Arm® Debug Interface Architecture Specification

ADIv5.0 to ADIv5.2

arm

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
ARM IHI 0031G (ID022122)

Arm Debug Interface Architecture Specification
ADIv5.0 to ADIv5.2

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
8 February 2006	A	Non-Confidential	First issue, for ADIv5.
14 May 2012	B	Confidential Beta	Second issue, for ADIv5.0 to ADIv5.2.
8 August 2013	C	Non-Confidential Final	Third issue, for ADIv5.0 to ADIv5.2.
9 March 2017	D	Non-Confidential	Fourth issue, for ADIv5.0 to ADIv5.2
30 April 2018	E	Non-Confidential	Fifth issue, for ADIv5.0 to ADIv5.2
24 July 2020	F	Non-Confidential	Sixth issue, for ADIv5.0 to ADIv5.2
25 February 2022	G	Non-Confidential	Seventh issue, for ADIv5.0 to ADIv5.2

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

Confidentiality Status

This document is Non-Confidential. Any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorised by Arm to disclose this document to you.

Product Status

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

Web Address

`http://www.arm.com`

Contents
Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2

Preface

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

Part A The Arm Debug Interface

Chapter A1 About the Arm Debug Interface

A1.1 ADI versions A1-20
A1.2 Purpose of the ADI A1-23
A1.3 The subdivisions of an ADIv5 implementation A1-25
A1.4 The Debug Port (DP) A1-27
A1.5 Access Ports (APs) A1-28
A1.6 Design choices and implementation examples A1-32

Part B The Debug Port

Chapter B1 About the DP

B1.1 MINDP, Minimal DP extension B1-40
B1.2 Sticky flags and DP error responses B1-41
B1.3 The transaction counter B1-43
B1.4 Pushed-compare and pushed-verify operations B1-44
B1.5 Power and reset control B1-46

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Chapter B2 **DP Reference Information**
B2.1 DP architecture versions B2-48
B2.2 DP register descriptions B2-53
B2.3 System and debug power control behavior B2-77
B2.4 Debug reset control behavior B2-82
B2.5 System reset control behavior B2-84

Chapter B3 **The JTAG Debug Port**
B3.1 About the JTAG-DP B3-86
B3.2 The scan chain interface B3-87
B3.3 IR scan chain and IR instructions B3-91
B3.4 DR scan chain and DR instructions B3-95

Chapter B4 **The Serial Wire Debug Port**
B4.1 About the SWD protocol B4-106
B4.2 SWD protocol operation B4-110
B4.3 SWD interface B4-122

Chapter B5 **The Serial Wire/JTAG Debug Port**
B5.1 About the SWJ-DP B5-126
B5.2 Switching between SWD and JTAG B5-128
B5.3 Dormant operation B5-131
B5.4 Restrictions on switching between operating modes B5-138

Part C **The Access Port**

Chapter C1 **About the AP**
C1.1 AP requirements C1-142
C1.2 Selecting and accessing an AP C1-143
C1.3 AP Programmers’ Model C1-144

Chapter C2 **The Memory Access Port**
C2.1 About the MEM-AP C2-148
C2.2 MEM-AP functions C2-152
C2.3 Implementing a MEM-AP C2-162
C2.4 MEM-AP examples of pushed-verify and pushed-compare C2-165
C2.5 MEM-AP Programmers’ Model C2-167
C2.6 MEM-AP register descriptions C2-168

Chapter C3 **The JTAG Access Port**
C3.1 About the JTAG-AP C3-190
C3.2 Operation of the JTAG-AP C3-195
C3.3 The JTAG Engine Byte Command Protocol C3-198
C3.4 JTAG-AP register summary C3-205
C3.5 JTAG-AP register descriptions C3-206

Chapter C4 **COM-AP programmers’ model**
4.1 About the COM-AP C4-220
4.2 COM-AP register map C4-221

Part D **ROM Tables**

Chapter D1 **About ROM Tables**
D1.1 ROM Tables Overview D1-226
D1.2 ROM Table Types D1-227

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

D1.3 Component and Peripheral ID Registers for ROM Tables D1-228
D1.4 Location of the ROM Table D1-229

Part E Appendixes

Appendix E1 Standard Memory Access Port Definitions

E1.1 Introduction E1-234
E1.2 AMBA AXI3 and AXI4 .... E1-235
E1.3 AMBA AXI4 with ACE-Lite E1-237
E1.4 AMBA AXI5 .... E1-240
E1.5 AMBA AHB3 .... E1-243
E1.6 AMBA AHB5 .... E1-245
E1.7 AMBA AHB5 with enhanced HPROT control E1-247
E1.8 AMBA APB2 and APB3 .... E1-249
E1.9 AMBA APB4 and APB5 .... E1-250

Appendix E2 Cross-over with the Arm Architecture

E2.1 Introduction E2-254
E2.2 Armv6-M, Armv7-M, and Armv8-M architecture profiles E2-255
E2.3 PEs with a physical address of up to 32 bits E2-256
E2.4 PEs with a physical address greater than 32 bits E2-257
E2.5 Summary of the requirements for ADIv5 implementations E2-258

Appendix E3 Pseudocode Definition

E3.1 About the Arm pseudocode E3-260
E3.2 Pseudocode for instruction descriptions E3-261
E3.3 Data types E3-263
E3.4 Operators E3-268
E3.5 Statements and control structures E3-274
E3.6 Built-in functions E3-279
E3.7 Miscellaneous helper procedures and functions E3-282
E3.8 Arm pseudocode definition index E3-284

Glossary

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Preface

This preface introduces the *Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*. It contains the following sections:

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this manual

This manual describes the *Architecture Specification* for the *ARM Debug Interface v5, ADIv5.0 to ADIv5.2* (ADIv5).

Intended audience

This specification is written for system designers and engineers who specify, design, or implement ADI5-compliant debug interfaces. The audience includes system designers and engineers who specify, design, or implement a *System-on-Chip* (SoC) that incorporates an ADIv5-compliant debug interface.

This specification is also intended for engineers who work with an ADIv5-compliant debug interface. This audience includes designers and engineers who:

- Specify, design, or implement hardware debuggers.
- Specify, design, or write debug software.

These engineers have no control over the design decisions that are made in the ADIv5 interface implementation to which they connect, but must be able to identify the ADIv5 interface components that are present, and understand how they operate.

This specification provides an architectural description of an ADIv5 interface. It does not describe how to implement the interface.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Using this book

This specification is organized into the following chapters:

Chapter A1 ***About the Arm Debug Interface***

Read this chapter for a high-level view of the *Arm Debug Interface* (ADI). This chapter defines the logical subdivisions of an ADI, and summarizes the design choices that are made when implementing an ADI.

Chapter B1 ***About the DP***

Every ADI includes a single *Debug Port* (DP). The DP can be one of several types: a *JTAG Debug Port* (JTAG-DP), a *Serial Wire Debug Port* (SW-DP), or a *Serial Wire/JTAG Debug Port* (SWJ-DP). Read this chapter for a description of the features that must be implemented on the DP of any ADI.

Chapter B2 ***DP Reference Information***

Read this chapter for detailed reference information that applies to all DP types.

Chapter B3 ***The JTAG Debug Port***

Read this chapter for a description of the *JTAG Debug Port* (JTAG-DP), and in particular, the *Debug Test Access Port State Machine* (DBGTAPSM) and the scan chains that access the JTAG-DP.

Chapter B4 ***The Serial Wire Debug Port***

Read this chapter for a description of the *Serial Wire Debug Port* (SW-DP), and the *Serial Wire Debug* (SWD) protocols, which are used to access an SW-DP.

Chapter B5 ***The Serial Wire/JTAG Debug Port***

Read this chapter for a description of multiple protocol interoperability as implemented in the *Serial Wire/JTAG Debug Port* (SWJ-DP) CoreSight component.

Chapter C1 ***About the AP***

Read this chapter for a description of ADI *Access Ports* (APs), and details of the features that every AP must implement.

Chapter C2 ***The Memory Access Port***

Read this chapter for a description of the ADI *Memory Access Port* (MEM-AP).

Chapter C3 ***The JTAG Access Port***

Read this chapter for a description of the ADI *JTAG Access Port* (JTAG-AP).

Chapter C4 ***COM-AP programmers' model***

Read this chapter for a description of the COM-AP programmers' model.

Chapter D1 ***About ROM Tables***

Read this chapter for a general description of Arm debug component ROM Tables. Any ADI can include a ROM Table. An ADI with more than one debug component must include at least one ROM Table.

Appendix E1 ***Standard Memory Access Port Definitions***

Read this appendix for information on implementing the *Memory Access Port* (MEM-AP).

Appendix E2 ***Cross-over with the Arm Architecture***

Read this appendix for a description of the required or recommended options for the Arm Debug Interface for Arm architecture profiles.

Appendix E3 ***Pseudocode Definition***

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

Read the Glossary for definitions of some of the terms that are used in this manual.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Conventions

The following sections describe conventions that this specification 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 citations.

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

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

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

SMALL CAPITALS

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

Colored text Indicates a link:

- A URL, for example `https://developer.arm.com`.
- A cross-reference, that, if it is not on the current page, includes the page number of the referenced information. For example, *Pseudocode descriptions* 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 *AMBA*.

Signals

The signal conventions are:

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

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

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

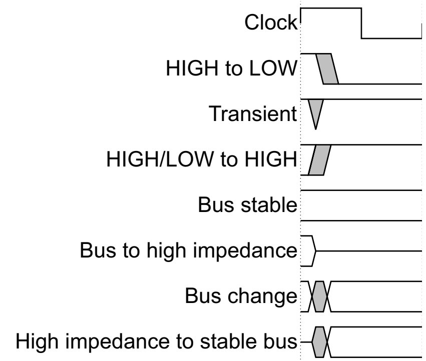
Prefix DBG Denotes debug signals.

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. Do not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.



Key to timing diagram conventions

Timing diagrams sometimes show single-bit signals as HIGH and LOW at the same time and they look similar to the bus change shown in *Key to timing diagram conventions*. If a timing diagram shows a single-bit signal in this way, its value does not affect the accompanying description.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`.

Pseudocode descriptions

This specification 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 © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Additional reading

This section lists relevant publications from Arm and third parties.

See Arm Developer https://developer.arm.com, for access to Arm documentation.

Arm publications

See the following documents for other information that is relevant to this specification:

- *Arm® Architecture Reference Manual, for A-profile architecture* (ARM DDI 0487).
- *Arm® Architecture Reference Manual, Armv7-A and Armv7-R edition* (ARM DDI 0406).
- *Arm® v8-M Architecture Reference Manual* (ARM DDI 0553).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *Arm® CoreSight™ SoC-400 Technical Reference Manual* (ARM 100536).
- *AMBA® AXI™ and ACE™ Protocol Specification* (ARM IHI 0022).
- *Arm® AMBA® 5 AHB™ Protocol Specification* (ARM IHI 0033).
- *AMBA® APB Protocol Specification* (ARM IHI 0024).
- *Arm1136JF-S™ and Arm1136J-S™ Technical Reference Manual* (ARM DDI 0211).
- *Advanced Communications Channel™ Architecture Specification* (ARM IHI 0076).

Other publications

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

- *IEEE Standard Test Access Port and Boundary Scan Architecture* (IEEE 1149.1-2001).
- *JEDEC Standard Manufacturer's Identification Code* (JEDEC JEP106).

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, ARM IHI 0031G.
- 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.

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive.

Arm strives to lead the industry and create change.

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact `terms@arm.com`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Part A
The Arm Debug Interface

Chapter A1
About the Arm Debug Interface

This chapter introduces the *Arm Debug Interface* (ADI) architecture and summarizes the design decisions that are required for an ADI implementation. It contains the following sections:

- *ADI versions* on page A1-20.
- *Purpose of the ADI* on page A1-23.
- *The subdivisions of an ADIv5 implementation* on page A1-25.
- *The Debug Port (DP)* on page A1-27.
- *Access Ports (APs)* on page A1-28.
- *Design choices and implementation examples* on page A1-32.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

A1.1 ADI versions

The *Arm Debug Interface version 5* (ADIv5) is the fifth major version of the Arm Debug Interface.

Note

The term ADIv5 refers to ADIv5.0, ADIv5.1, ADIv5.2, or any other release of the fifth major revision of ADI.

ADI versions before v5 are based on the IEEE 1149.1 JTAG interface, but are intended only for accessing Arm processor cores and *Embedded Trace Macrocells* (ETMs):

Debug interface versions 1 and 2

Implemented on the ARM7TDMI® and ARM9® families of processor cores.

Debug interface version 3

Introduced for the ARM10™ processor family.

ADIv4

The first version of the ADI to be linked with an Arm architecture version, rather than an implementation of an Arm processor core. Arm recommends that ADIv4 is used with implementations of the Armv5 architecture.

ADIv5 has the following major advantages:

- ADIv5 interfaces can access a greater range of devices.
- Implementing the ADI can be separated from implementing the resource, which makes it easier to reuse implementations. However, this separation is not required.
- Use of the ADIv5 abstractions permits reusing software tools, for example debuggers.

Debug interfaces that implement ADI versions older than ADIv5 require the physical connection to the interface to use an IEEE 1149.1 JTAG interface. ADIv5 specifies two alternatives for the physical connection:

- An IEEE 1149.1 JTAG interface.
- A Serial Wire Debug interface with a low pin count.

The main components of ADIv5 are split between two main architectures:

- The AP architecture.
- The DP architecture.

A1.1.1 About the minor versions of ADIv5

ADI versions 5.1 and 5.2 are backwards-compatible extensions of the original ADIv5 specification. From the introduction of ADIv5.1, the original ADIv5 specification is described as ADIv5.0.

Note

ADIv5.1 was originally defined only as a supplement to the original ADIv5 Architecture Specification, which corresponds to issue A of this document. From issue B of this document, the specification integrates the descriptions of ADIv5.0, ADIv5.1, and ADIv5.2.

