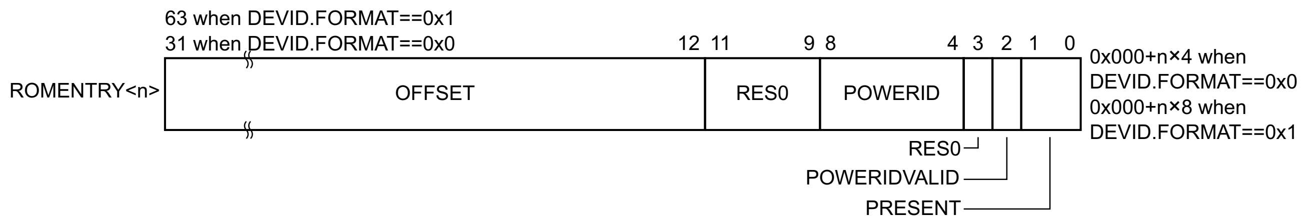
The ROMENTRY<n> attributes depend on the configuration of the CoreSight DEVID:

- If DEVID.FORMAT has the value `0x0`, the ROMENTRY<n> are 512 32-bit registers.
- If DEVID.FORMAT has the value `0x1`, the ROMENTRY<n> are 256 64-bit registers.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The ROMENTRY<n> bit assignments are:



OFFSET, bits[31:12] when DEVID.FORMAT has the value 0x0
OFFSET, bits[63:12] when DEVID.FORMAT has the value 0x1

The component address, relative to the base address of this ROM Table. The component address is calculated using the following equation:

Component Address = ROM Table Base Address + (OFFSET << 12).

If a component occupies more than a single 4KB block, OFFSET points to the 4KB block which contains the Peripheral ID and Component ID registers for the component.

Negative values of OFFSET are permitted, using two's complement.

Note

If bits[1:0] are not `0b00`, the OFFSET field of a ROM Table entry must not be zero, because a zero address offset points back to this ROM Table.

See also *The component address* on page D5-149.

Bits[11:9]

RES0.

POWERID, bits[8:4]

The power domain ID of the component. This field:

- Supports up to 32 power domains using values `0x00` to `0x1F`.
- Is only valid if the POWERIDVALID field, which consists of bit[2] of the same ROMENTRY<n>, is `0b1`, otherwise this field must be RES0.

Bit[3]

RES0.

POWERIDVALID, bit[2]

Indicates if the Power domain ID field contains a Power domain ID:

`0b0` A power domain ID is not provided.

`0b1` The POWERID field, which consists of bits[8:4] of the same ROMENTRY<n>, provides a power domain ID.

PRESENT, bits[1:0]

Indicates whether an entry is present at this location in the ROM Table. This field can have one of the following values:

`0b00` The ROM entry is not present, and this ROMENTRY<n> is the final entry in the ROM Table. If PRESENT has this value, all other fields in this ROMENTRY<n> must be zero.

`0b01` Reserved.

`0b10` The ROM entry is not present, and this ROMENTRY<n> is not the final entry in a ROM Table with fewer than the maximum number of entries. If PRESENT has this value, all other fields in this entry are UNKNOWN.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`0b11` The ROM Entry is present.

If the number of components is equal to the maximum number of ROM Table entries, the last ROM Table entry is not required to have a PRESENT field with the value `0b00`.

Accessing the ROMENTRY<n>

The ROMENTRY<n> for component *n* can be accessed at the following address:

Component	Offset	
	DEVID.FORMAT == `0b0`	DEVID.FORMAT == `0b1`
ROM Table	`0x000` + *n*	`0x000` + *n*×8

D7.5.18 SYSPCR<n>, Debug Power Control Registers

The SYSPCR<n> characteristics are:

Purpose

Power request version 0 supports up to 32 system power domains, with separate registers for each domain providing controls for power requests.

The register that controls power requests for system power domain *n* is SYSPCR<n>. SYSPCR<n> is used for the following purposes:

- To indicate whether a power request mechanism is implemented for system power domain *n*.
- To request power to system power domain *n*.

Usage constraints

System power requests are ignored when all debug functionality is disabled. For details, see the description of AUTHSTATUS.

The SYSPCR<n> are accessible as follows:

Default
RW

Configurations

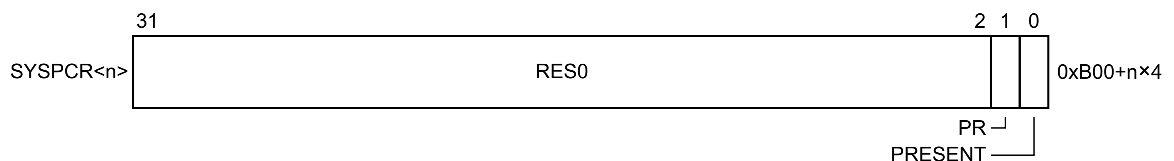
Included in all implementations.

Attributes

SYSPCR<n> are up to 32 32-bit registers.

Field Descriptions

The SYSPCR<n> bit assignments are:



Bits[31:2]

RES0.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

PR, bit[1]

Power request. This field can have one of the following values:

`0b0` Power is not requested for system power domain *n*.

`0b1` Power is requested for system power domain *n*.

This field is reserved if power requests are not implemented for system power domain *n*, which is the case if SYSPCR<n>.PRESENT has the value `0b0`.

PRESENT, bit[0]

This RO field indicates whether the power request for system power domain n is implemented. PRESENT can have one of the following values:

`0b0` Power request for system power domain *n* is not implemented.

`0b1` Power request for system power domain *n* is implemented, and therefore SYSPCR<n>.PR and SYSPSR<n> are also implemented.

Accessing the SYSPCR<n>

The SYSPCR<n> for system power domain *n* can be accessed at the following address:

Component	Offset
ROM Table	`0xB00` + *n*×4

D7.5.19 SYSPSR<n>, System Power Status Registers

The SYSPSR<n> characteristics are:

Purpose

Power request version 0 supports up to 32 system power domains, with separate registers for each domain providing controls for power requests.

The register that indicates the current power status for system power domain *n* is SYSPSR<n>.

Usage constraints

SYSPCR<n>.PRESENT indicates whether this register is implemented.

The SYSPSR<n> are accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

SYSPSR<n> are up to 32 32-bit registers.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The SYSPSR<n> bit assignments are:

Bits[31:2]

RES0.

PS, bits[1:0]

Power status of system power domain *n*. This field can have one of the following values:

`0b00` System power domain *n* might not be powered.

`0b01` System power domain *n* is powered.

`0b10` Reserved.

`0b11` System power domain *n* is powered and must remain powered until SYSPCR<n>.PR is set to `0b0`.

An implementation of the power request mechanism is not expected to be capable of indicating both the `0b01` and `0b11` values of SYSPSR<n>.PS.

- When SYSPSR<n>.PS reads as `0b01`, power is applied to system power domain *n*. If SYSPCR<n>.PR is `0b1`, ARM recommends that power to system power domain *n* is continually maintained until SYSPCR<n>.PR is set to `0b0`, to ensure that system-related programmed state is not lost during a debug session.
- When SYSPSR<n>.PS can read as `0b11`, the four-phase handshake with SYSPCR<n>.PR that is shown in Figure D7-10 guarantees that system power domain *n* is powered when required:
  - — At t0, SYSPCR<n>.PR is written with `0b1` to request power.
  - — At t1, SYSPSR<n>.PS is set to `0b11` to indicate that the power request has been seen and power has been provided and is maintained.
  - — At t2, SYSPCR<n>.PR is written with `0b0` to clear the power request.
  - — At t3, SYSPSR<n>.PS is cleared to `0b00` to acknowledge that the power request has been removed.

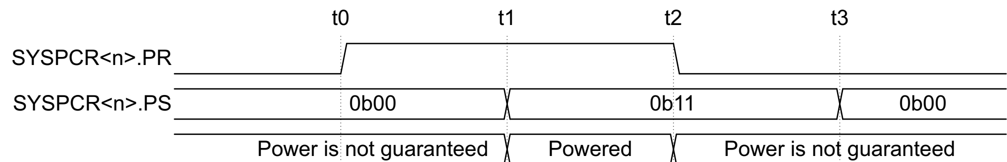


Figure D7-10 Power handshake

Between t1 and t2, power must be supplied to power domain `n`, and must not be removed.

If t1 is never reached because the system cannot power up the domain, SYSPSR<n>.PS must remain `0b00`. The debugger might choose to remove the power request by writing `0b0` to SYSPCR<n>.PR, although this practice is not recommended.

SYSPCR<n>.PR must not be used to initiate a power request when SYSPSR<n>.PS reads as `0b11`, because this value indicates that the handshake from a previous request is still completing.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing the SYSPSR<n>

The SYSPSR<n> for system power domain *n* can be accessed at the following address:

Component	Offset
ROM Table	`0xB80` + n×4

D7.5.20 SYSRSTAR, System Reset Acknowledge Register

The SYSRSTAR characteristics are:

Purpose

Used to indicate that a system reset has been completed, as an acknowledgement to a request that was made by using SYSRSTRR.

Usage constraints

PRIDR0.SYSRR indicates whether this register is implemented.

SYSRSTAR is accessible as follows:

Default
RO

Configurations

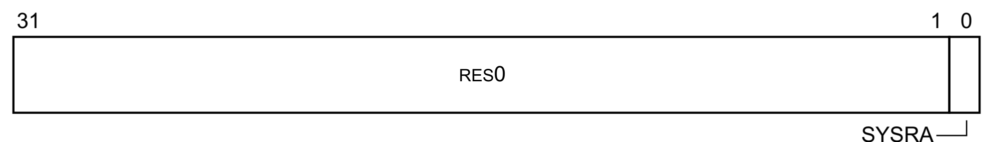
Included in all implementations.