Features that are defined by ADIv5.0

ADIv5.0 defines:

- Two Debug Ports, JTAG-DP and SW-DP.
- Two Access Ports:
  - JTAG-AP, for accessing legacy JTAG components.
  - MEM-AP, for accessing memory and components with memory-mapped interfaces.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- The identification model for Access Ports.
- A discovery mechanism for components that are attached to a MEM-AP.

Features added in v5.1

ADIv5.1 formalizes version numbering of Debug Ports:

- The programmers' model of the JTAG-DP defined by ADIv5.0 is defined as *Debug Port architecture version 0* (DPv0).
- The programmers' model of the SW-DP defined by ADIv5.0 is defined as *Debug Port architecture version 1* (DPv1).

ADIv5.1 adds the following functionality:

- Extensions to the AP identification model.
- Standard definitions for MEM-AP implementations for AMBA bus protocols.
- JTAG support in Debug Port architecture version 1.
- The Minimal Debug Port extension.
- *Debug Port architecture version 2* (DPv2).
- Multiple protocol interoperability extensions that provide the following features:
  - Simple switching between Serial Wire Debug and JTAG protocols. For more information, see Chapter B5 *The Serial Wire/JTAG Debug Port*.
  - A dormant state, for interoperability with other protocols. For more information, see *Switching between SWD and JTAG* on page B5-128.
- Serial Wire Debug protocol version 2, that provides a multi-drop capability. For more information, see Chapter B4 *The Serial Wire Debug Port*.
- The *Minimal Debug Port* (MINDP) extension, which is a simplified version of the Debug Port that is intended for low gate-count implementations. For more information, see *MINDP, Minimal DP extension* on page B1-40.

Features added in ADIv5.2

ADIv5.2 adds the following functionality:

- Extensions to the MEM-AP programmers' model to support:
  - Large physical address spaces of up to 64 bits.
  - Data sizes greater than 32 bits.
  - Barrier operations.
- Armv8-A and Armv9-A definitions of MEM-AP to include extensions for:
  - AMBA AXI3, AXI4, AXI4-Lite, and AXI5.
  - AMBA ACE.
- Recommended implementations for:
  - Armv7-A processors including the Large Physical Address Extension.
  - Armv8-A processors.
- In addition to ROM Tables with a CIDR1.CLASS of `0x1`, ROM Tables with a CIDR1.CLASS of `0x9`, are permitted, provided their DEVID.FORMAT is `0x0`, which indicates that they use the 32-bit ROM format 0. For more information, see *ROM Table Types* on page D1-227.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In addition to using 4-bit JTAG IR instruction encodings, it is permitted to use 8-bit encodings. The 8-bit encodings are specified in *IR scan chain and IR instructions* on page B3-91.
- Relaxation of the CDBGPWRUPREQ/CDBGPWRUPACK handshake mechanism. For more information, see *DLCR, Data Link Control register* on page B2-61.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

A1.2 Purpose of the ADI

The ADI provides access to debug functionality that is provided by debug components in an embedded SoC.

This section summarizes various types of debug functionality that can be found in SoCs. It contains the following subsections:

- *Embedded core debug functionality*.
- *System debug functionality*.
- *Compatibility between CoreSight and Arm debug interfaces* on page A1-24.

For information about compatibility with the CoreSight™ architecture, see *Compatibility between CoreSight and Arm debug interfaces* on page A1-24.

A1.2.1 Embedded core debug functionality

An embedded microprocessor can provide the following debug features to enable the debugging of applications:

Processor state modification

Facilities that enable an external host to modify the state of the processor, as defined by the contents of the internal registers and the memory system.

Processor state assessment

Facilities that enable an external host to assess the state of the processor by providing access to the contents of the internal registers and the memory system.

Programming debug events

Facilities that allow an external host to program debug events. An external host must be able to configure the debug logic so that when a special event occurs, such as the program flow reaching a certain instruction in the code, the core enters a special execution mode in which its state can be examined and modified by an external system. In this chapter, this special execution mode is referred to as Debug state.

Enter or exit Debug state

Facilities to allow an external system to force the processor to enter or exit Debug state, and determine when the core enters or leaves Debug state.

Trace features

Trace the program flow that is associated with programmable events.

Examples of technologies that provide these facilities are:

- The Armv8 Debug Architecture. For more information, see the *Arm® Architecture Reference Manual, Armv8, for A-profile architecture*.
- The Embedded Trace Macrocell. For more information, see the *ETM Architecture Specification*.

ADIv5 implementations can also access legacy components that implement an IEEE 1149.1 JTAG interface, which enables accessing debug resources in processors that implement earlier versions of the ADI.

A1.2.2 System debug functionality

The scope of debug information extends beyond the boundaries of an embedded microprocessor core, and includes the following elements:

- Components outside the cores that are embedded in the SoC.
- The interconnection fabric of the system.

To enable debugging these elements, an SoC can provide the following system-level debug features:

External host access

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Facilities that enable an external host to access the following debug information:

- System state parameters that might not be visible to the embedded microprocessor core.
- Trace information about the interconnection fabric, for example accesses by the microprocessor core, or accesses by other devices such as *Direct Memory Access* (DMA) engines.

Access to diagnostic information

A mechanism for the efficient collection and streaming of diagnostic information, for example program trace.

Diagnostic messaging

Mechanisms for low-intrusion diagnostic messaging between software and debugger.

Cross-triggering

Cross-triggering mechanisms that enable debug components to signal to each other.

Examples of technologies that provide these facilities are:

- The ADIv5 *Debug Access Port* (DAP).
- The CoreSight debug architecture. For more information, see the *Arm® CoreSight™ Architecture Specification*.
- CoreSight components. For more information, see the *CoreSight™ SoC Technical Reference Manual.*

A1.2.3 Compatibility between CoreSight and Arm debug interfaces

ADIv5 is compatible with the Arm CoreSight architecture:

- ADIv5 can be used to access and control CoreSight-compatible components.
- The ADIv5 specification does not require debug components to comply with the CoreSight architecture.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

A1.3 The subdivisions of an ADIv5 implementation

An implementation of the ADI is called a DAP. A DAP provides a debugger with a standard interface to access debug resources in systems that use resource-specific methods to expose their debug information.

A1.3.1 Connections to the ADI

The logical block diagram in Figure A1-1 shows how an ADI implementation is connected between a debugger and the system to be debugged.

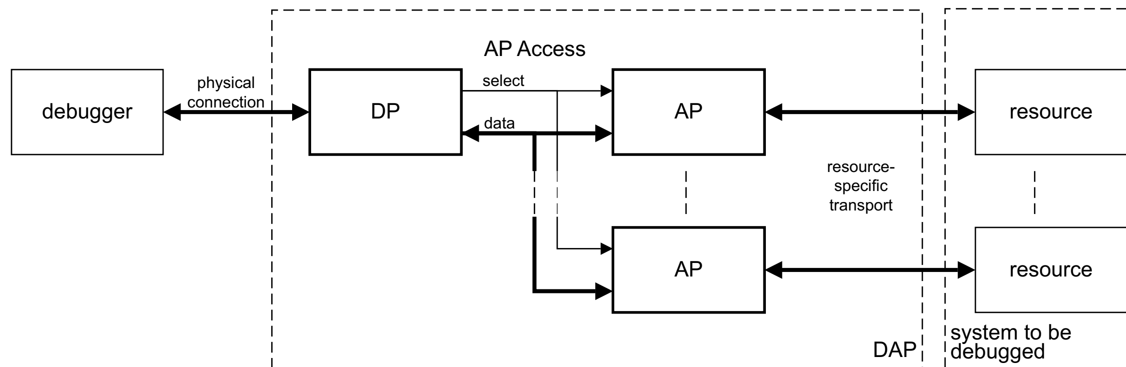


Figure A1-1 Block diagram of an ADIv5 implementation

To access a debug resource, the debugger passes the appropriate resource address information to the DAP. The DAP executes the request by selecting the appropriate resource and then accesses resource-specific transport methods that are presented by the system to be debugged. The DAP consists of the following elements:

Access Port (AP)

An AP uses a resource-specific transport mechanism to access debug information in the system to be debugged, and passes the information to the DP using the AP Access mechanism that is specified in this document. Examples of debug resources are:

- The debug registers of the core processor.
- ETM or trace port debug registers.
- A ROM Table, see Chapter D1 *About ROM Tables*.
- A memory system.
- A legacy JTAG device.

A debugger uses APACC accesses to exchange information held in the AP registers, as described in *Access Ports (APs)* on page A1-28

Note

A DAP must contain at least one AP, but if needed an ADI can implement multiple APs.

Debug Port (DP)

The DP provides a debugger with a common interface to access the information that is held in the APs. The DP includes the following elements:

- A physical connection to the debugger. ADIv5 supports the following physical connection types:
  - JTAG-DP.
  - SW-DP.
  - SWJ-DP.

  For details about the supported physical connections, see Chapter B1 *About the DP*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- DP registers, which hold information that is required to support the transport mechanism that is implemented by the DAP, as described in *Accessing the DAP*. A debugger uses the DPACC scan chain to exchange information held in the DP registers.

  For detailed information about the DP registers, see *DP register descriptions* on page B2-53.

Resource-specific transport

The connection between the DP and the APs performs the following tasks:

- Select the appropriate debug resource, based on the address information that was provided by the debugger.
- Transport the data between the APs and the DP.

A1.3.2 Accessing the DAP

The diagram in Figure A1-1 on page A1-25 shows how a debugger logically accesses the DP and AP registers.

- Although the DP is involved in responding to APACC requests, this involvement is transparent to the debugger at the level of the APACC.
- The debugger can use the DPACC method to access the DP registers, and achieve one of the following:
  - Set the parameters for an imminent APACC. For example, the selection of a particular AP is done by setting the DP register SELECT.
  - Read status information for a previous APACC. For example, the status of the sticky flags resulting from previous resource accesses is available from the DP register CTRL/STAT.

For details about the communication between the debugger and the DAP, see *The Debug Port (DP)* on page A1-27.

Note

Although this specification defines the ADIv5 in terms of the elements that are shown in Figure A1-1 on page A1-25, it is not mandatory to structure implementations in this way. The elements that are shown in the figure, however, provide a convenient representation for describing the programmers' model.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

A1.4 The Debug Port (DP)

An ADI implementation includes a single DP that provides the following features:

- An external physical connection to the interface. The signals that make up the physical connection depend on the DP type.
- A method to obtain the identification code of the DAP.
- DP and AP access methods, which depend on the DP type.
- A method to abort a register access that appears to have failed.

The ADIv5 specification supports the following DP types:

The JTAG Debug Port (JTAG-DP)

The JTAG-DP is accessed by IEEE 1149.1-compliant DBGTAP scan chains to read and write register information.

- For more information about DBGTAP scan chains, see Chapter B3 *The JTAG Debug Port*.
- *IEEE Standard 1149.1 Test Access Port and Boundary Scan Architecture* contains detailed information about the requirements for JTAG scan chains.

The Serial Wire Debug Port (SW-DP)

The SW-DP is a two-pin serial interface that uses a packet-based protocol to read or write registers. The protocol requires the following steps for communication between the host, which is the debugger, and the target, which is the ADI:

1. A host-to-target packet request, which includes whether the required access is to a DP register (DPACC) or to an AP register (APACC), and a two-bit register address.
2. A target-to-host acknowledge response.
3. A data transfer phase, if necessary. This phase can be target-to-host or host-to-target, depending on the request that is made in the first phase.

For details about the SW-DP protocol, see Chapter B4 *The Serial Wire Debug Port*.

The Serial Wire/JTAG Debug Port (SWJ-DP)

The SWJ-DP interface combines the SWD and JTAG Data Link protocols using the following mechanism:

- The pins that carry the signals are shared between the two options.
- The debugger can select which of the protocols it wants to use.

For details about how to implement the SWJ-DP, see Chapter B5 *The Serial Wire/JTAG Debug Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5 Access Ports (APs)

An AP uses a resource-specific transport mechanism to access debug information in the system to be debugged. The AP passes the information to the DP from where it can be accessed by a debugger using a standardized protocol over a standard physical connection.

The implementation of an AP depends on the resources it accesses. This specification includes programmers' models for following two types of resources:

- Memory-mapped resources, such as debug peripherals, for which ADIv5 defines a MEM-AP programmers' model. For a complete description of the MEM-AP programmers' model, see *Guide to the detailed description of a MEM-AP* on page A1-30.
- Legacy IEEE 1149.1 JTAG devices, for which ADIv5 defines a JTAG-AP and associated programmers' model. For a complete description of the JTAG-AP programmers' model, see *Guide to the detailed description of a JTAG-AP* on page A1-31.

Note

This specification does not specify exact requirements for the transport between the AP and the resource. In particular, it does not require a MEM-AP to use a bus to connect to the system being debugged. For example, ADIv5 might be directly integrated into the resource. In logical terms, however, a MEM-AP always accesses a memory-mapped resource in the system being debugged, which is why this specification describes MEM-AP accesses to the system being debugged as memory accesses.

In the future, more Arm APs might become available.

An ADI can include APs that are specified by companies other than Arm.

All APs must follow a base standard for identification, and debuggers must be able to recognize and ignore APs that they do not support. For more information, see Chapter C1 *About the AP*.

As described in *The subdivisions of an ADIv5 implementation* on page A1-25:

- The simplest ADI has only one AP. This AP can be either a MEM-AP or a JTAG-AP.
- ADIs can have multiple APs. For example:
  - A mixture of MEM-APs and JTAG-APs.
  - All MEM-APs.
  - All JTAG-APs.
- Debuggers must be able to recognize and ignore unsupported APs.

For more information, see Chapter C1 *About the AP*.