Attributes

SYSRSTAR is a 32-bit register.

Field Descriptions

The SYSRSTAR bit assignments are:



Bits[31:1]

RES0.

SYSRA, bit[0]

System Reset Acknowledge:

`0b0` The system reset request is not initiated or not completed.

`0b1` The system reset request has been completed.

After a power-on reset, this field is set to `0b0`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SYSRSTAR

SYSRSTAR can be accessed at the following address:

Component	Offset
ROM Table	`0xC1C`

D7.5.21 SYSRSTRR, System Reset Request Register

The SYSRSTRR characteristics are:

Purpose

SYSRSTRR is used to request a reset of the entire system or subsystem. The scope of this reset is IMPLEMENTATION DEFINED.

SYSRSTRR is used with SYSRSTAR in a handshake mechanism that indicates when a reset has been completed.

Usage constraints

PRIDR0.SYSRR indicates whether this register is implemented.

System reset requests are ignored when invasive debug functionality is disabled. For details, see the description of the AUTHSTATUS register.

SYSRSTRR is accessible as follows:

Default
RW

Configurations

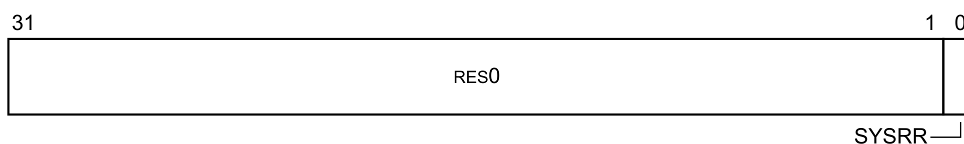
Included in all implementations.

Attributes

SYSRSTRR is a 32-bit register.

Field Descriptions

The SYSRSTRR bit assignments are:



Bits[31:1]

RES0.

SYSRR, bit[0]

System Reset Request:

`0b0` A system reset is not requested.

`0b1` A system reset is requested. The request remains asserted until this field is explicitly overwritten with the value `0b0`.

After a power-on reset, this field is set to `0b0`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SYSRSTRR

SYSRSTRR can be accessed at the following address:

Component	Offset
ROM Table	`0xC18`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D8
Topology Detection at the System Level

This chapter describes topology detection at the system level. It contains the following sections:

- *About topology detection at the system level* on page D8-212.
- *Detection* on page D8-213.
- *Components that are not recognized* on page D8-214.
- *Detection algorithm* on page D8-215.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1 About topology detection at the system level

Chapter B3 *Topology Detection* describes the topology detection requirements of CoreSight components. Chapter C7 *Topology Detection at the Component Level* describes how to perform topology detection on each interface type. This chapter describes how debuggers can use this information to detect the topology of a target system.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2 Detection

When connecting to a CoreSight system, a debugger performs the following steps:

1. It finds the Debug Port.
2. It ensures that the system is powered up, and that its clocks are running. The Debug Port provides facilities to assist with this assessment.
3. It looks for a ROM Table with the location of all components.
4. It compares the Peripheral ID of the ROM Table against a list of saved system descriptions. For information on this ID, see *PIDR0-PIDR7, Peripheral Identification Registers* on page D6-163.
5. If a description of the system with this ID is saved, it uses that description. If not, the debugger continues with the following steps:
   a. It identifies each component.
   b. It looks up information that is known about that component to determine what interfaces are supported and how to control them for topology detection.
   c. It performs topology detection. See *Detection algorithm* on page D8-215.
   d. It saves the description for later use.

D8.2.1 Saved descriptions

Because topology detection can be invasive, it is important that the description of the system is saved when discovered, so that the debugger can be connected noninvasively in the future. The debugger must have the possibility to force topology detection to be redone, in case two different targets are accidentally assigned the same ROM Table ID.

Note

Software running on the system must be able to determine the topology of the CoreSight system, and keep functioning when topology detection registers are enabled.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D8.3 Components that are not recognized

When an unrecognized component is encountered, the JEDEC code and CoreSight component class of the component is used to indicate which type of component has been encountered and who to ask for further information. Alternatively, DEVARCH, if present, can be used to determine the generic architecture of a component. The component must be otherwise ignored.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4 Detection algorithm

ARM recommends that a debugger connecting to a system executes the following algorithm, to determine the topology of the system:

```
for each component, c
        execute (component preamble) for c
for each interface type, t
        for each master interface and bidirectional interface of type t, m
                execute (master preamble) for interface m
        for each slave interface and bidirectional interface of type t, s
                execute (slave preamble) for interface s
        for each master interface and bidirectional interface of type t, m
                execute (master assert) for interface m
                for each slave interface and bidirectional interface of type t, s
                        if (slave check asserted) for interface s
                               record connection between m and s
                for each slave interface and bidirectional interface of type t, s
                        execute slave post-assert for interface s
                execute (master deassert) for interface m
                for each slave interface and bidirectional interface of type t, s
                        if not (slave check deasserted) for interface s
                                raise error
                for each slave interface and bidirectional interface of type t, s
                        execute (slave post-deassert) for interface s
for each component, c
        execute (component postamble) for c
```

Signals for topology detection on page C7-108 describes preambles, and assert and deassert sequences for common interfaces. If a component does not specify a preamble or postamble, they are as follows:

Component preamble

Set ITCTRL.IME to `0b1`.

Component postamble

Clear ITCTRL.IME to `0b0`.

Note

After a device has been in integration mode, it might behave differently than before. After performing integration or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components that are affected by the integration or topology detection.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D9
Compliance Requirements

This chapter describes the requirements for CoreSight compliance. It contains the following sections:

- *About compliance classes* on page D9-218.
- *CoreSight debug* on page D9-219.
- *CoreSight trace* on page D9-221.
- *Multiple DPs* on page D9-224.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D9.1 About compliance classes

This chapter defines the requirements that a system must meet to claim CoreSight compliance. It refers to specific revisions of components available from ARM.

These requirements are aimed at interoperability between debuggers, and only cover behavior that is visible to such tools. The following behavior is specified:

- Minimum functionality. This functionality must be available in all compliant systems.
- Optional functionality. ARM recommends that debuggers aiming to support compliant systems support this functionality.

Note

Systems can implement extra functionality, provided it does not affect the use of the minimum functionality. Debuggers might not be able to support this extra functionality.

Two levels of compliance are defined:

- CoreSight debug, which is the basic level of compliance. A processor supporting CoreSight debug does not need to comply with the CoreSight visible component architecture, although doing so makes it easier to build a CoreSight system.
- CoreSight trace, which includes all the requirements for CoreSight debug, and adds trace functionality.

The level of compliance is claimed for each individual processor in the system. For example, a system incorporating three processors might claim CoreSight trace for the first processor, CoreSight debug for the second, and no CoreSight compliance for the third.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D9.2 CoreSight debug

This section defines the CoreSight debug compliance class.

Note

A CoreSight component is a component that implements the CoreSight visible component architecture.

D9.2.1 Minimum debug functionality

Systems claiming CoreSight debug compliance must conform to the following rules:

- Each CoreSight system must contain exactly one DP, and implement a *JTAG Debug Port* (JTAG-DP) or a *Serial Wire Debug Port* (SW-DP) component. The JTAG or Serial Wire interface of the component must be accessible to debug tools. For more information about implementing multiple systems containing DPs, see *Multiple DPs* on page D9-224.
- All CoreSight components must comply with all the following requirements:
  - They must be accessible through a MEM-AP.
  - They must be discoverable through a valid ROM Table, that must itself conform to the requirements for CoreSight components.
- All processors claiming CoreSight debug compliance must observe at least one of the following requirements:
  - They must conform to the CoreSight visible component architecture, while conforming to the requirements for CoreSight components.
  - They must be accessible using a JTAG TAP Controller that is connected in series with the JTAG TAP Controller of the JTAG-DP, connected to the **TDI** side of the JTAG-DP as Figure D9-8 on page D9-224 shows.
  - They must be accessible using a JTAG TAP Controller that is connected in a chain of TAP Controllers that are controlled by the *JTAG Access Port* (JTAG-AP).
- All debug functionality must be visible and detectable, with its clocks running, when Debug Power Up is requested in the JTAG-DP programmers' model, except where it is hidden due to security restrictions.
- All debug functionality must be operational when System Power Up is requested in the JTAG-DP programmers' model, except where it is hidden due to security restrictions.
- All debug functionality must be reset to its initial state when Debug Reset is requested in the JTAG-DP programmers' model.
- For each CoreSight component and JTAG controlled processor, all inputs and outputs that are defined as type event are connected to a *Cross Trigger Interface* (CTI) component, unless there is only one component in the system with event inputs or outputs, in which case no CTI is required. For ARM JTAG controlled processors, the required connections are documented in the *CoreSight Technology System Design Guide*.
- All channel interfaces of CoreSight components, for example interfaces that are present on CTIs, are connected together, so that the channels are shared between all components. CoreSight technology from ARM provides a *Cross Trigger Matrix* (CTM) for connecting three or more channel interfaces together where required.
- Unless stated otherwise in this specification, extra logic between components that is visible to the tools is not permitted. See also *Variant interfaces* on page B3-68.
- ARM recommends all system designs to be CoreSight compliant, but recognizes that this recommendation might not always be achievable. If a system requires certain operations to be performed before it complies with the CoreSight compliance criteria, clearly state what these operations are, and clearly state that it is not CoreSight compliant until they have been performed.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D9.2.2 Optional debug functionality

CoreSight debug systems can also implement visibility of system components through a MEM-AP.

Single-core debug

Figure D9-1 shows the simplest CoreSight debug configuration for a single-core system. In this configuration, no trace capabilities are provided. The processor is accessed via a JTAG-AP, to ensure that it can be powered down without affecting other components on the master JTAG TAP chain.

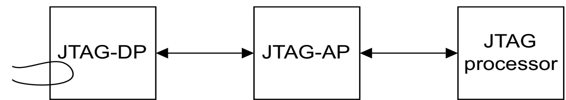


Figure D9-1 Single core with JTAG debug access

Multi-core debug

Figure D9-2 shows a multi-core CoreSight debug system:

- One of the processors is a fully compliant CoreSight component.
- Cross triggering is supported between processors.
- Both processors provide access to program the CTI and processor with interfaces that comply with the CoreSight architecture.

An alternative method to provide memory access that is not shown in the figure is to use AHB-AP.

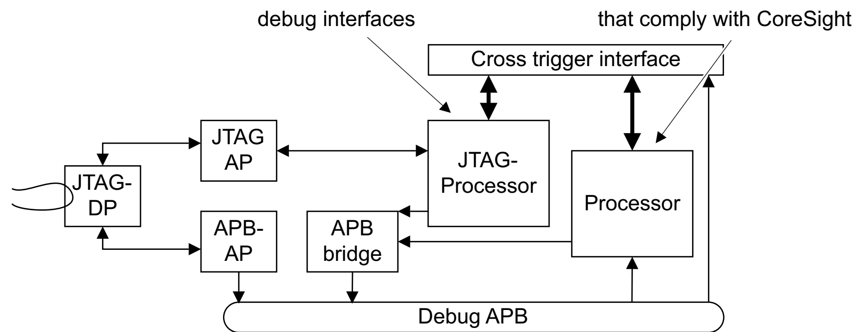


Figure D9-2 Multi-core system

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D9.3 CoreSight trace

This section defines the CoreSight trace compliance class.

D9.3.1 Minimum trace functionality

Systems claiming CoreSight trace compliance must comply with the minimum requirements for CoreSight debug, plus the following:

- All ARM-compatible processors claiming CoreSight trace compliance must implement an ARM CoreSight ETM.
- Processors that are not ARM-compatible must implement a trace solution that complies with the following requirements:
  - It must implement the CoreSight visible component architecture.
  - It must provide the processor with at least instruction trace as a CoreSight trace source.
- The system must implement one or more trace sinks:
  - If a TPIU is implemented, its output is connected to a compliant connector as defined in Chapter D3 *Physical Interface*.
- All CoreSight trace sources must drain into one or more of the trace sinks:
  - Where two or more trace sources drain into the same trace sink, they are connected through one or more CoreSight trace funnels.
  - The trace cannot travel through multiple paths to reach the same endpoint. See the example in Figure D9-3.

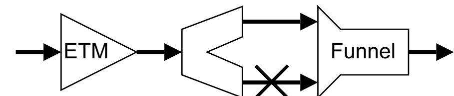


Figure D9-3 Non-compliant Replicator and CoreSight trace funnel connection

A particular example that must be avoided is feedback. See example in Figure D9-4.

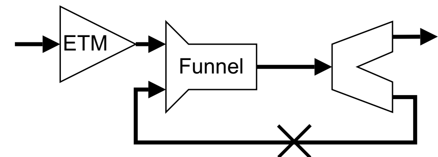


Figure D9-4 Non-compliant feedback loop

D9.3.2 Optional trace functionality

CoreSight debug systems can also implement CoreSight debug optional functionality and tracing of AHB buses using the ARM *AHB Trace Macrocell* (HTM).

Basic single-core trace

Figure D9-5 on page D9-222 shows an example system with single-core trace using the CoreSight infrastructure. The ETM, which complies with the CoreSigth architecture, outputs directly to a TPIU for direct output of core trace off-chip. The tracing of only a single trace source enables the TPIU to be configured in bypass mode because source IDs do not need to be embedded in the trace data.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

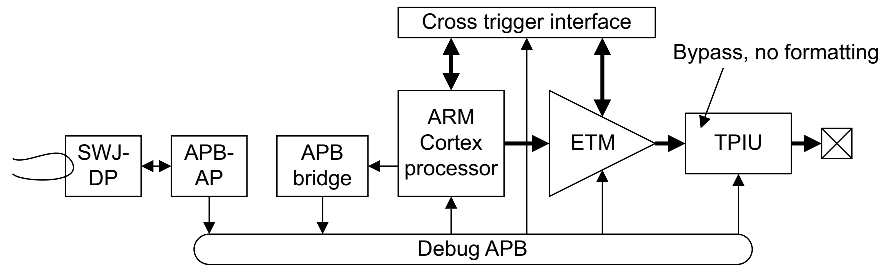


Figure D9-5 Single-core trace with formatting bypass

Advanced single-core trace

Figure D9-6 shows an example system with full trace capabilities in a single-core system. The ETM provides ARM processor tracing, and the HTM provides bus tracing. The CoreSight trace funnel combines trace from both sources into a single trace stream, that is then replicated to provide on-chip storage using the CoreSight ETB and output off-chip using the TPIU. Components can be configured via the Debug-APB and cross triggered using the CTIs, through the CTM.

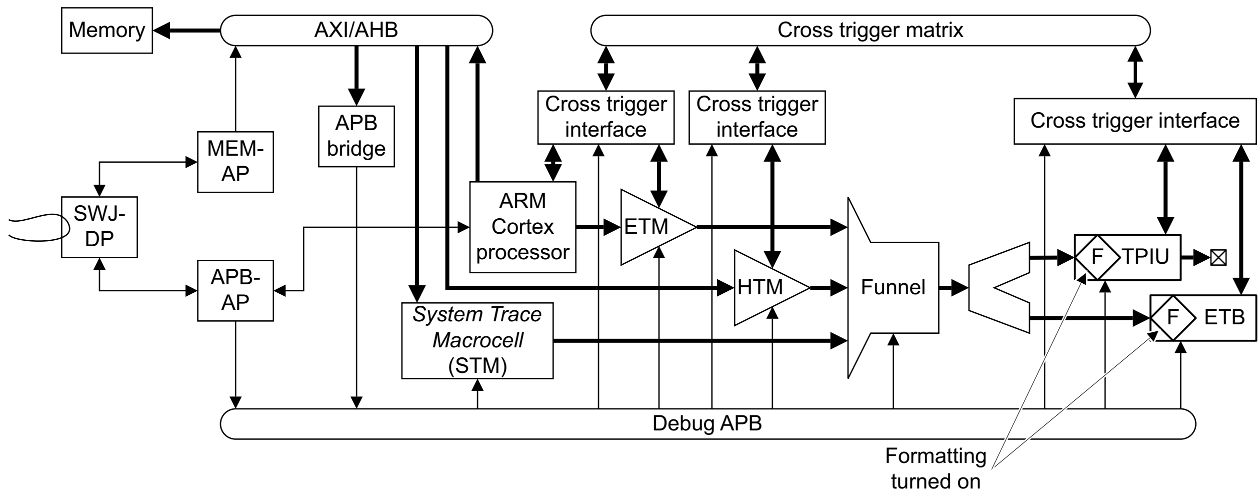


Figure D9-6 Full CoreSight trace with single core

Multi-core trace

Figure D9-7 on page D9-223 shows a system with an ARM processor and a DSP. A third smaller subsystem is added to support merging of multiple CoreSight AMBA ATB interfaces into a single trace stream.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

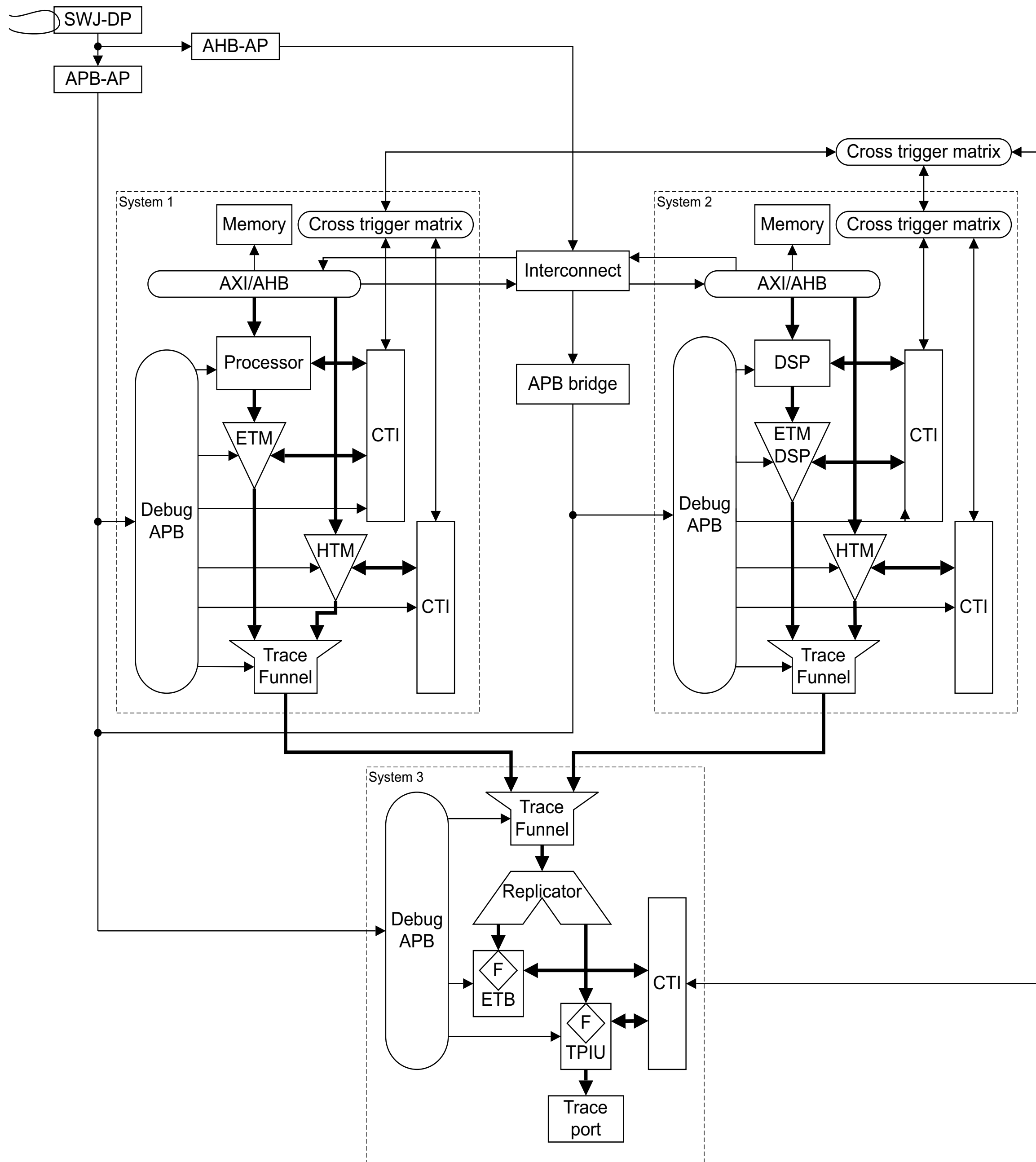


Figure D9-7 Full system trace with ARM processor and CoreSight compliant DSP

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

D9.4 Multiple DPs

In the context of this specification, a system is defined as one of the following:

- All components that are accessible through a single DP, a MEM-AP, or a JTAG-AP.
- All components before a JTAG-DP in a serial JTAG TAP chain.

The following rules apply to the arrangement of multiple DPs:

- Connections between JTAG TAP Controllers cannot be interleaved between systems. For example, if there are two systems sharing a JTAG TAP chain, each with a JTAG-DP and two JTAG processors connected in series with the JTAG-DP, the connections that are shown in Figure D9-8 are permitted, while the connections shown in Figure D9-9 are not permitted.

  In Figure D9-8:

  — System 1 is defined as the two processors before the first JTAG-DP in the TAP chain.

  — System 2 is defined as the two processors before the second JTAG-DP in the TAP chain.

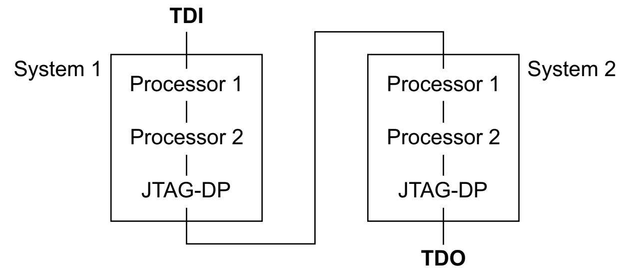


Figure D9-8 JTAG connections across systems

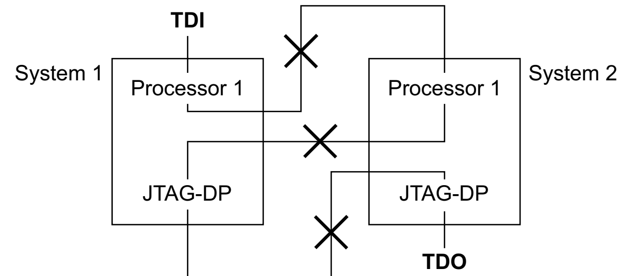


Figure D9-9 Non-compliant interleaved JTAG connections across systems

- Extra JTAG TAP Controllers can be implemented in series with JTAG TAP Controllers of the CoreSight systems. For example, in Figure D9-10, processor A is not part of either CoreSight system 1 or 2. The debugger considers processor A to be part of system 2, because the JTAG-DP closest to the **TDO** side of processor A is in system 2. If the debugger does not recognize processor A, then it is ignored, otherwise the debugger attempts topology detection on system 2 with processor A, and fails to find any connections between processor A and system 2.

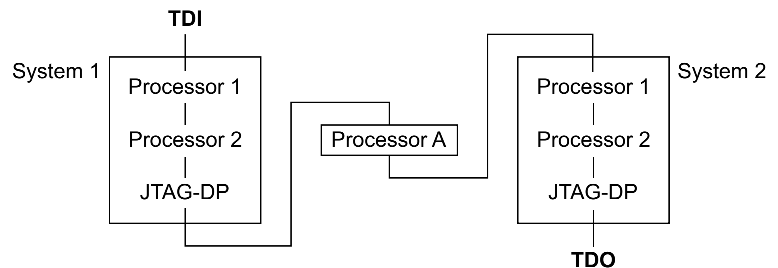


Figure D9-10 Systems with extra JTAG TAP Controllers

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- A JTAG-DP must not be accessed through the JTAG-AP of another system, as shown in Figure D9-11 on page D9-225.

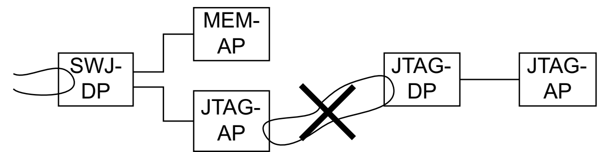


Figure D9-11 Non-compliant JTAG-DP connection

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Part E

Appendixes

Appendix E1
Power Requestor

This appendix describes the power requestor which ARM recommends that some CoreSight components implement. It contains the following sections:

- *About the power requestor* on page E1-230.
- *Register descriptions* on page E1-231.
- *Powering non-visible components* on page E1-248.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E1.1 About the power requestor

The power requestor belongs to the component class 0x9, CoreSight component.

The power requestor can control the application or removal of power for up to 32 power domains.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2 Register descriptions

Table E1-1 shows the power requestor registers, in order of their address offset in the 4KB block where the programmers' model resides. The remainder of the chapter describes how to implement the registers, in alphabetical order.

Table E1-1 Power requestor register summary

Offset	Type	Name	Description
`0x000`	RW	CDBGPWRUPREQ	Debug Power Request Register
`0x004`	RO	CDBGPWRUPACK	Debug Power Request Acknowledge Register
`0x008-0xEFC`	RES0	-	Reserved
`0xF00`	RW	ITCTRL	Integration Mode Control Register
`0xF04-0xF9C`	RES0	-	Reserved
`0xFA0`	RW	CLAIMSET	Claim Tag Set Register
`0xFA4`	RW	CLAIMCLR	Claim Tag Clear Register
`0xFA8-0xFAC`	RES0	-	Reserved
`0xFB0`	WO	LAR	Software Lock Access Register
`0xFB4`	RO	LSR	Software Lock Status Register
`0xFB8`	RO	AUTHSTATUS	Authentication Status Register
`0xFBC`	RO	DEVARCH	Device Architecture Register
`0xFC0-0xFC4`	RES0	-	Reserved
`0xFC8`	RO	DEVID	Device configuration Register
`0xFCC`	RO	DEVTYPE	Device Type identifier Register
`0xFD0-0xFDC`	RO	PIDR4-PIDR7	Peripheral Identification Registers
`0xFE0-0xFEC`	RO	PIDR0-PIDR3	
`0xFF0-0xFFC`	RO	CIDR0-CIDR3	Component Identification Registers

E1.2.1 AUTHSTATUS, Authentication Status Register

For a full description of this register, see *AUTHSTATUS, Authentication Status Register* on page B2-45.

The AUTHSTATUS characteristics are:

Purpose

Reports the required security level and status of the authentication interface. Where functionality changes on a given security level, this change in status must be reported in this register.

Usage constraints

AUTHSTATUS is accessible as follows:

Default
RO

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Configurations

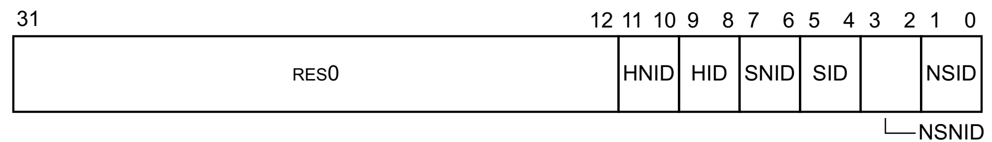
Included in all implementations.

Attributes

AUTHSTATUS is a 32-bit register.

Field Descriptions

The AUTHSTATUS bit assignments are:



Bits[31:12]

RES0.

HNID, bits[11:10]

Hypervisor non-invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor non-invasive debug are not implemented, or no hypervisor is implemented. For ARMv7 processors that implement the Virtualization Extensions, and for ARMv8 processors that implement EL2, if separate controls for hypervisor debug visibility are not implemented, the hypervisor debug visibility is indicated by the relevant Non-secure debug visibility fields NSNID and NSID. See the relevant ARM Architecture Manual for more information about Virtualization Extensions and EL2.

`0b10` Supported and disabled.
(**HIDEN** | **HNIDEN**) & (**DBGEN** | **NIDEN**) == FALSE.

`0b11` Supported and enabled.
(**HIDEN** | **HNIDEN**) & (**DBGEN** | **NIDEN**) == TRUE.

All other values are reserved.

HID, bits[9:8]

Hypervisor invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor invasive debug are not implemented, or no hypervisor is implemented. For ARMv7 processors that implement the Virtualization Extensions, and for ARMv8 processors that implement EL2, if separate controls for hypervisor debug visibility are not implemented, the hypervisor debug visibility is indicated by the relevant Non-secure debug visibility fields NSNID and NSID. See the relevant ARM Architecture Manual for more information about Virtualization Extensions and EL2.

`0b10` Supported and disabled. (**HIDEN** & **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**HIDEN** & **DBGEN**) == TRUE.

All other values are reserved.

SNID, bits[7:6]

Secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == FALSE.