A1.5.1 Using the Debug Port to access Access Ports

Figure A1-2 on page A1-30 shows the different levels between the physical connection to the debugger and the debug resources of the system being debugged. These levels are designed to enable efficient access to the system being debugged, and several levels provide registers within the DAP. This section describes how these register accesses are implemented.

The DAP supports two types of accesses: DP accesses and AP accesses. Because debuggers usually have serial interfaces, the methods of making these accesses are kept as short as possible, and all accesses are 32-bits.

The description that is given here is of scan chain access to the registers, from a debugger that is connected to a JTAG Debug Port. However, the process is similar when the access is from an SWD interface connection to an SW-DP. Differences when accessing the registers from a Serial Wire Debug interface connection are described in Chapter B4 *The Serial Wire Debug Port*.

Every AP or DP access transaction from the debugger includes two address bits, A[3:2]:

- For a DP register access, the address bits A[3:2] and SELECT.DPBANKSEL determine which register is accessed.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- For an AP register access, SELECT.APSEL selects an AP to access, and the address bits A[3:2] are combined with SELECT.APBANKSEL to determine which AP register is accessed, as shown in Figure A1-2 on page A1-30. The two address bits A[3:2] are decoded to select one of the four 32-bit words from the register bank that is indicated by SELECT.APBANKSEL in the AP indicated by SELECT.APSEL.

Bits[1:0] of all AP and DP register addresses are `0b00`.

For example, to access the register at address `0x14` in the AP that is selected when SELECT.APSEL is `0x00`, the debugger must:

- Use a DP register write to set:
  - SELECT.APSEL to `0x00`.
  - SELECT.APBANKSEL to `0x1`.
- Use an AP register access with A[3:2] = `0b01`.

The DAP combines A[3:2] with SELECT.APBANKSEL to generate the AP register address, `0x14`. The debugger can access any of the four registers from `0x10` to `0x1C` without changing SELECT.

This access model is shown in Figure A1-2 on page A1-30. This figure shows how the contents of the SELECT register are combined with the A[3:2] bits of the APACC scan-chain to form the address of a register in an AP. Other parts of the JTAG-DP are also shown. These parts are explained in greater detail in later sections.

- Figure C2-1 on page C2-149, for a MEM-AP implementation.
- Figure C3-1 on page C3-191, for a JTAG-AP implementation.

These figures give more detail of the connections to the debug or system resources.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

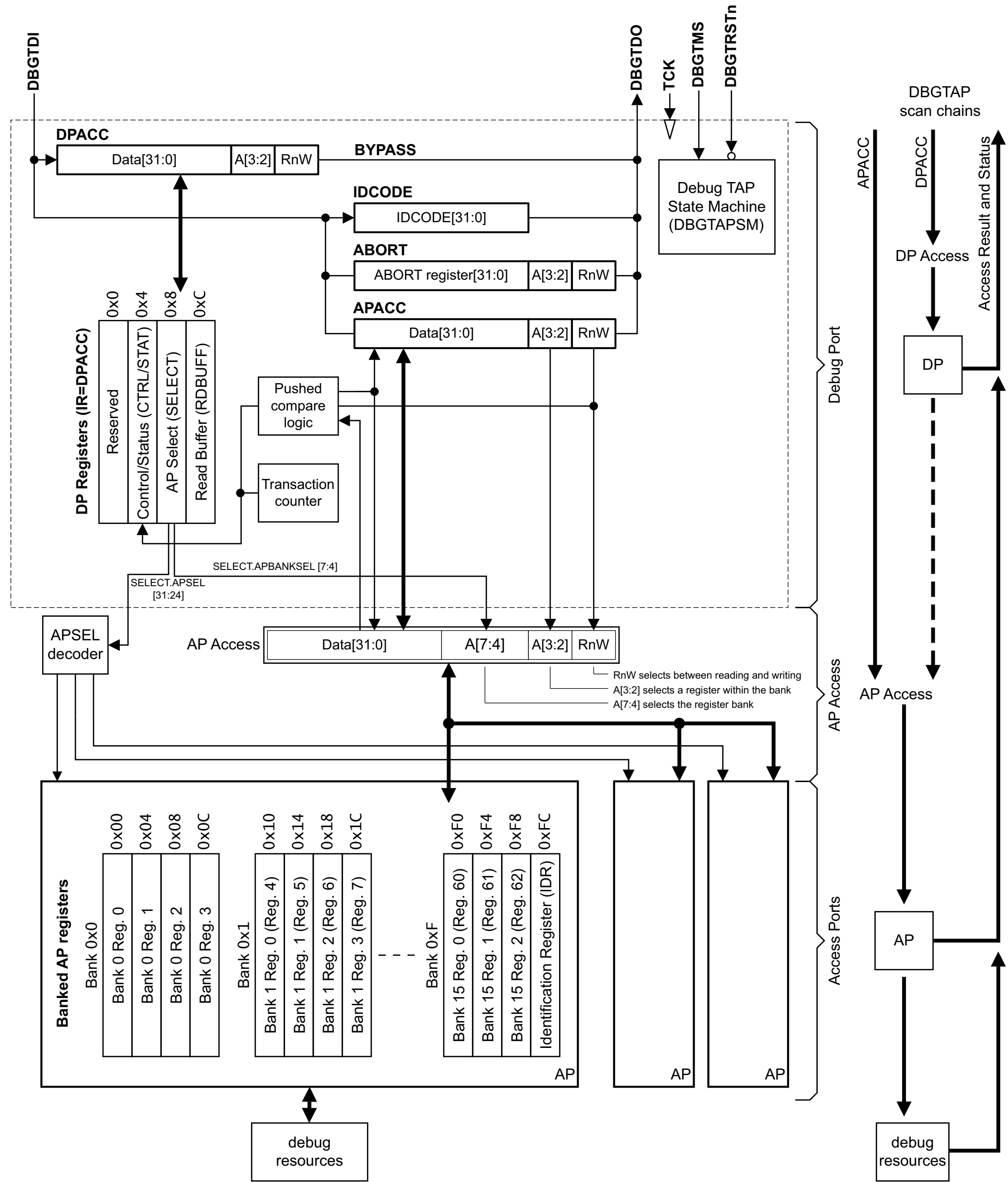


Figure A1-2 Structure of the DAP, showing DPv0 JTAG-DP accesses to a generic AP

A1.5.2 Guide to the detailed description of a MEM-AP

The operation and use of a MEM-AP must be understood within the content of interactions with all of the following components:

- The MEM-AP itself.
- The MEM-AP registers.
- The standard debug components registers that you access through the MEM-AP.

The MEM-AP is described in the following chapters of this specification:

- Chapter C1 *About the AP*.
- Chapter C2 *The Memory Access Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

The MEM-AP provides access to zero, one, or more debug components. Any debug component that complies with the Arm Generic Identification Registers specification implements a set of Component Identification Registers. These registers are described in the *Arm® CoreSight™ Architecture Specification*.

If the MEM-AP connects to more than one debug component, the system that is accessed by the MEM-AP must also include at least one ROM Table. ROM Tables are accessed through a MEM-AP and are described in Chapter D1 *About ROM Tables*.

Note

As shown in *Design choices and implementation examples* on page A1-32, a system with only one functional debug component might also implement a ROM Table.

A1.5.3 Guide to the detailed description of a JTAG-AP

To understand the operation and use of a JTAG-AP, you must understand:

- The JTAG-AP itself.
- The JTAG-AP registers.

The JTAG-AP is described in the following chapters of this specification:

- Chapter C1 *About the AP*.
- Chapter C3 *The JTAG Access Port*.

The JTAG-AP provides a standard JTAG connection to one or more legacy components. The connection between the JTAG-AP and the components is described by the *IEEE 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture*. Details on how to use of this connection are outside the scope of this specification.

A1.5.4 Using the AP to access debug resources

Accessing the AP gives access to the system being debugged, which is shown as access to *Debug resources* in Figure A1-2 on page A1-30.

In summary:

- In a MEM-AP, the debug resources are logically memory-mapped. Chapter C2 *The Memory Access Port* section *MEM-AP register accesses and memory accesses* on page C2-150 describes the method for accessing these resources. The connection between the MEM-AP and a debug resource, however, is outside the scope of this specification.
- In a JTAG-AP, the debug resources are connected through a standard JTAG serial connection, as defined in *IEEE 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture*. For more information about accessing the resources, see Chapter C3 *The JTAG Access Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

A1.6 Design choices and implementation examples

Figure A1-1 on page A1-25 introduces the components that comprise an ADI.

Before implementing an ADI, certain design choices must be made, as described in this section regarding the following functional blocks of the interface:

- Choices for the DP.
- Choices for the APs.

Note

This specification is written for engineers implementing an Arm Debug Interface, and for engineers using an Arm Debug Interface. The design choices outlined in this specification for a debug interface have an implicit purpose. If the reasoning behind the design choices are not explicit, the implementer of the debug interface must be contacted for further information.

A1.6.1 Choices for the DP

The DP determines which type of physical connection the ADI presents to the debugger. A DAP has only one DP, so your choice for the DP type decides the physical connection for the entire design. You can choose from the following DP types:

- JTAG-DP.
- SW-DP.
- SWJ-DP.

Note

Note the following with respect to the DP types mentioned in this document:

- In an illustration of an ADI, a component that is labeled DP can represent any of the available options.
- Arm might define more DP types in the future.

A1.6.2 Choices for the APs

A single ADI uses a single AP to connect to a single debug component, for example:

- A MEM-AP that connects to a single microprocessor core, as shown in Figure A1-3.
- A JTAG-AP that connects to a single legacy IEEE 1149.1 device, as shown in Figure A1-4 on page A1-33.

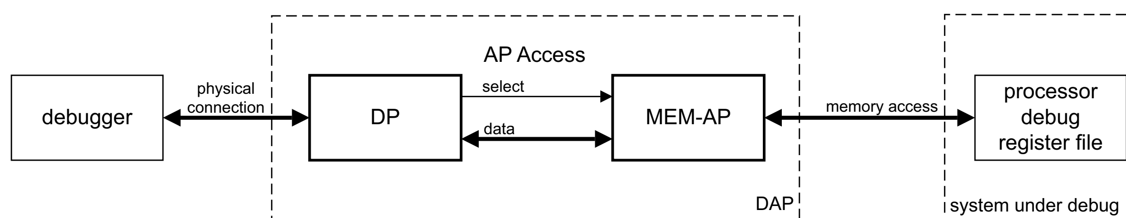


Figure A1-3 Simple ADI MEM-AP Implementation

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

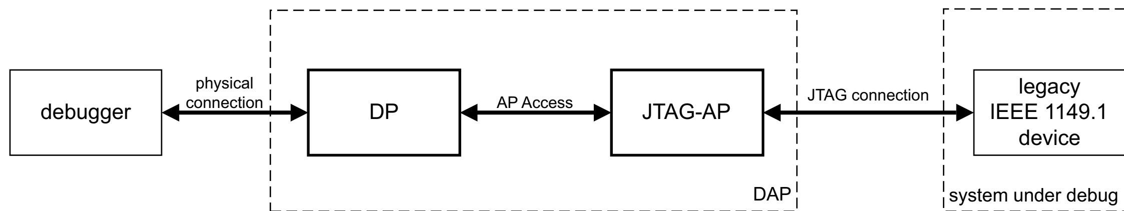


Figure A1-4 Simple ADI JTAG-AP Implementation

A system with only a single debug component often implements a ROM table, as explained in *ROM Tables* on page C2-148 and shown in Figure A1-5.

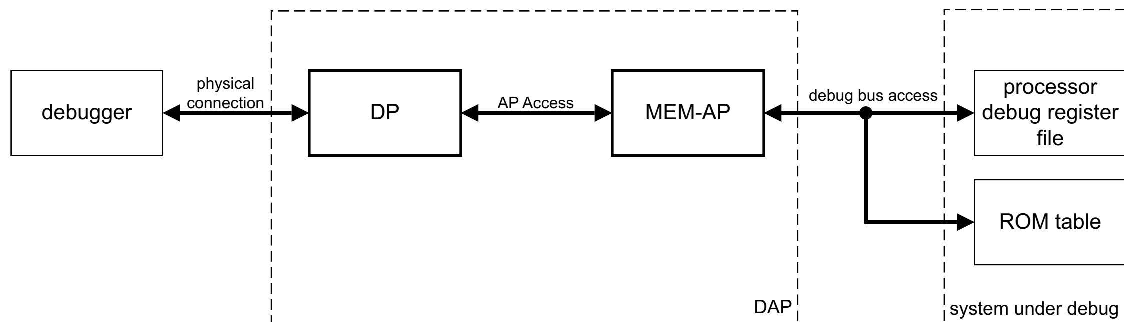


Figure A1-5 Simple example of an ADI implementation, with ROM Table

Because a single ADI can include multiple APs, design choices for APs must be made at two levels:

- Choosing the number of APs in the ADI, and whether each AP is a MEM-AP or a JTAG-AP. These decisions are outlined in *Top-level AP planning choices*.
- The choices that have to be made for each implemented AP, as outlined in the following sections:
  - *Choices for JTAG-APs* on page A1-35.
  - *Choices for MEM-APs* on page A1-35.

Top-level AP planning choices

In a more complex system, there can be multiple APs. Each AP can be connected to multiple components, or multiple address spaces. An AP can be implemented as one of the following three types:

- As a MEM-AP with a memory-mapped debug bus connection. The debug bus connects directly to one or more debug register files.
- As a MEM-AP with a memory-mapped system bus connection. The MEM-AP connection to the system bus provides access to one or more debug register files.
- As a JTAG-AP. A JTAG-AP connects directly to one or more JTAG devices, and enables connection to legacy hardware components.

Note

The connection between legacy hardware components and a JTAG-AP is defined by the JTAG standard. For more information, see Chapter C3 *The JTAG Access Port*.

If you are designing or specifying an ADI, it must be decided how many APs and of which types are required.This depends largely on the debug components of the system to which your ADI connects.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Figure A1-6 shows a more complex ADI and illustrates the different AP types.

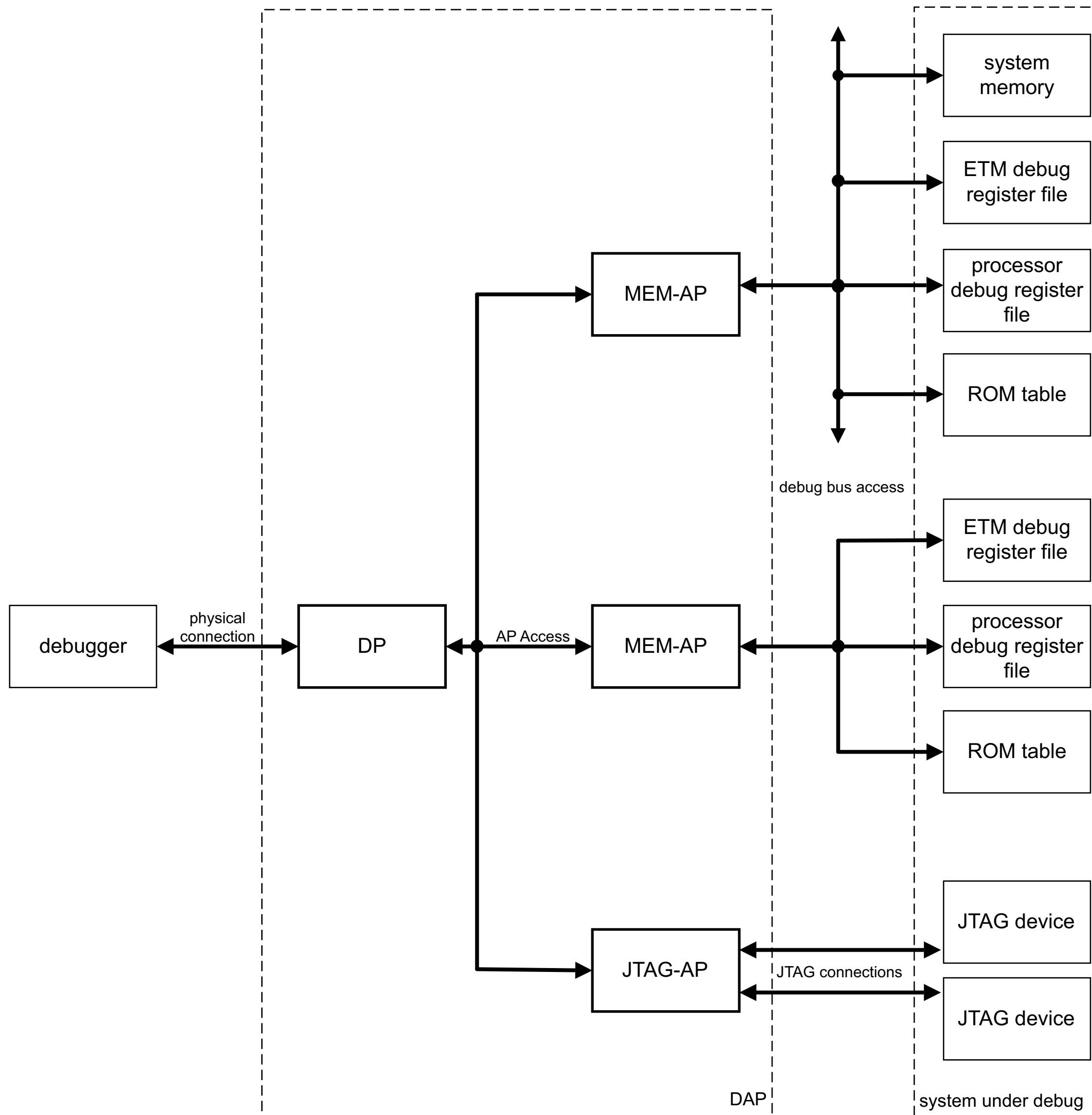


Figure A1-6 Complex Arm Debug Interface that uses several AP types

The ADIv5 architecture specification supports the following features:

- A DAP is permitted to contain multiple APs.
- A single MEM-AP is permitted to access multiple register files.
- An AP is permitted to access a mixture of system memory and debug register files.

When implementing these features, however, you must observe the following conditions:

- Every AP must follow the basic standard for identification that is described this specification.
- Debuggers must have a way to ignore APs that they do not recognize.

In illustrations such as Figure A1-2 on page A1-30, the DP can be of any DP type that is defined by ADIv5.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Choices for MEM-APs

The following design decisions must be made for a MEM-AP:

Decisions that depend on the requirements of those debug components

The main decisions to be made for a MEM-AP concern the connection between the MEM-AP and the memory-mapped debug components that are connected to it. These decisions include:

- Whether a bus is required for this connection.
- The width of the bus, if implemented.
- The memory map of the MEM-AP address space.

Inclusion of a ROM Table

If a MEM-AP connects to more than one debug component, the system must include one or more ROM Tables to provide information about the debug system. A system that has only one other component does not require a ROM Table, but a system designer might choose to include one anyway. For more information, see *ROM Tables* on page C2-148.

The inclusion of IMPLEMENTATION DEFINED MEM-AP features

- Certain features must be included if the connection is less than 32-bits wide.
- The debug components can place limitations on the connection, for example a component might require 32-bit access.

For more information about conditional MEM-AP features, see:

- *MEM-AP functions* on page C2-152.
- *MEM-AP implementation requirements* on page C2-162.

For detailed information about implementing a MEM-AP, see Chapter C2 *The Memory Access Port*.

Choices for JTAG-APs

The following design decisions must be made for a JTAG-AP:

The number of JTAG scan chains that are connected to the JTAG-AP

A single JTAG-AP can connect to up to eight JTAG scan chains. These scan chains can be split across multiple devices or components within the system being debugged.

The number of TAPs in each scan chain

A single JTAG scan chain can contain multiple *Test Access Ports* (TAPs). However, Arm recommends that each scan chain connected to a JTAG-AP contains only one TAP.

For more information, see Chapter C3 *The JTAG Access Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Part B

The Debug Port

Chapter B1
About the DP

This part describes the features that are implemented by the DP.

A DP can be implemented as a JTAG Debug Port (JTAG-DP), a Serial Wire Debug Port (SW-DP), or a combined Serial Wire/JTAG Debug Port (SWJ-DP).

Requirements that apply to all DP types are described in the following sections in this chapter:

- *MINDP, Minimal DP extension* on page B1-40.
- *Sticky flags and DP error responses* on page B1-41.
- *The transaction counter* on page B1-43.
- *Pushed-compare and pushed-verify operations* on page B1-44.
- *Power and reset control* on page B1-46.

Reference information for all DP types is described in the following chapter:

- Chapter B2 *DP Reference Information*.

Specific information for each of the DP types is described in the following chapters:

- Chapter B3 *The JTAG Debug Port*.
- Chapter B4 *The Serial Wire Debug Port*.
- Chapter B5 *The Serial Wire/JTAG Debug Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.1 MINDP, Minimal DP extension

The MINDP programmers' model is a simplified version of the DP that is intended for low gate-count implementations. MINDP implementations must use DPv1 or later.

MINDP implementations must omit the following DP features:

- Pushed-verify operation.
- Pushed-compare operation.
- The transaction counter.

MINDP implementations must observe the following conventions:

- The DPIDR.MIN field is RAO.
- The following fields of the CTRL/STAT register are RES0:
  - TRNCNT.
  - MASKLANE.
  - STICKYCMP.
  - TRNMODE.

  See also CTRL/STAT.
- The ABORT.STKCMPCLR field is SBZ. Writing `0b1` to this bit is UNPREDICTABLE.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B1.2 Sticky flags and DP error responses

Sticky flags signal transaction errors and are persistent between transactions. When set, a sticky flag remains set until the debugger actively clears it, even if the condition that caused the flag to be set no longer applies.

In the CTRL/STAT register, the sticky error flags are:

- STICKYERR, bit[5].
- STICKYCMP, bit [4].
- STICKYORUN, bit[1].
- WDATAERR, bit[7], SW-DP only.

After performing a series of APACC transactions, a debugger must check the CTRL/STAT register to check if an error occurred. If the debugger finds that a sticky flag is set, it clears the flag, and, if necessary, initiates extra APACC transactions to determine why the sticky flag was set. Because the flags are sticky, the debugger does not have to check the flags after every transaction, and must only check the CTRL/STAT register periodically, which reduces the overhead of checking for errors.

When an error is flagged, the current transaction is completed and subsequent APACC transactions are discarded until the sticky flag is cleared.

The DP response to an error condition and the method to clear the sticky flags depends on the DP type:

- An SW-DP immediately signals an error response.
- A JTAG-DP immediately discards all transaction and marks them as complete.

For details on how to clear the sticky flags for each DP type, see the descriptions of the sticky flag fields in *CTRL/STAT, Control/Status register* on page B2-55.

If pushed transactions are supported, the sticky flag CTRL/STAT.STICKYCMP reports the result of pushed operations, see *Pushed-compare and pushed-verify operations* on page B1-44. CTRL/STAT.STICKYCMP behaves in the same way as the sticky flags described in this section.

The DP uses the sticky flags in the CTRL/STAT register to signal the following transaction errors:

Read and write errors

A read or write error can occur in the DAP or in the resource being accessed. In either case, when the error is detected, the Sticky Error flag CTRL/STAT.STICKYERR is set to `0b1`.

For example, a read or write error might occur if the debugger makes an AP transaction request while the debug power domain is powered down. See *Power and reset control* on page B1-46 for information about power domains.

Overrun detection

DPs support an overrun detection mode, which enables a debugger to send blocks of commands using a connection with high latency and high throughput. These commands must be sent with sufficient in-line delays to make overrun errors unlikely. To implement an overrun detection mode, the DAP can be programmed to set the Sticky Overrun flag, CTRL/STAT.STICKYORUN, to `0b1` if an overrun error occurs. In overrun detection mode, the debugger must check the Sticky Overrun flag for overrun errors after each sequence of APACC transactions.

Overrun detection mode is enabled by setting the Overrun Detect bit, CTRL/STAT.ORUNDETECT, to `0b1`.

Due to the differences between the JTAG-DP and the SW-DP, their behavior in overrun detection mode is DATA LINK DEFINED:

JTAG-DP If the response to any transaction is not OK/FAULT, the Sticky Overrun flag, CTRL/STAT.STICKYORUN, is set to `0b1`.

The response to a transaction is WAIT until the previous AP transaction is complete. Subsequent responses are OK/FAULT. See *Sticky overrun behavior on DPACC and APACC accesses* on page B3-100.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

SW-DP If the response to any transaction is not OK, the Sticky Overrun flag, CTRL/STAT. STICKYORUN, is set to `0b1`.

If a previous AP transaction is incomplete, the first response to a transaction is WAIT. Subsequent responses are FAULT, because the STICKYORUN flag is `0b1`. See *Sticky overrun behavior* on page B4-115.

The value of the Sticky Error flag, CTRL/STAT.STICKYERR, is not changed.

Note

The method for clearing the STICKYORUN flag depends on whether the DP type is SW-DP or JTAG-DP. See the descriptions of the STICKYORUN field in *CTRL/STAT, Control/Status register* on page B2-55 for more information.

If a new transaction results in an overrun error while an earlier transaction is incomplete, the earlier transaction completes normally. Other sticky flags, however, might be set to `0b1` during completion of the earlier transaction.

If the debugger clears the ORUNDETECT flag while STICKYORUN is `0b1`, the resulting value of STICKYORUN is UNKNOWN.

To leave overrun detection mode, a debugger must perform the following steps:

1. Check the value of the CTRL/STAT.STICKYORUN flag.
2. If the STICKYORUN flag is `0b1`, clear it to `0b0`.
3. To disable overrun detection mode, clear the ORUNDETECT flag to `0b0`.

Protocol errors (SW-DP only)

The SW-DP can generate protocol errors, for example in the case of wire-level errors.

Note

Although protocol errors can only occur in the SW-DP, they are described in this chapter because they are part of the sticky flags error-handling mechanism.

The required response is as follows:

- If the SW-DP detects a protocol error in a packet request, the DP does not respond to the message.
- If the SW-DP detects a parity error in the data phase of a write transaction, it sets the Sticky Write Data Error flag, CTRL/STAT.WDATAERR. The Sticky Write Data Error flag is treated in the same way as the other sticky flags described in this section.

For more information, see *Parity* on page B4-108 and *Protocol error response* on page B4-114.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B1.3 The transaction counter

Except for MINDP implementations, DPs must include an AP transaction counter, CTRL/STAT.TRNCNT. The transaction counter enables a debugger to generate a sequence of AP transactions with a single AP transaction request. With a MEM-AP access, the transaction counter enables an AP transaction to generate a sequence of accesses to the connected memory system.

Note

Each AP defines which registers support sequences of transactions. If an AP register does not support sequences of transactions, or SELECT.APSEL selects an AP that is not present, then the result of a sequence of transactions to that register is UNPREDICTABLE. Reserved AP registers and the common AP IDR do not support sequences of transactions.

Examples of the use of the transaction counter are:

Memory fill operations

To facilitate memory fill operations, the transaction counter can repeatedly write a single data value that is supplied in an AP transaction request. The MEM-AP includes a mechanism that initiates a series of AP accesses and automatically increments the access address after each AP access. This mechanism results in the supplied data value being written to a sequence of memory addresses under the control of the transaction counter. For more information, see *Packed transfers* on page C2-158.

Fast searches and memory verification

To perform a fast search, or verify of an area of memory, the transaction counter can be used when reading from the DRW register, with pushed-compare or pushed-verify operations enabled. For examples of this application, see *Pushed-compare and pushed-verify operations* on page B1-44, and, for more details, *Example of using the transaction counter for a pushed-compare operation on a MEM-AP* on page C2-166.

Writing a value other than zero to the CTRL/STAT.TRNCNT field generates multiple AP transactions. For example, writing `0x001` to this field generates two AP transactions, and writing `0x002` generates three transactions.

If the transaction counter is not zero, it is decremented after each successful transaction. If one of the following is true, the transaction counter is not decremented and the transaction is not repeated:

- The transaction counter is zero.
- The CTRL/STAT.STICKYERR flag is `0b1`.
- The CTRL/STAT.STICKYCMP flag is `0b1`.

If a sequence of operations is terminated because the Sticky Error or Sticky Compare flag was set to `0b1`, the transaction counter remains at the value from the last successful transaction, which enables the software to recover the location of the error, or determine where the compare or verify operation terminated.

The transaction counter does not automatically reload when it reaches zero.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4 Pushed-compare and pushed-verify operations

The DP supports pushed operations. Pushed operations improve performance where writes might be faster than reads. They are used as part of in-line tests, for example Flash ROM programming and monitor communication.

Pushed operations use the following mechanism:

1. The debugger initiates an AP write transaction. The value to be written is stored in the DP.
2. The DP reads a value from the AP.

   Note

   Whenever an AP write transaction is performed with pushed-compare or pushed-verify enabled, the AP access that results is a read operation, not a write.

3. The DP then compares the two values and updates the Sticky Compare flag, CTRL/STAT.STICKYCMP, based on the result of the comparison. Whenever the STICKYCMP bit is set to `0b1` in this way, any outstanding transaction repeats are canceled.

Pushed operations can affect AP behavior:

- Performing an AP read transaction with pushed-compare or pushed-verify enabled causes UNPREDICTABLE behavior.
- If an SW-DP performs an AP read transaction with pushed-compare or pushed-verify, an UNKNOWN value is returned, and the read has UNPREDICTABLE side effects, even though the wire-level protocol remains coherent.
- Each AP defines which registers support pushed transactions. If an AP register does not support pushed transactions, or SELECT.APSEL selects an AP that is not present, a pushed transaction sets STICKYCMP to an UNKNOWN value. Reserved AP registers and the common AP IDR do not support pushed transactions.

To configure pushed operations, use the CTRL/STAT register:

1. Enable the appropriate transfer mode using the Transfer Mode field, TRNMODE:
   - A value of `0b01` in TRNMODE selects pushed-verify operations: if the values match, the STICKYCMP flag is set to `0b1`.
   - A value of `0b10` in TRNMODE selects pushed-compare operations: if the values do not match, the STICKYCMP flag is set to `0b1`.
2. Select the byte lanes to be included in the comparison using the byte lane mask field, MASKLANE. A value of `0b1` for bit *n* of MASKLANE includes byte *n* of the APACC write value and the current AP value in the comparison. For details about the MASKLANE field, see *CTRL/STAT, Control/Status register* on page B2-55.

The following are examples of applications of pushed-verify and pushed-compare MEM-AP operations:

- Pushed-verify can be used to verify the contents of system memory. A series of expected values are written as AP transactions. With each write, the pushed-verify logic initiates an AP read access, and compares the result of this access with the expected value. If the values do not match, the CTRL/STAT.STICKYCMP flag is set to `0b1`. This operation is described in more detail in *Example of using a pushed-verify operation on a MEM-AP* on page C2-165.
- Pushed-compare can be used to search system memory for a given value. However, this feature is most useful when it is performed using the AP transaction counter, which is described in *The transaction counter* on page B1-43. This operation is described in more detail in Chapter C2 *The Memory Access Port* section *Example of using the transaction counter for a pushed-compare operation on a MEM-AP* on page C2-166.

The following example describes pushed operations on a specific AP, which makes it easier to understand how pushed operations are implemented. Consider an AP write transaction to the Data Read/Write (DRW) register in a MEM-AP with a TRNMODE value of `0b10`, and a MASKLANE value of `0b0101`. The following actions take place:

1. The DP holds the data value from the AP write transaction in the pushed-compare logic, see Figure A1-2 on page A1-30.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

2. The AP reads from the address indicated by the MEM-AP *Transfer Address Register* (TAR).
3. The value that is returned by this read is compared with the value held in the pushed-compare logic. The comparison is masked using the value of MASKLANE. The example value, `0b0101`, includes byte lanes zero and two in the comparison. The result is either a match or a mismatch.
4. In the example, the TRNMODE value of `0b10` selects pushed-compare operations:
    - If the result of the comparison was a mismatch, the CTRL/STAT.STICKYCMP flag is set to `0b1` and any outstanding transactions are canceled.
    - If the result of the comparison was a match, nothing happens.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B1.5 Power and reset control

The DP supports the following power and reset control fields in the CTRL/STAT register:

- Control fields for system and debug power control, CDBGPWRUPREQ, CDBGPWRUPACK, CSYSPWRUPREQ, and CSYSPWRUPACK. For more information, see *System and debug power control behavior* on page B2-77.
- Control fields for debug reset control, CDBGRSTREQ and CDBGRSTACK. For more information, see *Debug reset control behavior* on page B2-82.

These control bits are programmable by the debugger, and drive signals into the target system.

The DP does not provide any control bits for requesting a system reset. However, it is common for the physical interface to the debugger to include a system reset pin, **nSRST**, which is intended to provide requests or stimuli into existing power and reset controllers. For details about how to implement a system reset pin, see *System reset control behavior* on page B2-84.

ADI does not replace the system power and reset controllers. This specification does not place any requirements on the operation of system power and reset controllers.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B2
DP Reference Information

This chapter contains the following reference information for the DP:

- *DP architecture versions* on page B2-48.
- *DP register descriptions* on page B2-53.
- *System and debug power control behavior* on page B2-77.
- *Debug reset control behavior* on page B2-82.
- *System reset control behavior* on page B2-84.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.1 DP architecture versions

This section introduces the concept of DP architecture versions and describes the DP registers for the DP architecture versions DPv0, DPv1, and DPv2. It contains the following subsections:

- *DP architecture versions summary*.
- *DP architecture version 0 (DPv0) address map* on page B2-49.
- *DP architecture version 1 (DPv1) address map* on page B2-50.
- *DP architecture version 2 (DPv2) address map* on page B2-51.

One of the significant differences between the JTAG-DP and the SW-DP is how the registers are accessed. For this reason, the tables that describe the registers do not include register address information. This information is included at the start of the detailed description of each register for each DP type.

Several aspects of the DP architecture are DATA LINK DEFINED, and described in the following chapters:

- Chapter B3 *The JTAG Debug Port*.
- Chapter B4 *The Serial Wire Debug Port*.
- Chapter B5 *The Serial Wire/JTAG Debug Port*.

B2.1.1 DP architecture versions summary

Every ADI includes a single DP that is compliant with one of the DP architecture versions. Table B2-1 shows the DP architecture versions.

Table B2-1 DP architecture versions

Version number	Description	Debug Port Support	Notes
DPv0	DP architecture version 0	JTAG-DP	JTAG-DP in ADIv5.0
DPv1	DP architecture version 1	SW-DP, JTAG-DP	SW-DP in ADIv5.0
DPv2	DP architecture version 2	SW-DP, JTAG-DP	SW-DP version 2 in ADIv5.1

Although the DP architecture versions are different, their register sets are similar, as summarized in Table B2-2. For details about how the register is implemented in a specific architecture version, and if the implementation is DATA LINK DEFINED, see *DP register descriptions*.

Table B2-2 Summary of DP registers

Name	DP architecture version		
	DPv0	DPv1	DPv2
ABORT	Yes	Yes	Yes
DPIDR	No	Yes	Yes
CTRL/STAT	Yes	Yes	Yes
SELECT	Yes	Yes	Yes
RDBUFF	Yes	Yes	Yes
DLCR	No	Yes	Yes
RESEND	No	Yes	Yes

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table B2-2 Summary of DP registers (continued)

Name	DP architecture version		
	DPv0	DPv1	DPv2
TARGETID	No	No	Yes
DLPIDR	No	No	Yes
TARGETSEL	No	No	Yes

B2.1.2 DP architecture version 0 (DPv0) address map

DPv0 supports JTAG-DP in ADIv5.

The JTAG-DP register accessed depends on both:

- The *Instruction Register* (IR) value for the DAP access.
- A[3:2] from the address field of the DAP access.

For more information, see *Accessing the JTAG-DP registers*.

Table B2-3 shows the DPv0 register map. The A[3:2] field of the DPACC scan chain provides bits[3:2] of the address. Bits[1:0] of the address are always `0b00`.

Table B2-3 DPv0 register map

Address	Name	Access
`0x0`[a]	-	-
`0x4`	CTRL/STAT	RW
`0x8`	SELECT	RW
`0xC`	RDBUFF	R
	-	W[a]

a. Reserved, UNPREDICTABLE.

The DP must implement the ABORT register. How this register is accessed is DATA LINK DEFINED. In JTAG-DP, the register is implemented through the ABORT instruction.

Accessing the JTAG-DP registers

The JTAG-DP registers are only accessed when the IR for the DAP access contains the DPACC or ABORT instruction. The register accesses for each instruction are:

DPACC The DPACC scan chain accesses the DP CTRL/STAT, SELECT, and RDBUFF registers at addresses `0x0` to `0xC`, although register address `0x0` is reserved, and the RDBUFF register at `0xC` is always RAZ/WI on a JTAG-DP.

These registers are shown in the illustration of the JTAG-DP in Figure A1-2 on page A1-30.

ABORT For a write access with address `0x0`, the ABORT scan chain accesses the ABORT register.

For a read access with address `0x0`, and for any access with address `0x4` to `0xC`, the behavior of the ABORT scan chain is UNPREDICTABLE.

For more information about the JTAG-DP scan chains, see Chapter B3 *The JTAG Debug Port*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.1.3 DP architecture version 1 (DPv1) address map

DPv1 extends DPv0 by adding support for SWD protocol version 1 and defining the following extra registers:

- The Debug Port Identification Register, DPIDR.
- The Data Link Control Register, DLCR.
- More DATA LINK DEFINED registers.

In addition, the definition of some of the DPv0 registers is changed:

- The behavior of writes to bits[4:1] of the ABORT register is defined.
- The behavior on writing to bits[5:4, 1] of the CTRL/STAT register is DATA LINK DEFINED.
- The SELECT register is write only.

For most register addresses, different registers are addressed on read and write accesses. In addition, the SELECT.DPBANKSEL bit determines which register is accessed at address `0x04`.

Table B2-4 shows the DPv1 register map.

Table B2-4 DPv1 register map

Address[a]	DPBANKSEL[b]	Name	Access	Notes
`0x0`	x	DPIDR	RO	-
		-	WO	DATA LINK DEFINED, as either: • ABORT • Reserved, UNPREDICTABLE
`0x4`	`0x0`	CTRL/STAT	RW	-
	`0x1`	DLCR	RW	-
	`0x2` to `0xF`	-	-	UNPREDICTABLE
`0x8`	x	-	RO	DATA LINK DEFINED
		SELECT	WO	-
`0xC`	x	RDBUFF	RO	-
		-	WO	DATA LINK DEFINED

a. Bits [1:0] of the address are always `0b00`.
b. SELECT.DPBANKSEL.

The DP must implement the ABORT register. How this register is accessed is DATA LINK DEFINED, and:

- If defined by the data link, DP register 0 is reserved for this purpose.
- In a JTAG-DP, this register is implemented through the ABORT instruction.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

SW-DP DATA LINK DEFINED registers, DPv1

Table B2-5 shows the DATA LINK DEFINED SW-DP registers for a DPv1 implementation. Bits[1:0] of the address are always `0b00`.

Table B2-5 SW-DP data link defined registers, DPv1

Address	Name	Access
`0x0`	ABORT	WO
`0x8`	RESEND	RO
`0xC`[a]	-	WO

a. Reserved, SBZ.

For a DPv1 JTAG-DP, all DATA LINK DEFINED registers are reserved. Accesses to a reserved DATA LINK DEFINED register are UNPREDICTABLE.

B2.1.4 DP architecture version 2 (DPv2) address map

DPv2 extends DPv1 with support for SWD protocol version 2 and definitions for the following registers:

- The Target Identifier register, TARGETID.
- The Data Link Protocol Identification Register, DLPIDR.
- The DATA LINK DEFINED Target Selection register, TARGETSEL.
- The EVENT Status register, EVENTSTAT.

For most register addresses, different registers are addressed on read and write accesses. In addition, an extended SELECT.DPBANKSEL field determines which register is accessed at address `0x04`.

Table B2-6 shows the DPv2 register map.

Table B2-6 DPv2 address map

Address[a]	DPBANKSEL[b]	Name	Access	Notes
`0x0`	x	DPIDR	RO	-
		-	WO	DATA LINK DEFINED, as either: • ABORT • Reserved, RES0
`0x4`	`0x0`	CTRL/STAT	RW	-
	`0x1`	DLCR	RW	-
	`0x2`	TARGETID	RO	-
	`0x3`	DLPIDR	RO	-
	`0x4`	EVENTSTAT	RO	-
	All other values	-	-	Reserved, RES0
`0x8`	x	-	RO	DATA LINK DEFINED
		SELECT	WO	-

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-6 DPv2 address map (continued)

Address[a]	DPBANKSEL[b]	Name	Access	Notes
0xC	x	RDBUFF	RO	-
		-	WO	DATA LINK DEFINED

a. Bits [1:0] of the address are always 0b00.
b. SELECT.DPBANKSEL field.

The DP must implement the ABORT register. How this register is accessed is DATA LINK DEFINED, and:

- If defined by the data link, DP register 0 is reserved for this purpose.
- In JTAG-DP, the ABORT register is implemented through the ABORT instruction.

SW-DP DATA LINK DEFINED registers, DPv2

Table B2-7 shows the DATA LINK DEFINED SW-DP registers for a DPv2 implementation:

Table B2-7 SW-DP data link defined registers, DPv2

Address[a]	SWD protocol version	Name	Access	See also:
0x0	x	ABORT	WO	-
0x8	x	RESEND	RO	-
0xC	v1	-	WO	Reserved, SBZ
	v2	TARGETSEL	WO	-

a. Bits [1:0] of the address are always 0b00.

For a DPv2 JTAG-DP, all DATA LINK DEFINED registers are RES0.

B2.1.5 Register maps, and accesses to reserved addresses

The register memory maps for the DP and the AP within the DAP are shown in:

- Figure A1-2 on page A1-30, for accesses to JTAG-DP registers.
- Figure C2-1 on page C2-149, for accesses to MEM-AP registers.
- Figure C3-1 on page C3-191, for accesses to JTAG-AP registers.

There are several reserved addresses in these register maps. Reserved AP registers are RES0.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2 DP register descriptions

This section gives full descriptions of the DP registers.

The registers are listed alphabetically by name.

B2.2.1 ABORT, Abort register

The ABORT characteristics are:

Purpose

ABORT forces an AP transaction abort.

From a software perspective, an abort is a fatal operation. It discards any outstanding and pending transactions, and leaves the AP in an UNPREDICTABLE state. On an SW-DP, however, the sticky error bits are not cleared to `0b0`.

Writing `0b1` to the ABORT.DAPABORT register bit generates a DAP abort, causing the current AP transaction to abort. This action also terminates the transaction counter, if it was active. It is IMPLEMENTATION DEFINED whether the AP propagates the abort, for example by aborting a transaction in progress.

After a DAP abort:

- It is IMPLEMENTATION DEFINED which registers, if any, in the AP that was aborted can be accessed. If the register cannot be accessed, the DP returns a WAIT response to an AP access to the register. Arm recommends that any AP register that is not directly related to a stalling transaction is accessible, to allow a debugger to diagnose the cause of the error.
- A DP access or an AP access to any other AP are accepted by the DP. This includes AP accesses to non-existent APs, which are defined to behave as RAZ/WI.

Caution

Use this function only in extreme cases, when debug host software has observed stalled target hardware for an extended period. Stalled target hardware is indicated by repeated WAIT responses.

Note

In DPv1 and DPv2 only, the ABORT register has extra fields that clear error and sticky flag conditions. See the descriptions of the flag fields in *CTRL/STAT, Control/Status register* on page B2-55. In DPv0, these fields are reserved, SBZ.

Usage Constraints

ABORT is accessible as follows:

Default
WO

Configurations

ABORT is defined and implemented in DPv0, DPv1, and DPv2.

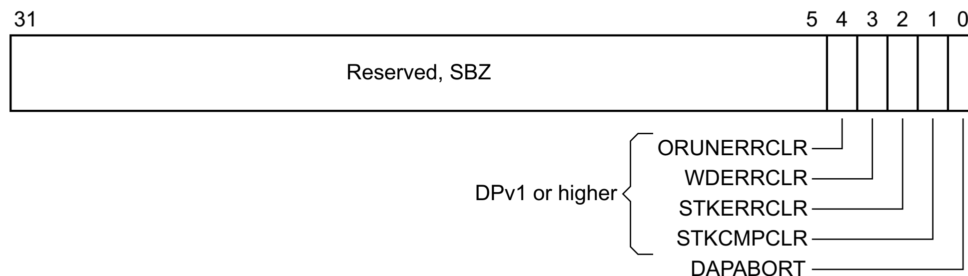
Attributes

ABORT is a 32-bit write-only DP architecture register.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

The ABORT bit assignments are:



Bits[31:1], DPv0
Bits[31:5], DPv1 or higher

Reserved, SBZ.

ORUNERRCLR, bit[4], DPv1 or higher

To clear the CTRL/STAT.STICKYORUN overrun error bit to `0b0`, write `0b1` to this bit.

WDERRCLR, bit[3], DPv1 or higher

To clear the CTRL/STAT.WDATAERR write data error bit to `0b0`, write `0b1` to this bit.

STKERRCLR, bit[2], DPv1 or higher

To clear the CTRL/STAT.STICKYERR sticky error bit to `0b0`, write `0b1` to this bit.

STKCMPCLR, bit[1], DPv1 or higher

To clear the CTRL/STAT.STICKYCMP sticky compare bit to `0b0`, write `0b1` to this bit. It is IMPLEMENTATION DEFINED whether the CTRL/STAT.STICKYCMP bit is implemented. See *MINDP, Minimal DP extension* on page B1-40.

DAPABORT, bit[0]

To generate a DAP abort, which aborts the current AP transaction, write `0b1` to this bit.

Do this write only if the debugger has received WAIT responses over an extended period.

In DPv0, this bit is SBO.

Accessing ABORT

Access to ABORT is DATA LINK DEFINED:

JTAG-DP Access is through its own scan-chain. See the *ABORT, JTAG-DP Abort register* on page B3-96.

SW-DP Accessed by a write to offset `0x0` of the DP register map.

DP Offset A[a]	SELECT.DPBANKSEL
`0x0`	X

a. Bits[1:0] of the register address are always `0b00`.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.2 CTRL/STAT, Control/Status register

The CTRL/STAT characteristics are:

Purpose

CTRL/STAT is a DP architecture register that is used to control and obtains status information about the DP.

Usage Constraints

Access to the register and its value after a powerup reset are defined for each field individually, as shown in the table. Some of fields are RO, meaning they ignore writes. See the field descriptions for detailed information.

Field	Access	Value after powerup reset
CDBGPWRUPACK	RO	
CDBGPWRUPREQ	RW	`0b0`
CDBGRSTACK	RO	
CDBGRSTREQ	IMPLEMENTATION DEFINED, RW, or RAZ/WI. See *Emulation of debug reset request*.	`0b0`
CSYSPWRUPACK	RO	
CSYSPWRUPREQ	RW	`0b0`
MASKLANE[a]	RW	UNKNOWN
ORUNDETECT	RW	`0b0`
READOK[b]	DATA LINK DEFINED, RES0 or RO/WI. See field description.	`0b0`
STICKYCMP[a]	DATA LINK DEFINED, R/W1C or RO/WI. See field description.	`0b0`
STICKYWERR	DATA LINK DEFINED, R/W1C or RO/WI. See field description.	`0b0`
STICKYORUN	DATA LINK DEFINED, R/W1C or RO/WI. See field description.	
TRNCNT[a]	RW	UNKNOWN
TRNMODE[a]	RW	UNKNOWN
WDATAERR	DATA LINK DEFINED, RES0 or RO/WI. See field description.	`0b0`

a. MASKLANE, TRNCNT, and TRNMODE are not supported in MINDP configuration. In MINDP configuration, the effect of writing a value other than zero to either TRNCNT or TRNMODE is UNPREDICTABLE.

b. DPv1 or higher.

Configurations

Included in all implementations.

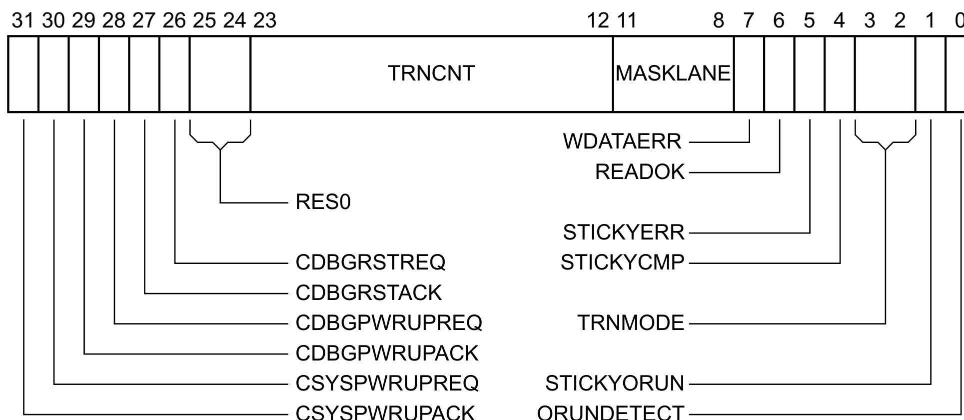
Attributes

CTRL/STAT is a 32-bit read/write register.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Field descriptions

The CTRL/STAT bit assignments are:



CSYSPWRUPACK, bit[31]

System powerup acknowledge. Indicates the status of the **CSYSPWRUPACK** signal. See *Power control requirements and operation* on page B2-79.

This bit is RO, meaning it ignores writes.

CSYSPWRUPREQ, bit[30]

System powerup request. This bit controls the **CSYSPWRUPREQ** signal. See *Power control requirements and operation* on page B2-79.

After a powerup reset, this bit is `0b0`.

CDBGPWRUPACK, bit[29]

Debug powerup acknowledge. Indicates the status of the **CDBGPWRUPACK** signal. See *Power control requirements and operation* on page B2-79.

This bit is RO, meaning it ignores writes.

CDBGPWRUPREQ, bit[28]

Debug powerup request. This bit controls the **CDBGPWRUPREQ** signal. See *Power control requirements and operation* on page B2-79.

After a powerup reset, this bit is `0b0`.

CDBGRSTACK, bit[27]

Debug reset acknowledge. Indicates the status of the **CDBGRSTACK** signal. See *Debug reset control behavior* on page B2-82.

This bit is RO, meaning it ignores writes.

CDBGRSTREQ, bit[26]

Debug reset request. This bit controls the **CDBGRSTREQ** signal. See *Debug reset control behavior* on page B2-82.

It is IMPLEMENTATION DEFINED whether this bit is RW or RAZ/WI. See *Emulation of debug reset request* on page B2-83.

After a powerup reset, this bit is `0b0`.

Bits[25:24]

Reserved, RES0.

TRNCNT, bits[23:12]

Transaction counter. See *The transaction counter* on page B1-43.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

After a powerup reset, the value of this field is UNKNOWN.

Note

It is IMPLEMENTATION DEFINED whether this field is implemented.

TRNCNT is not supported in MINDP configuration. In MINDP configuration, the effect of writing a value other than zero to TRNCNT or TRNMODE is UNPREDICTABLE. See also *MINDP, Minimal DP extension* on page B1-40.

MASKLANE, bits[11:8]

For pushed operations, the DP performs a byte-by-byte comparison of the word that is supplied in an AP write transaction with the current word at the target AP address. The MASKLANE field is used to select the bytes to be included in this comparison. For more information about pushed operations, see *Pushed-compare and pushed-verify operations* on page B1-44.

Each of the 4 bits of the MASKLANE field corresponds to one of the 4 bytes of the words to be compared. Therefore, each bit is said to control one byte lane of the compare operation.

Table B2-8 shows how the bits of MASKLANE control the comparison masking.

Table B2-8 Control of pushed operation comparisons by MASKLANE

MASKLANE	Effect	Bits included in comparisons[a]
`0b1xxx`	Include byte lane 3 in comparisons.	Bits[31:24].
`0bx1xx`	Include byte lane 2 in comparisons.	Bits[23:16].
`0bxx1x`	Include byte lane 1 in comparisons.	Bits[15:8].
`0bxxx1`	Include byte lane 0 in comparisons.	Bits[7:0].

a. Whether other bits are included is determined by the other bits of MASKLANE:
To compare the whole word, MASKLANE is set to `0b1111` to include all byte lanes.
If a MASKLANE bit is `0b0`, the corresponding byte lane is excluded from the comparison.

Note

The MASKLANE field is only relevant if the Transfer Mode field TRNMODE is `0b01`, for *pushed-verify operations*, or `0b10`, for *pushed-compare operations*. See the description of the TRNMODE field and *Pushed-compare and pushed-verify operations* on page B1-44.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal DP extension* on page B1-40.

After a powerup reset, the value of this field is UNKNOWN.

WDATAERR, bit[7]

This bit is set to `0b1` if one of the following Write Data Error occurs:

- A parity or framing error on the data phase of a write.
- A write that has been accepted by the DP is then discarded without being submitted to the AP.

For more information, see *Sticky flags and DP error responses* on page B1-41.

Access to and how to clear this field are DATA LINK DEFINED:

JTAG-DP, all implementations

Access is reserved, RES0.

SW-DP, all implementations, and JTAG-DP, DPv1 and higher

Access is RO/WI.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To clear WDATAERR to `0b0`, write `0b1` to the ABORT.WDERRCLR field in the ABORT register. A single write of the ABORT register can be used to clear multiple flags if necessary.

After clearing the WDATAERR flag, the data must typically be resent.

After a powerup reset, WDATAERR is `0b0`.

READOK, bit[6]

This bit is DATA LINK DEFINED.

- On JTAG-DP, the bit is reserved, RES0.
- On SW-DP, access is RO/WI.

  If the response to the previous AP read or RDBUFF read was OK, the bit is set to `0b1`. If the response was not OK, it is cleared to `0b0`.

  This flag always indicates the response to the last AP read access. See *Protocol error response* on page B4-114.

After a powerup reset, this bit is `0b0`.

Note

This field is defined for DPv1 and higher only.

STICKYERR, bit[5]

This bit is set to `0b1` if an error is returned by an AP transaction. See *Sticky flags and DP error responses* on page B1-41.

Access to and how to clear this field are DATA LINK DEFINED:

JTAG-DP, all implementations

- Access is R/W1C.
- To clear STICKYERR to `0b0`, write `0b1` to STICKYERR. This signals the DP to clear the flag and set it to `0b0`. A single write of the CTRL/STAT register can be used to clear multiple flags if necessary.

  STICKYERR can also be cleared using the ABORT.STKERRCLR field.

SW-DP, all implementations, and JTAG-DP, DPv1 and higher

- Access is RO/WI.
- To clear STICKYERR to `0b0`, write `0b1` to the ABORT.STKERRCLR field in the ABORT register. A single write of the ABORT register can be used to clear multiple flags if necessary.

After clearing CTRL/STAT.STICKYERR, you must find the location where the error that caused the flag to be set occurred.

After a powerup reset, this bit is `0b0`.

STICKYCMP, bit[4]

This bit is set to `0b1` when a mismatch occurs during a pushed-compare operation or a match occurs during a pushed-verify operation. See *Pushed-compare and pushed-verify operations* on page B1-44.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal DP extension* on page B1-40.

Access to and how to clear this field are DATA LINK DEFINED:

JTAG-DP, all implementations

- Access is R/W1C.
- To clear STICKYCMP to `0b0`, write `0b1` to STICKYCMP. This signals the DP to clear the flag and set it to `0b0`. A single write of the CTRL/STAT register can be used to clear multiple flags if necessary.

  STICKYCMP can also be cleared using the ABORT.STKERRCLR field.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

SW-DP, all implementations, and JTAG-DP, DPv1 and higher

- Access is RO/WI.
- To clear STICKYCMP to `0b0`, write `0b1` to the ABORT.STKCMPCLR field in the ABORT register. A single write of the ABORT register can be used to clear multiple flags if necessary.

After clearing STICKYCMP, you must retrieve the value of the transaction counter to find the location where the error that caused the flag to be set occurred.

After a powerup reset, this bit is `0b0`.

TRNMODE, bits[3:2]

This field sets the transfer mode for AP operations.

In normal operation, AP transactions are passed to the AP for processing, as described in *Using the AP to access debug resources* on page A1-31.

In pushed-verify and pushed-compare operations, the DP compares the value that is supplied in an AP write transaction with the value held in the target AP address. The AP write transaction generates a read access to the debug memory system as described in *Pushed-compare and pushed-verify operations* on page B1-44.

TRNMODE can have one of the following values:

`0b00` Normal operation.

`0b01` Pushed-verify mode.

`0b10` Pushed-compare mode.

`0b11` Reserved.

After a powerup reset, the value of this field is UNKNOWN.

Note

It is IMPLEMENTATION DEFINED whether this field is implemented.

TRNMODE is not supported in MINDP configuration. In MINDP configuration, the effect of writing a value other than zero to TRNCNT or TRNMODE is UNPREDICTABLE. See also *MINDP, Minimal DP extension* on page B1-40.

STICKYORUN, bit[1]

If overrun detection is enabled, this bit is set to `0b1` when an overrun occurs. See bit[0] of this register for details of enabling overrun detection.

Access to and how to clear this field are DATA LINK DEFINED:

JTAG-DP, all implementations

- Access is R/W1C.
- To clear STICKYORUN to `0b0`, write `0b1` to STICKYORUN. This signals the DP to clear the flag and set it to `0b0`. A single write of the CTRL/STAT register can be used to clear multiple flags if necessary.

  STICKYORUN can also be cleared using the ABORT.STKERRCLR field.

SW-DP, all implementations, and JTAG-DP, DPv1 and higher

- Access is RO/WI.
- To clear STICKYORUN to `0b0`, write `0b1` to the ABORT.ORUNERRCLR field in the ABORT register. A single write of the ABORT register can be used to clear multiple flags if necessary.

After clearing STICKYORUN, the specific DP or AP transaction initiated the overrun that caused the flag to be set must be identified. The transactions for that DP or AP are repeated from the transaction pointed to by the transaction counter.

After a powerup reset, this bit is `0b0`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

ORUNDETECT, bit[0]

This bit can have one of the following values:

`0b0` Overrun detection is disabled.

`0b1` Overrun detection is enabled.

For more information about overrun detection, see *Sticky flags and DP error responses* on page B1-41.

After a powerup reset, this bit is `0b0`.

Accessing CTRL/STAT

CTRL/STAT can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x4`	`0x0`

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.3 DLCR, Data Link Control register

The DLCR characteristics are:

Purpose

DLCR controls the operating mode of the Data Link.

Usage Constraints

DLCR is DATA LINK DEFINED:

- For a JTAG DP, the DLCR register is RES0.
- For an SW-DP, the DLCR register has the fields that are described in *Field descriptions* on page B2-63.

DLCR is accessible as follows:

Default
RW

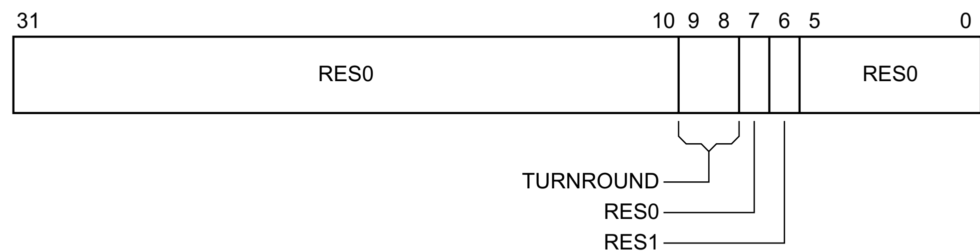
Configurations

DLCR is implemented in DPv1 and DPv2.

Attributes DLCR is a 32-bit DATA LINK DEFINED DP architecture register.

Field descriptions

The DLCR bit assignments for an SW-DP are:



Bits[31:10]

Reserved, RES0.

TURNROUND, bits[9:8]

For an SW-DP, this field defines the turnaround tristate period. For details about line turnaround, see *Line turnaround* on page B4-107. Table B2-9 shows the permitted values of this field, and their meanings.

Table B2-9 Turnaround tristate period field, TURNROUND, bit definitions

DLCR.TURNROUND	Turnaround tristate period
`0b00`	1 data period[a].
`0b01`	2 data periods[a].
`0b10`	3 data periods[a].
`0b11`	4 data periods[a].

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. A *data period* is the period of a single data bit on the SWD interface.

After a powerup or line reset, this field is `0b00`.

Note

Support for varying the turnaround tristate period is IMPLEMENTATION DEFINED. An implementation that does not support variable turnaround must treat writing a value other than `0b00` to the TURNROUND field as an immediate protocol error.

Bit[7] Reserved, RES0.

Bit[6] Reserved, RES1.

Bits[5:0]

Reserved, RES0.

Accessing DLCR

DLCR can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x4`	`0x1`

a. Bits[1:0] of the register address are always `0b00`.

B2.2.4 DLPIDR, Data Link Protocol Identification register

The DLPIDR characteristics are:

Purpose DLPIDR provides protocol version information.

Configurations

DLPIDR is implemented in DPv2.

Note

An SWD Port that implements DPv2 must implement at least SWD protocol version 2.

Usage Constraints

For a JTAG-DP, DLPIDR is reserved and any result from accessing the register is UNPREDICTABLE.

DLPIDR is accessible as follows:

Default
RO

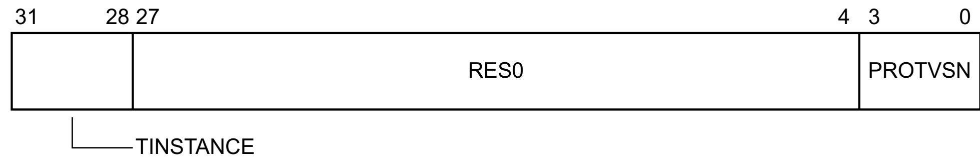
Attributes

DLPIDR is a 32-bit DATA LINK DEFINED register.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Field descriptions

For an SW-DP, the DLPIDR bit assignments are:



TINSTANCE, bits[31:28]

IMPLEMENTATION DEFINED. Defines an instance number for this device. This value must be unique for all devices with identical TARGETID.TPARTNO and TARGETID.TDESIGNER fields that are connected together in a multi-drop system.

Bits[27:4] RES0.

PROTVSN, bits[3:0]

Defines the SWD protocol version that is implemented. Valid values for this field are:

`0x1` SWD protocol version 2. Adds support for multidrop extensions. See Chapter B4 *The Serial Wire Debug Port*.

All other values of this field are reserved.

Accessing DLPIDR

DLPIDR can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x4`	`0x3`

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2.5 DPIDR, Debug Port Identification Register

The DPIDR characteristics are:

Purpose

DPIDR provides information about the DP.

Usage Constraints

DPIDR is accessible as follows:

Default
RO

Configurations

DPIDR is defined and implemented only in DPv1 and DPv2.

Note

In DPv0, the DPIDR is reserved and accesses are UNPREDICTABLE.

In all DP architecture versions, a JTAG-DP implementation must implement the IDCODE instruction and IDCODE scan-chain. The architecture does not require that the TAP IDCODE register value and the DPIDR value are the same.

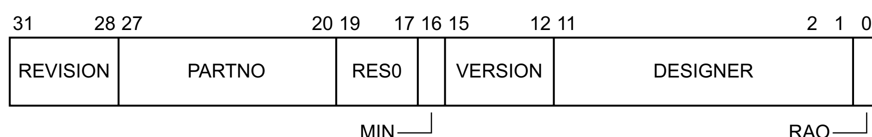
Attributes

A 32-bit DP architecture register.

Access to the DPIDR is not affected by the value of SELECT.DPBANKSEL.

Field descriptions

The DPIDR bit assignments are:



REVISION, bits[31:28]

Revision code. The meaning of this field is IMPLEMENTATION DEFINED.

PARTNO, bits[27:20]

Part Number for the DP. This value is provided by the designer of the DP and must not be changed.

Bits[19:17] Reserved, RES0.

MIN, bit[16] MINDP functions implemented:

`0b0` Transaction counter, Pushed-verify, and Pushed-find operations are implemented.

`0b1` Transaction counter, Pushed-verify, and Pushed-find operations are not implemented.

VERSION, bits[15:12]

Version of the DP architecture implemented. Permitted values are:

`0x0` Reserved. Implementations of DPv0 do not implement DPIDR.

`0x1` DPv1 is implemented.

`0x2` DPv2 is implemented.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

All remaining values are reserved.

DESIGNER, bits[11:1]

Code that identifies the designer of the DP.

This field indicates the designer of the DP 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 encoding that is used in the DPIDR is as follows:

- The JEP106 continuation code, DPIDR bits[11:8], is the number of times that `0x7F` appears before the final number. For example, for Arm Limited this field is `0x4`.
- The JEP106 identification code, IDR bits[7:1], equals bits[6:0] of the final number of the JEDEC code. For example, for Arm® Limited this field is `0x3B`.

Bit[0] RAO.

Accessing DPIDR

DPIDR can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x0`	`X`

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.6 EVENTSTAT, Event Status register

The EVENTSTAT characteristics are:

Purpose EVENTSTAT is used by the system to signal an event to the external debugger. The nature of the event is IMPLEMENTATION DEFINED.

Arm recommends connecting EVENTSTAT to one of the following:

- An output trigger of a CoreSight *Cross-Trigger Interface* (CTI) with software acknowledge.
- An output from a uniprocessor system that indicates whether the processor is halted:
  - — For Armv6-M, Armv7-M, and Armv8-M processors, the recommended **HALTED** signal.
  - — For all other Arm architecture processors, the recommended **DBGACK** signal.

Usage Constraints

EVENTSTAT is accessible as follows:

Default
RO

Configurations

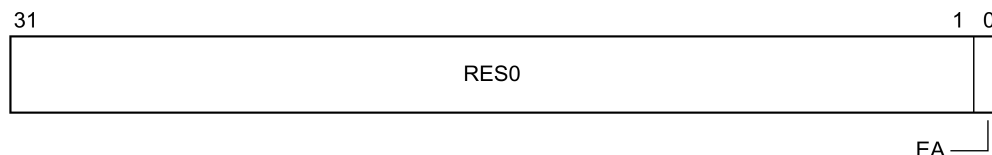
EVENTSTAT is implemented in DPv2.

Attributes

A 32-bit RO register.

Field descriptions

The EVENTSTAT bit assignments are:



Bits[31:1] Reserved, RES0.

EA, bit[0] If an event is implemented, this field is the event status flag. Valid values for the flag are:

`0b0` An event requires attention.

`0b1` There is no event requiring attention.

If no event is implemented, this field is RAZ.

Note

The status of the event is inverted in the register, and when debugging an implementation that does not implement an event, debuggers interpret a value of zero as an event requiring attention, and poll other registers to detect the status of the system.

Accessing EVENTSTAT

EVENTSTAT can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x4`	`0x4`

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

B2.2.7 RDBUFF, Read Buffer register

The RDBUFF characteristics are:

Purpose

The purpose and behavior of RDBUFF is DATA LINK DEFINED:

JTAG-DP The Read Buffer is architecturally defined to provide a DP read operation that does not have any side effects. This definition allows a debugger to insert a DP read of RDBUFF at the end of a sequence of operations, to return the final AP Read Result and ACK values.

SW-DP On an SW-DP, the Read Buffer presents data that was captured during the previous AP read, enabling repeatedly returning the value without generating a new AP access.

Note

After reading the DP Read Buffer, its contents are no longer valid. The result of a second read of the DP Read Buffer is UNKNOWN.

If you require the value from an AP register read, that read must be followed by one of:

- A second AP register access, with the appropriate AP selected as the current AP.
- A read of the DP Read Buffer.

The second access to either the AP or the DP stalls until the result of the original AP read is available.

Usage Constraints

RDBUFF is accessible as follows:

Default
RO

Configurations

RDBUFF is implemented in DPv0, DPv1, and DPv2.

Attributes

A 32-bit read-only buffer.

The RDBUFF bit assignments are:

Bits[31:0] DATA LINK DEFINED:

JTAG-DP RAZ/WI

SW-DP Data for previous AP read.

Accessing RDBUFF

RDBUFF can be accessed at the following address:

DP Offset[a]	SELECT.DPBANKSEL
`0xC`	x

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2.8 RESEND, Read Resend register

The RESEND register characteristics are:

Purpose

Performing a read to RESEND does not capture new data from the AP, but returns the value that was returned by the last AP read or DP RDBUFF read.

RESEND enables the debugger to recover read data from a corrupted SW-DP transfer without having to re-issue the original read request, or generate a new access to the connected debug memory system.

RESEND can be accessed multiple times, and always returns the same value until a new access is made to an AP register or the DP RDBUFF register.

Usage Constraints

Arm recommends that debuggers only access RESEND when a failed read has been indicated by the SW-DP, and at no other time. The reason for this is that, if an implementation cannot resend the information, it is permitted to treat reads RESEND as a protocol error.

RESEND is accessible as follows:

Default
RO

Configurations

RESEND is implemented in DPv1 and DPv2.

For these versions, RESEND is included in all implementations.

Attributes

A 32-bit read-only DP architecture register.

RESEND is:

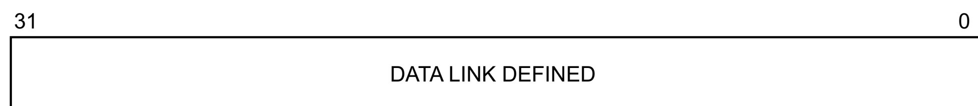
- A read-only register.
- Accessed by a read of offset `0x8` in the DP register map.
- DATA LINK DEFINED:

  JTAG-DP The register is reserved, any access is UNPREDICTABLE.

  SW-DP The value that was returned by the last AP read or DP RDBUFF read.

Field descriptions

The RESEND bit assignments are:



Bits[31:0] DATA LINK DEFINED:

JTAG-DP The register is reserved, any access is UNPREDICTABLE.

SW-DP Data for previous AP read.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing RESEND

RESEND can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x8`	`x`

a. Bits[1:0] of the register address are always `0b00`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.9 SELECT, AP Select register

The SELECT characteristics are:

Purpose

SELECT:

- Selects an AP and the active register banks within that AP.
- Selects the DP address bank.

Usage Constraints

SELECT is accessible as follows:

DPv0	DPv1	DPv2
RW	WO	WO

Configurations

A DP architecture register. SELECT is implemented in DPv0, DPv1, and DPv2.

Note

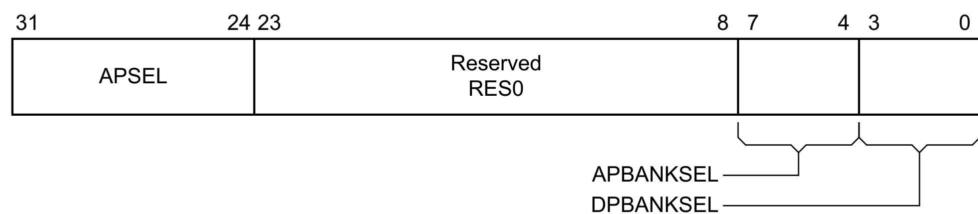
This specification deprecates reading SELECT in a DPv0 implementation.

Attributes

SELECT is a 32-bit DP architecture register.

Field descriptions

The SELECT bit assignments are:



APSEL, bits[31:24]

Selects the AP with the ID number APSEL. If there is no AP with the ID APSEL, all AP transactions return zero on reads and are ignored on writes. See *Register maps, and accesses to reserved addresses* on page B2-52.

After a powerup reset, the value of this field is UNKNOWN.

Note

Every ADI implementation must include at least one AP.

Bits[23:8] Reserved, RES0.

APBANKSEL, bits[7:4]

Selects the active four-word register bank on the current AP. See *Using the AP to access debug resources* on page A1-31.

After a powerup reset, the value of this field is UNKNOWN.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DPBANKSEL, bit[3:0]

Debug Port address bank select.

The behavior of SELECT.DPBANKSEL depends on the DP version, as follows:

DPv0 In DPv0, the SELECT.DPBANKSEL field must be written as zero, otherwise accesses to DP register `0x4` are UNPREDICTABLE.

DPv1 In DPv1, the SELECT.DPBANKSEL field controls which DP register is selected at address `0x4`, and Table B2-10 shows the permitted values of this field.

Table B2-10 DPBANKSEL DP register allocation in DPv1

DPBANKSEL	DP register at address `0x4`
`0x0`	CTRL/STAT
`0x1`	DLCR

All other values of SELECT.DPBANKSEL are reserved. If the field is set to a reserved value, accesses to DP register `0x4` are UNPREDICTABLE.

DPv2 In DPv2 the SELECT.DPBANKSEL field controls which DP register is selected at address `0x4`, and Table B2-11 shows the permitted values of this field.

Table B2-11 DPBANKSEL DP register allocation in DPv2

DPBANKSEL	DP register at address `0x4`
`0x0`	CTRL/STAT
`0x1`	DLCR
`0x2`	TARGETID
`0x3`	DLPIDR
`0x4`	EVENTSTAT

All other values of SELECT.DPBANKSEL are reserved. If the field is set to a reserved value, accesses to DP register `0x4` are RES0.

After a powerup reset, this field is `0x0`.

Note

Some previous ADI revisions have described DPBANKSEL as a single-bit field called CTRSEL, defined only for SW-DP. From Issue B of this specification, DPBANKSEL is redefined. The new definition is backwards-compatible.

Accessing SELECT

SELECT can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x8`	Not applicable

a. Bits[1:0] of the register address are always `0b00`.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.10 TARGETID, Target Identification register

The TARGETID characteristics are:

Purpose

TARGETID provides information about the target when the host is connected to a single device.

Usage Constraints

TARGETID is accessible as follows:

Default
RO

Configurations

TARGETID is implemented in DPv2.

Attributes

TARGETID is a 32-bit read-only register.

Field descriptions

The TARGETID bit assignments are:

31 - 28	27 - 12	11 - 1	0
TREVISION	TPARTNO	TDESIGNER	1

TREVISION, bits[31:28]

Target revision.

TPARTNO, bits[27:12]

IMPLEMENTATION DEFINED. The value is assigned by the designer of the part. The value must be unique to the part.

TDESIGNER, bits[11:1]

IMPLEMENTATION DEFINED.

This field indicates the designer of the part and not the implementer, except where the two are the same.

Designers must insert their JEDEC-assigned code here.

Note

The Arm JEP106 value is not shown for the TDESIGNER field. Arm might design a DP containing the TARGETID register, but typically, the designer of the part, referenced in the TPARTNO field, is another designer who creates a part around the licensed Arm IP. If the designer of the part is Arm, then the value of this field is `0x23B`.

To obtain a number, or to see the assignment of these codes, contact JEDEC at http://www.jedec.org.

A JEP106 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`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encoding that is used in TARGETID is as follows:

- The JEP106 continuation code, TARGETID bits[11:8], is the number of times that `0x7F` appears before the final number. For example, for Arm Limited this field is `0x4`.
- The JEP106 identification code, TARGETID bits[7:1], equals bits[6:0] of the final number of the JEDEC code.

Bit[0] RAO.

Accessing TARGETID

TARGETID can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0x4`	`0x2`

a. Bits[1:0] of the register address are always `0b00`.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.2.11 TARGETSEL, Target Selection register

The TARGETSEL characteristics are:

Purpose

TARGETSEL selects the target device in an SWD multi-drop system.

On a write to TARGETSEL immediately following a line reset sequence, the target is selected if both the following conditions are met:

- Bits[31:28] match bits[31:28] in the DLPIDR.
- Bits[27:0] match bits[27:0] in the TARGETID register.

Writing any other value de-selects the target. Debug tools must write `0xFFFFFFFF` to de-select all targets. `0xFFFFFFFF` is an invalid TARGETID value. All other invalid TARGETID values are reserved.

During the response phase of a write to the TARGETSEL register, the target does not drive the line. See *Sticky flags and DP error responses* on page B1-41 for more information.

Usage Constraints

The register is DATA LINK DEFINED:

JTAG-DP The register is reserved, any access is UNPREDICTABLE.

SW-DP If SWD protocol version 2 is implemented, the register is implemented.

The register is accessible as follows:

SW-DP	JTAG-DP
WO	UNPREDICTABLE

Configurations

TARGETSEL is implemented in DPv2.

Note

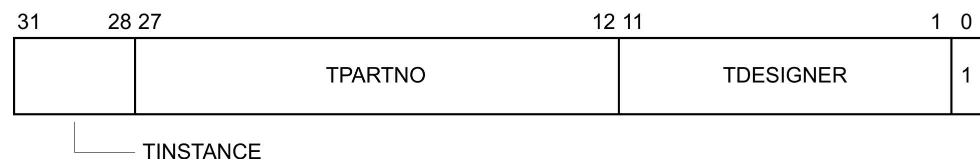
In Issue A of this document, this register was called ROUTESEL, and was a reserved write-only register.

Attributes

TARGETSEL is a 32-bit DP architecture register.

Field descriptions

For an SW-DP, the TARGETSEL bit assignments are:



TINSTANCE, bits[31:28]

IMPLEMENTATION DEFINED. The instance number for this device. See DLPIDR.

TPARTNO, bits[27:12]

IMPLEMENTATION DEFINED. The value that is assigned by the designer of the part. See TARGETID.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDESIGNER, bits[11:1]

IMPLEMENTATION DEFINED. The 11-bit code that is formed from the JEDEC JEP106 continuation code and identity code. See TARGETID.

Bit[0] SBO.

Accessing TARGETSEL

TARGETSEL can be accessed at the following address:

DP Offset A[a]	SELECT.DPBANKSEL
`0xC`	`x`

a. Bits[1:0] of the register address are always `0b00`.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.3 System and debug power control behavior

This section gives detailed information about system and debug power.

B2.3.1 The DAP power domains model

The DAP model supports multiple power domains, which provide support for debug components that can be powered down.

Three power domains are modeled:

Always-on power domain

Power domain that must be powered up for the debugger to connect to the device.

System power domain

Power domains that include system components.

Debug power domain

Power domain that includes the entire debug subsystem.

The system and debug power domains can be subdivided if necessary. However, to define a simple debug interface, the device must be partitioned into system and debug power domains at the top level. Any finer-grained control is outside the scope of this model.

In most situations, debuggers power up the complete SoC. However, if a debugger is investigating an energy management issue, it might want to power up only the debug domain. To achieve this goal, SoC designers might want to map the power controller into a bus segment that the DAP can access when only the debug power domain is powered up.

When using an Arm Debug Interface, for the debug process to work correctly, systems must not remove power from the DP during a debug session. If power is removed, the DAP controller state is lost. However, the DAP is designed to permit the rest of the DAP and the system to be powered down and debugged while maintaining power to the DP.

The DP registers reside in the always-on power domain, on the external interface side of the DP. Therefore, DP registers can always be driven, enabling powerup requests to be made to a system power controller. The power and reset control bits are part of the DP CTRL/STAT register. See *Debug reset control behavior* on page B2-82 for more information about the reset control bits in this register.

ADIv5 defines two pairs of power control signals:

- **CDBGPWRUPREQ** and **CDBGPWRUPACK**.
- **CSYSPWRUPREQ** and **CSYSPWRUPACK**.

Table B2-12 summarizes the programmers' model for the power control signal pairs.

Table B2-12 Debug Port programmers' model

Signal	Programmers' model
CDBGPWRUPREQ	Bit[28] of the CTRL/STAT register
CDBGPWRUPACK	Bit[29] of the CTRL/STAT register
CSYSPWRUPREQ	Bit[30] of the CTRL/STAT register
CSYSPWRUPACK	Bit[31] of the CTRL/STAT register

These signals are expected to provide requests to the system power and clock controller. The following sections describe these signal pairs.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CDBGPWRUPREQ and CDBGPWRUPACK

CDBGPWRUPREQ is the signal from the debug interface to the power controller. This signal requests the system power controller to fully power up and ensure the clocks are available in the debug power domain.

CDBGPWRUPACK is the signal from the power controller to the debug interface. When **CDBGPWRUPREQ** is asserted, the power controller powers up the debug power domain and then asserts **CDBGPWRUPACK** to acknowledge that it has responded to the request.

It is IMPLEMENTATION DEFINED which components are in the debug power domain that is controlled by **CDBGPWRUPREQ**. This domain might include all debug components in the system, or it might, for example, be limited to exclude components that have extra levels of power control. The **CDBGPWRUPREQ** signal indicates that the debugger requires the debug resources of these components to be communicative. Communicative means that the debugger can access at least enough registers of the debug resource for it to determine the state of the resource. Whether the resource is active is IMPLEMENTATION DEFINED. The power and clock controller must power up and run the clocks of as many domains as necessary to comply with this request from the debugger for the resources to be communicative.

The power and clock controller must honor **CDBGPWRUPREQ** for as long as it is asserted. For example, if a component in a debug power domain requests to be powered down, the request must be emulated for non-debug logic within that power domain, including all components with a single shared domain.

If some debug resources of a component are not in the debug power domain, then at least the minimal debug interface of the component must be powered up. If the following requirements are met, power can be removed from the remainder of the component:

- There is some means to save and restore the state of the debug resources.
- The debugger can communicate with the debug resources when the remainder of the component is not powered.

The means to save and restore the values that are held in these resources might include software solutions. If the debug resources do lose their value when power is removed from the remainder of the component, then the debug interface must include means for the debugger to discover that the programmed values have been lost.

CDBGPWRUPACK is the acknowledge signal for the **CDBGPWRUPREQ** request signal. **CDBGPWRUPACK** must be asserted for as long as **CDBGPWRUPREQ** is asserted. See *Powerup request and acknowledgement timing on page B2-80*.

CSYSPWRUPREQ and CSYSPWRUPACK

CSYSPWRUPREQ is the signal from the debug interface to the power controller. This signal requests the system power controller to fully power up and ensure the clocks are available in the system power domain.

CSYSPWRUPACK is the signal from the power controller to the debug interface. When **CSYSPWRUPREQ** is asserted, the power controller powers up the system power domain and