`0b11` Supported and enabled.
(**SPIDEN** | **SPNIDEN**) & (**DBGEN** | **NIDEN**) == TRUE.

All other values are reserved.

SID, bits[5:4]

Secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**SPIDEN** & **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**SPIDEN** & **DBGEN**) == TRUE.

All other values are reserved.

NSNID, bits[3:2]

Non-secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. (**NIDEN** | **DBGEN**) == FALSE.

`0b11` Supported and enabled. (**NIDEN** | **DBGEN**) == TRUE.

All other values are reserved.

NSID, bits[1:0]

Non-secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

`0b10` Supported and disabled. **DBGEN** == FALSE.

`0b11` Supported and enabled. **DBGEN** == TRUE.

All other values are reserved.

Accessing AUTHSTATUS

AUTHSTATUS can be accessed at the following address:

Component	Offset
GPR	`0xFB8`

E1.2.2 CDBGPWRUPACK, Debug Power Request Acknowledge Register

The CDBGPWRUPACK characteristics are:

Purpose

Returns the status of the power requests that CDBGPWRUPREQ issues.

Usage constraints

The register can monitor the power requests for up to 32 power domains.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

CDBGPWRUPACK is accessible as follows:

Default
RO

Configurations

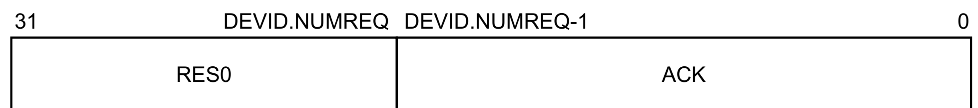
Included in all implementations.

Attributes

CDBGPWRUPACK is a 32-bit register.

Field Descriptions

The CDBGPWRUPACK bit assignments are:



Bits[31:DEVID.NUMREQ]

RES0.

ACK, bits[DEVID.NUMREQ-1:0]

The size of this field is IMPLEMENTATION DEFINED, and equals the value of DEVID.NUMREQ.

Permitted values of bit[*n*] are:

`0` Power domain *n* is not powered.

`1` Power domain *n* is powered.

Accessing CDBGPWRUPACK

CDBGPWRUPACK can be accessed at the following address:

Component	Offset
GPR	`0x004`

E1.2.3 CDBGPWRUPREQ, Debug Power Request Register

The CDBGPWRUPREQ characteristics are:

Purpose

Controls whether a power request is active for a power domain.

Usage constraints

CDBGPWRUPREQ can issue power requests for up to 32 power domains.

CDBGPWRUPREQ is accessible as follows:

Default
RW

 Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Configurations

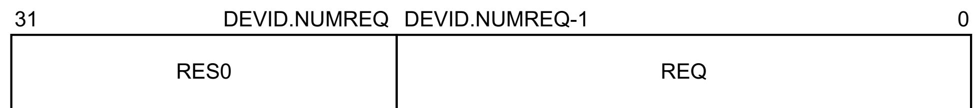
Included in all implementations.

Attributes

CDBGPWRUPREQ is a 32-bit register.

Field Descriptions

The CDBGPWRUPREQ bit assignments are:



Bits[31:DEVID.NUMREQ]

RES0.

REQ, bits[DEVID.NUMREQ-1:0]

The size of this field is IMPLEMENTATION DEFINED, and equals the value of DEVID.NUMREQ.

Permitted values of bit[*n*] are:

`0` Power request for power domain *n* is not active.

`1` Power request for power domain *n* is active.

Accessing CDBGPWRUPREQ

CDBGPWRUPREQ can be accessed at the following address:

Component	Offset
GPR	`0x000`

E1.2.4 CIDR0-CIDR3, Component Identification Registers

This section describes the bit assignments for GPR components that implement the CIDR0-CIDR3. For a full description of the CIDR, see *Component and Peripheral Identification Registers* on page B2-38.

The CIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

CIDR0-CIDR3 are accessible as follows:

Default
RO

Configurations

Included in all implementations.

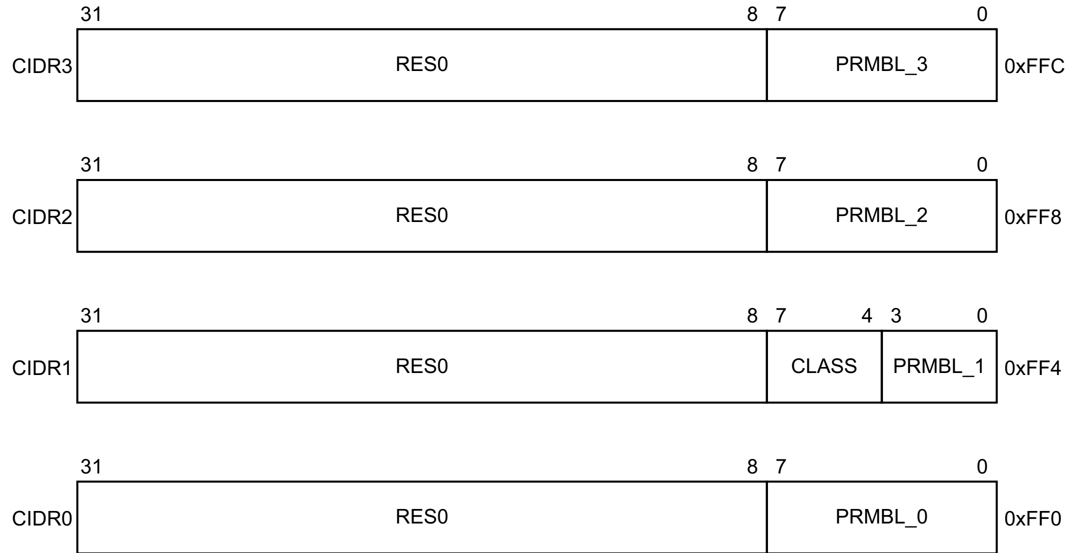
Attributes

CIDR0-CIDR3 are four 32-bit management registers.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Field Descriptions

The CIDR0 bit assignments are:



CIDR3 bits[31:8]

RES0.

PRMBL_3, CIDR3 bits[7:0]

`0xB1`.

CIDR2 bits[31:8]

RES0.

PRMBL_2, CIDR2 bits[7:0]

`0x05`.

CIDR1 bits[31:8]

RES0.

CLASS, CIDR1 bits[7:4]

`0x9` CoreSight component.

PRMBL_1, CIDR1 bits[3:0]

`0x0`.

CIDR0 bits[31:8]

RES0.

PRMBL_0, CIDR0 bits[7:0]

`0x0D`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing the CIDR

CIDR0-CIDR3 can be accessed at the following address:

Component	Offset			
	CIDR0	CIDR1	CIDR2	CIDR3
GPR	0xFF0	0xFF4	0xFF8	0xFFC

E1.2.5 CLAIMCLR, Claim Tag Clear Register

For a full description of this register, and how to deploy it in a claim tag protocol, see *CLAIMSET and CLAIMCLR, Claim Tag Set Register and Claim Tag Clear Register* on page B2-47.

The CLAIMCLR characteristics are:

Purpose

Clears claim tags and returns the current claim tag values.

Usage constraints

Must be used with CLAIMSET.

To indicate the width of the area that represents valid claim tags, a component must use CLAIMSET.

If CLAIMCLR and CLAIMSET are implemented, all debug agents that use them must implement a common claim tag protocol.

The value of CLAIMCLR must be zero at reset.

CLAIMCLR is accessible as follows:

Default
RW

Configurations

Included in all implementations.

Attributes

CLAIMCLR is a 32-bit management register.

Field Descriptions

The CLAIMCLR bit assignments are:

Bits[31:nTags]

RAZ/WI.

CLR, bits[nTags-1:0]

The size n of this field is IMPLEMENTATION DEFINED, and equals the number of bits set in CLAIMSET.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Permitted values of bit[*n*] are:

Write `0` No effect.

Write `1` Clear the claim tag for bit[*n*].

Read `0` The debug functionality that is tagged by bit[*n*] is available.

Read `1` The debug functionality that is tagged by bit[*n*] is claimed.

Accessing CLAIMCLR

CLAIMCLR can be accessed at the following address:

Component	Offset
GPR	`0xFA4`

E1.2.6 CLAIMSET, Claim Tag Set Register

For a full description of this register, and how to deploy it in a claim tag protocol, see *CLAIMSET and CLAIMCLR, Claim Tag Set Register and Claim Tag Clear Register* on page B2-47.

The CLAIMSET characteristics are:

Purpose

Sets claim tags and returns the valid claim tags.

Usage constraints

Must be used with CLAIMSET.

The bits indicating valid claim tags must be consecutive.

If CLAIMCLR and CLAIMSET are implemented, all debug agents that use them must implement a common claim tag protocol.

CLAIMSET is accessible as follows:

Default
RW

Configurations Included in all implementations.

Attributes

CLAIMSET is a 32-bit management register.

Field Descriptions

The CLAIMSET bit assignments are:

Bits[31:nTags]

RAZ/WI.

SET, bits[nTags-1:0]

The size n of this field is IMPLEMENTATION DEFINED.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Permitted values of bit[*n*] are:

Write `0` No effect.

Write `1` Set the claim tag for bit[*n*].

Read `0` The claim tag that is represented by bit[*n*] is implemented.

Read `1` The claim tag that is represented by bit[*n*] is not implemented.

Accessing CLAIMSET

CLAIMSET can be accessed at the following address:

Component	Offset
GPR	`0xFA0`

E1.2.7 DEVARCH, Device Architecture Register

This section describes the bit assignments for GPR components that implement DEVARCH. For a full description of DEVARCH, see *DEVARCH, Device Architecture Register* on page B2-51.

The DEVARCH characteristics are:

Purpose

Identifies the architect and architecture of a CoreSight component. The architect might differ from the designer of a component, for example when ARM defines the architecture but another company designs and implements the component.

Usage constraints

DEVARCH is accessible as follows:

Default
RO

Configurations

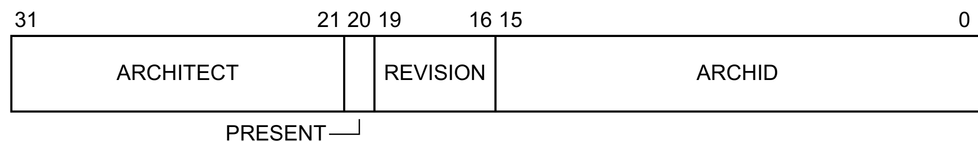
Included in all implementations.

Attributes

DEVARCH is a 32-bit management register.

Field Descriptions

The DEVARCH bit assignments are:



ARCHITECT, bits[31:21]

`0x23B` The architect is ARM.

PRESENT, bit[20]

`1` DEVARCH is present.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION, bits[19:16]

`0x0`

ARCHID, bits[15:0]

`0xA34` Power Requestor.

Accessing DEVARCH

DEVARCH can be accessed at the following address:

Component	Offset
GPR	`0xFBC`

E1.2.8 DEVID, Device configuration Register

This section describes the bit assignments for GPR components that implement DEVID. For a full description of DEVID, see *DEVID, Device Configuration Register* on page B2-53.

The DEVID characteristics are:

Purpose

Indicates how many power domains the power requestor supports.

Usage constraints

DEVID is accessible as follows:

Default
RO

Configurations

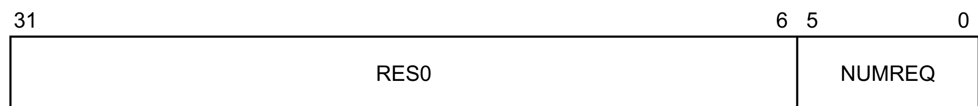
Included in all implementations.

Attributes

DEVID is a 32-bit management register.

Field Descriptions

The DEVID bit assignments are:



Bits[31:6]

RES0 Ensures that the bits that are not associated with the component type have a well-defined value.

NUMREQ, bits[5:0]

IMPLEMENTATION DEFINED

Number of power domains to be managed by CDBGPWRUPREQ and CDBGPWRUPACK.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DEVID

DEVID can be accessed at the following address:

Component	Offset
GPR	0xFC8

E1.2.9 DEVTYPE, Device Type Register

This section describes the bit assignments for GPR components that implement DEVTYPE. For a full description of DEVTYPE, see *DEVTYPE, Device Type Identifier Register* on page B2-55.

The DEVTYPE characteristics are:

Purpose

If the part number field is not recognized, a debugger can report the information that is provided by DEVTYPE about the component instead.

Usage constraints

DEVTYPE is accessible as follows:

Default
RO

Configurations

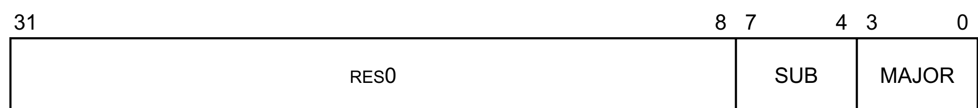
Included in all implementations.

Attributes

DEVTYPE is a 32-bit management register.

Field Descriptions

The DEVTYPE bit assignments are:



Bits[31:8]

RES0. Ensures that the bits that are not associated with the component type have a well-defined value.

SUB, bits[7:4]

`0x3` Power Requestor.

MAJOR, bits[3:0]

`0x4` Debug Control.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DEVTYPE

DEVTYPE can be accessed at the following address:

Component	Offset
GPR	`0xFCC`

E1.2.10 ITCTRL, Integration Mode Control Register

This section describes the bit assignments for GPR components that implement ITCTRL. For a full description of ITCTRL, see *ITCTRL, Integration Mode Control Register* on page B2-58.

The ITCTRL characteristics are:

Purpose

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

In integration mode, topology detection is enabled. For more information, see Chapter B3 *Topology Detection*.

Usage constraints

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.

ITCTRL is accessible as follows:

Default
RW

Configurations

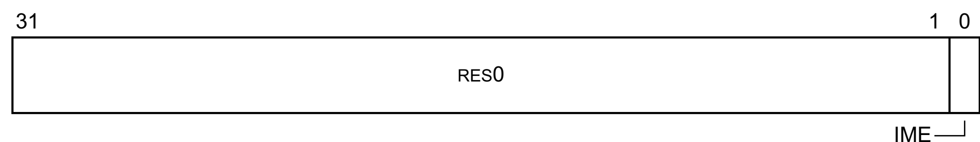
This register is not required. If no integration functionality is implemented, this register must be RAZ.

Attributes

ITCTRL is a 32-bit management register.

Field Descriptions

The ITCTRL bit assignments are:



Bits[31:1]

RES0.

IME, bits[0]

Permitted values of IME are:

`0` The component must enter functional mode.

`1` The component must enter integration mode, and enable support for topology detection and integration testing.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ITCTRL Register

ITCTRL can be accessed at the following address:

Component	Offset
GPR	`0xF00`

E1.2.11 LAR, Lock Access Register

For a full description of this register, and how to deploy it in a Software lock mechanism, see *LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.

The LAR characteristics are:

Purpose

Components can use this register to enable write access to device registers.

Usage constraints

LAR is accessible as follows:

Default
WO

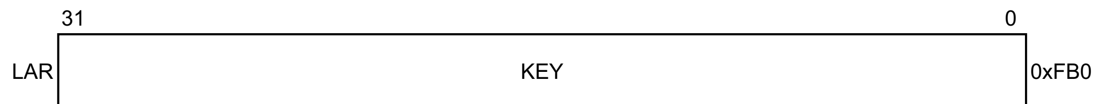
Configurations

LSR.SLI indicates whether a Software lock mechanism is implemented. If a Software lock mechanism is implemented, LAR is implemented, and must be used with LSR.**Attributes**

LAR is a 32-bit management register.

Field Descriptions

The LAR bit assignments are:



KEY, bits[31:0]

Writing a value to this field controls write access to the control registers.

Permitted values of KEY are:

Write `0xC5ACCE55`

Signals that LSR must permit writing to the control registers.

Write any other value

Signals that LSR must block writing to the control registers.

Accessing LAR

LAR can be accessed at the following address:

Component	Offset
GPR	`0xFB0`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2.12 LSR, Lock Status Register

For a full description of this register, and how to deploy it in a Software lock mechanism, see *LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.

The LSR characteristics are:

Purpose

Defines the parameters for a Software lock mechanism that can be implemented to control write access to device registers.

Usage constraints

LSR is accessible as follows:

Default
RO

Configurations

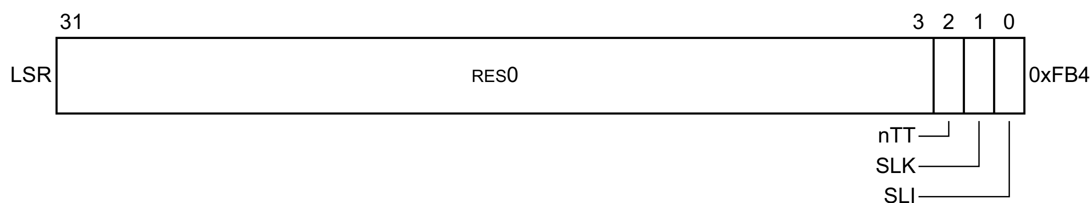
Included in all implementations.

Attributes

LSR is a 32-bit management register.

Field Descriptions

The LSR bit assignments are:



Bits[31:3]

RES0.

nTT, bits[2]

`0` LAR is a 32-bit register.

SLK, bits[1]

This field is used to return the current software lock status.

Permitted values of SLK are:

`0` Writing to the control registers must be permitted.

`1` Writing to the control registers must be blocked.

SLI, bits[0]

This field indicates whether a Software lock mechanism is implemented.

Permitted values of SLI are:

`0` Software lock mechanism is not implemented.

`1` Software lock mechanism is implemented.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing LSR

LSR can be accessed at the following address:

Component	Offset
GPR	`0xFB4`

E1.2.13 PIDR0-PIDR7, Peripheral Identification Register

This section describes the bit assignments for GPR components that implement PIDR0-PIDR7. For a full description of the PIDR, see *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

The PIDR characteristics are:

Purpose

Provide information to identify a CoreSight component.

Usage constraints

PIDR0-PIDR7 are accessible as follows:

Default
IMPLEMENTATION DEFINED

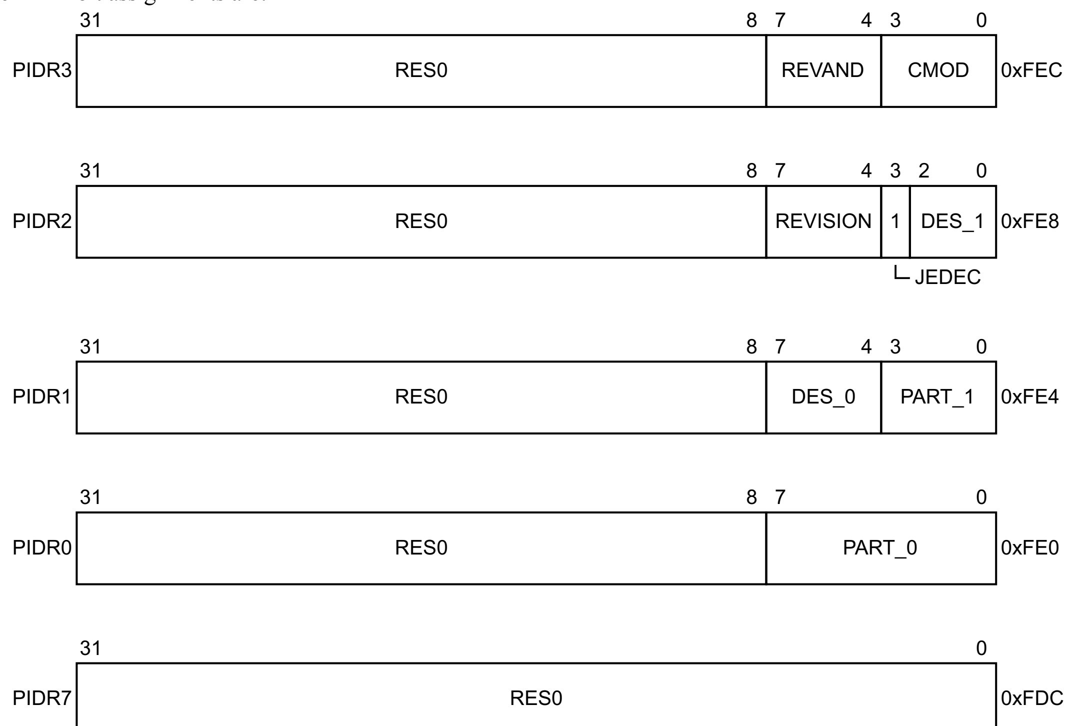
Configurations

Included in all implementations.

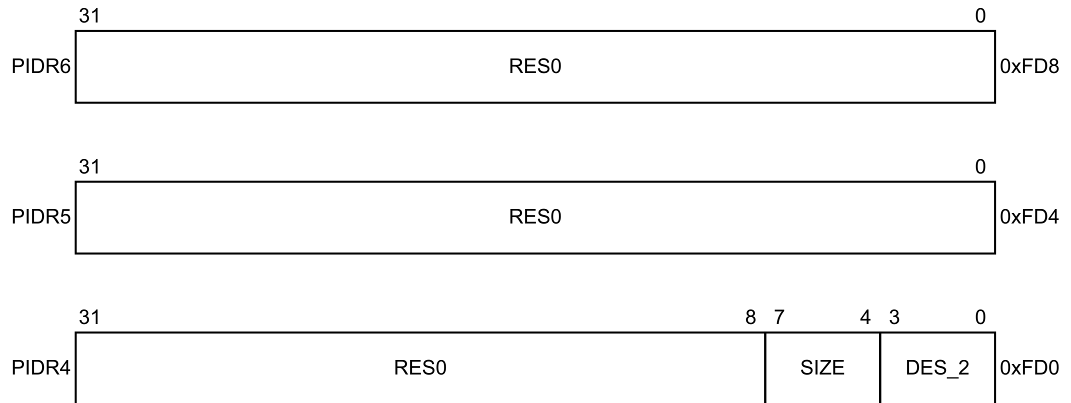
Attributes PIDR0-PIDR7 are eight 32-bit management registers.

Field Descriptions

The PIDR bit assignments are:



Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential



PIDR3 bits[31:8]

RES0.

REVAND, PIDR3 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

CMOD, PIDR3 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR2 bits[31:8]

RES0.

REVISION, PIDR2 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

JEDEC, PIDR2 bits[3]

`0b1`.

DES_1, PIDR2 bits[2:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR1 bits[31:8]

RES0.

DES_0, PIDR1 bits[7:4]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PART_1, PIDR1 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

PIDR0 bits[31:8]

RES0.

PART_0, PIDR0 bits[7:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

PIDR7 bits[31:0]

RES0.

PIDR6 bits[31:0]

RES0.

PIDR5 bits[31:0]

RES0.

PIDR4 bits[31:8]

RES0.

SIZE, PIDR4 bits[7:4]

`0x0` The GPR uses a single 4KB memory block.

DES_2, PIDR4 bits[3:0]

See register descriptions in *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.

Accessing the PIDR

PIDR0-PIDR7 can be accessed at the following address:

Component	Offset							
	PIDR0	PIDR1	PIDR2	PIDR3	PIDR4	PIDR5	PIDR6	PIDR7
GPR	`0xFE0`	`0xFE4`	`0xFE8`	`0xFEC`	`0xFD0`	`0xFD4`	`0xFD8`	`0xFDC`

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3 Powering non-visible components

Some components do not have a visible programmers' model, for example a *Cross Trigger Matrix* (CTM), which is used in cross-triggering components in a CoreSight system. When requesting power for a visible component, power must be supplied to any associated non-visible components as well.

For example, if two *Cross Trigger Interfaces* (CTIs) are connected through a CTM, a response to a power request for the CTIs must also power the CTM.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix E2
Revisions

This appendix describes the main technical changes between released versions of this specification.

Table E2-1 Differences between v1.0 and v2.0

Change	Location
Renamed register fields for consistency across ARM documentation.	Entire document.
Clarified that all registers are accessed in little-endian format.	*About the programmers' model* on page B2-32.
Added new registers to Class `0x9` CoreSight component.	*DEVID1, Device Configuration Register 1* on page B2-54. *DEVID2, Device Configuration Register 2* on page B2-55. *DEVARCH, Device Architecture Register* on page B2-51. *DEVAFF0-DEVAFF1, Device Affinity Registers* on page B2-50.
Added new interfaces.	Chapter C3 *Event Interface*. Chapter C6 *Timestamp Interface*.
Updated channel interface.	Chapter C4 *Channel interface*.
Updated the definition of the authentication interface to deprecate some previously permitted encodings.	Chapter C5 *Authentication Interface*.
Updated the connector information.	Chapter D3 *Physical Interface*.
Updated the permitted trace ID values to include `0x7D`.	*Special trace source IDs* on page D4-140.
Added the power requestor and ROM Table values.	Appendix E1 *Power Requestor*. *ROM Tables Overview* on page D5-146.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Table E2-2 Differences between v2.0 and v3.0

Change	Location
The use of LAR and LSR to implement the Software lock mechanism is deprecated.	*LSR and LAR, Software Lock Status Register and Software Lock Access Register* on page B2-59.
The use of PADDRDBG[31] to split the memory map and indicate the difference between external and internal accesses is deprecated.	*Debug APB interface memory map* on page D2-116
Use of the PIDR4.SIZE field is deprecated.	*Components that occupy more than 4KB of address space* on page B2-34 and *PIDR0-PIDR7, Peripheral Identification Registers* on page B2-40.
The AUTHSTATUS description is extended to include optional fields that can be used to indicate hypervisor debug visibility.	*AUTHSTATUS, Authentication Status Register* on page B2-45.
The Authentication Interface is extended to support signals that control debug for a hypervisor.	Chapter C5 *Authentication Interface*.
The rules for assigning a Unique Component Identifier and a revision number to a component have been updated.	Chapter B2 *Component and Peripheral Identification registers*.
Trace Formatter ID 0x7B, which was reserved in earlier versions, is allocated to indicate a flush response.	*Special trace source IDs* on page D4-140.
Introduced Class 0x9 ROM Tables, and adopted the designation Class 0x1 ROM Tables for the existing format.	Chapter D5 *About ROM Tables*. Chapter D6 *Class 0x1 ROM Tables*. Chapter D7 *Class 0x9 ROM Tables*.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix E3
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:

- *About ARM pseudocode* on page E3-252.
- *Data types* on page E3-253.
- *Expressions* on page E3-257.
- *Operators and built-in functions* on page E3-259.
- *Statements and program structure* on page E3-264.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.1 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 E3-253.
- *Expressions* on page E3-257.
- *Operators and built-in functions* on page E3-259.
- *Statements and program structure* on page E3-264.

E3.1.1 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 E3-264.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page E3-254.
- *Reals* on page E3-254.
- *Booleans* on page E3-254.
- *Enumerations* on page E3-254.
- *Lists* on page E3-255.
- *Arrays* on page E3-256.

E3.2.1 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.

E3.2.2 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 E3-259.

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 registers, memory locations, and instructions. All the remaining data types are abstract.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2.3 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} needs to be written in hexadecimal, it must be written as `-0x80000000`.

E3.2.4 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.

E3.2.5 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`.

E3.2.6 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};
```


Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2.7 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 above 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2.8 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 `..` followed by 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.3 Expressions

This section describes:

- *General expression syntax*.
- *Operators and functions - polymorphism and prototypes* on page E3-258.
- *Precedence rules* on page E3-258.

E3.3.1 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.

 Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.3.2 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.

E3.3.3 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 need not be 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.4 Operators and built-in functions

This section describes:

- *Operations on generic types*.
- *Operations on Booleans*.
- *Bitstring manipulation*.
- *Arithmetic* on page E3-262.

E3.4.1 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 as well as `'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`.

E3.4.2 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.

E3.4.3 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 just 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. For example, the encoding diagram for the DEVARCH shows its bit<20> as PRESENT. In such cases, the syntax DECVARCH.PRESENT is used as a more readable synonym for DEVARCH<20>.

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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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


Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
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;
```

E3.4.4 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`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

E3.5 Statements and program structure

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

- *Simple statements*.
- *Compound statements* on page E3-265.
- *Comments* on page E3-268.

E3.5.1 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>;
```

 Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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.

E3.5.2 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>
```


Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 E3-259.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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>
```

E3.5.3 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 `*/`.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

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

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 and AHB-Lite.

AHB Access Port (AHB-AP)

An optional component that provides an AHB interface to a SoC.

CoreSight supports access to a system bus infrastructure using the *AHB Access Port* (AHB-AP). The AHB-AP provides an AHB master port for direct access to system memory. Other bus protocols can use AHB bridges to map transactions. For example, you can use AHB to AXI bridges to provide AHB access to an AXI bus matrix.

See also Debug Access Port (DAP).

AHB Trace Macrocell (HTM)

A trace source that makes bus information visible. This information cannot be inferred from the processor using just a trace macrocell. HTM trace can provide:

- An understanding of multi-layer bus utilization.
- Software debug. For example, visibility of access to memory areas and data accesses.
- Bus event detection for trace trigger or filters, and for bus profiling.

See also AHB.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

AHB-Lite
A subset of the full AMBA AHB protocol specification. It provides all the basic functions that are required by most AMBA AHB slave and master designs, particularly when used with a multi-layer AMBA interconnect.

Aligned
A data item that is 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.

APB Access Port (APB-AP)
An optional component that provides an APB interface to a SoC, usually to its main functional buses.

APB-AP
See APB Access Port (APB-AP).

ATB
An AMBA bus protocol for trace data. A trace device can use an ATB to share CoreSight capture resources.

ATB bridge
A synchronous ATB bridge provides a register slice that meets timing requirements by adding a pipeline stage. It provides a unidirectional link between two synchronous ATB domains.

An asynchronous ATB bridge provides a unidirectional link between two ATB domains with asynchronous clocks, and connects components in different clock domains.

See also ATB.

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.

See also AXI coherency extensions (ACE).

AXI coherency extensions (ACE)
The *AXI coherency extensions* (ACE) provide extra channels and signaling to an AXI interface to support system level cache coherency.

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

See also Warm reset.

Core reset
See Warm reset.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

CoreSight ARM on-chip debug and trace components, that provide the infrastructure for monitoring, tracing, and debugging a complete system on chip.

See also CoreSight ECT and CoreSight ETM.

CoreSight ECT *See* Embedded Cross Trigger (ECT).

CoreSight ETB *See* Embedded Trace Buffer (ETB).

CoreSight ETM *See* Embedded Trace Macrocell (ETM).

Cross Trigger Interface (CTI)
Part of an *Embedded Cross Trigger* (ECT) device. In an ECT, the CTI provides the interface between a processor or ETM and the CTM.

Cross Trigger Matrix (CTM)
Part of an *Embedded Cross Trigger* (ECT) device. In an ECT, the CTM combines the trigger requests generated by CTIs and broadcasts them to all CTIs as channel triggers.

CTI *See* Cross Trigger Interface (CTI).

CTM *See* Cross Trigger Matrix (CTM).

DAP *See* Debug Access Port (DAP).

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

Debug Access Port (DAP)
A collection of Debug Ports and Access Ports that are compliant with the *ARM Debug Interface* (ADI), and provide access to system buses on a debug target.

Debug Test Access Port (DBGTAP)
A debug control and data interface based on IEEE 1149.1 JTAG *Test Access Port* (TAP). The interface has four or five signals.

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

DNM *See* Do-Not-Modify (DNM).

Do-Not-Modify (DNM) A value that must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same core.

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.

ECT *See* Embedded Cross Trigger (ECT).

Embedded Cross Trigger (ECT)
A modular system that supports the interaction and synchronization of multiple triggering events with an SoC. It comprises:
- *Cross Trigger Interface* (CTI).
- *Cross Trigger Matrix* (CTM).

Embedded Trace Buffer (ETB)
A Logic block that extends the information capture functionality of a trace macrocell.

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

ETB *See* Embedded Trace Buffer (ETB).

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

ETM
See Embedded Trace Macrocell (ETM).

Event
In an ARM trace macrocell:

Simple An observable condition that a trace macrocell can use to control aspects of a trace.

Complex A boolean combination of simple events that a trace macrocell can use to control aspects of a trace.

Formatter
In an ETB or TPIU, an internal input block that embeds the trace source ID in the data to create a single trace stream.

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.

HTM
See AHB Trace Macrocell (HTM).

IEEE 1149.1
The IEEE Standard that defines TAP. Commonly referred to as JTAG.

See *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture* specification available from the IEEE Standards Association `http://standards.ieee.org`.

IGN
An abbreviation for Ignore, when describing the behavior of a register or memory access.

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.

IMPLEMENTATION SPECIFIC
In the context of ARM trace macrocells, behavior that is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option that is chosen does not affect software compatibility.

When IMPLEMENTATION SPECIFIC is used with this meaning in body text, it is always in SMALL CAPITALS.

See also IMPLEMENTATION DEFINED.

In-Circuit Emulator
A device that provides access to the signals of a circuit while that circuit is operating, and lets you moderate those signals.

Instruction Synchronization Barrier (ISB)
An operation to ensure that any instruction that comes after the ISB operation is fetched only after the ISB has completed.

Instrumentation trace
A component for debugging real-time systems through a simple memory-mapped trace interface. It provides `printf` style debugging.

Intelligent Energy Manager
An energy manager solution consisting of both software and hardware components that function together to prolong battery life in an ARM processor-based device.

ISB
See Instruction Synchronization Barrier (ISB).

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Joint Test Action Group (JTAG)
An IEEE group that is focused 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 `http://standards.ieee.org`.

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

JTAG Access Port (JTAG-AP)
An optional component that provides debugger access to on-chip scan chains.

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

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

nSRST Abbreviation of *System Reset*. The signal that causes the target system other than the TAP Controller to be reset.

See also nTRST and Joint Test Action Group (JTAG).

nTRST Abbreviation of *TAP Reset*. The electronic signal that causes the target system TAP Controller to be reset.

See also nSRST and Joint Test Action Group (JTAG).

Power-on reset *See* Cold reset.

Program Flow Trace (PFT)
The *Program Flow Trace* (PFT) architecture assumes that any trace decompressor has a copy of the program being traced, and generally outputs only enough trace for the decompressor to reconstruct the program flow. However, its trace output also enables a decompressor to reconstruct the program flow when it does not have a copy of parts of the program, for example because the program uses self-modifying code.

A *Program Flow Trace Macrocell* (PTM) implements the Program Flow Trace architecture.

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 1, or to a field that reads as all 1s.

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

RAZ/WI Read-As-One, 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 on writes being ignored. This description can apply to a single bit that reads as 0, 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 1, 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 0, or to a field that reads as all 0s.

RealView ICE An ARM JTAG interface unit for debugging embedded processor cores that uses a DBGTAP or Serial Wire interface.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Replicator

In an ARM trace macrocell, a replicator enables two-trace sinks to be wired together and to operate independently on the same incoming trace stream. The input trace stream is output onto two independent ATB ports.

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.

Therefore, 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 0. In this case:
   - Reads of the bit always return 0.
   - 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 0.
   - A read of the bit returns the last value that is 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 that is 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 0.
- A read of the bit must return the value that was last successfully written to the bit, regardless of the use of the register when the bit was written.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 that is 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 0.
- Must use an *SBZP* policy to write to the bit.

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

- For a read-only bit, RES0 indicates that the bit reads as 0, 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 0, 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.

Therefore, 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 1. In this case:
   - Reads of the bit always return 1.
   - 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value that is 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 that is 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 1.
- 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 that is 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 1.
- Must use an *SBOP* policy to write to the bit.

The RES1 description can apply to bits or bitfields that are read-only or write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, 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 0, or to a field that should be written as its preserved value or as all 1s.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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:

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).

Serial wire debug (SWD)

A debug implementation that uses a serial connection between the SoC and a debugger.

The SWDP consists of two terminals that provide synchronous access to debug interfaces. The terminals are **SWDIO** and **SWCLK**.

Serial Wire Debug Port (SWDP)

The interface for serial wire debug.

Serial Wire JTAG Debug Port (SWJ - DP)

The SWJ - DP is a combined JTAG-DP and SWDP 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 1, 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 that was 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 1, 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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 0, 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 fields 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 that was 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 0, 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).

SWDP *See* Serial Wire Debug Port (SWDP).

SWJ - DP *See* Serial Wire JTAG Debug Port (SWJ - DP)

TAP Controller Logic on a device that enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE 1149.1.

See also Joint Test Action Group (JTAG).

TCD *See* Trace Capture Device (TCD).

TCK The clock for the TAP data lines **TMS**, **TDI**, and **TDO**.

See also Test Data Input (TDI), Test Data Output (TDO).

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), TAP Controller, TCK, Test Data Input (TDI), Test Data Output (TDO), TMS.

Test Data Input (TDI) *Test Data Input* (TDI) is the input to a TAP Controller from the data source (upstream). Usually, this input connects the RealView ICE run control unit to the first TAP controller.

See also Joint Test Action Group (JTAG), RealView ICE, *and* TAP Controller.

Test Data Output (TDO)

Test Data Output (TDO) is the electronic signal output from a TAP Controller to the downstream data sink. Usually, this output connects the last TAP controller to the RealView ICE run control unit.

See also Joint Test Action Group (JTAG), RealView ICE, TAP Controller.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

TMS Test Mode Select.

TPA *See* Trace Port Analyzer (TPA).

TPIU *See* Trace Port Interface Unit (TPIU).

Trace Capture Device (TCD)
A generic term to describe Trace Port Analyzers, logic analyzers, and on-chip trace buffers.

Trace funnel In an ARM trace macrocell, a device that combines multiple trace sources onto a single bus.

See also AHB Trace Macrocell (HTM), CoreSight.

Trace hardware A term for a device that contains an ARM trace macrocell.

Trace port A port on a device, such as a processor or ASIC, used to output trace information.

Trace Port Analyzer (TPA)
A hardware device that captures trace information output on a trace port. This device can be a low-cost product that is designed specifically for trace acquisition, or a logic analyzer.

Trace Port Interface Unit (TPIU)
Drains trace data and acts as a bridge between the on-chip trace data and the data stream that is captured by a TPA.

Trigger In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the core. The exact information that is displayed depends on the position of the trigger within the buffer.

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.

UNK An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

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 that 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, if at least one instruction that is not unpredictable can be trapped to the hypervisor if executed at that lower level of privilege.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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.

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

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.

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.

Copyright © 2004, 2005, 2012, 2013, 2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential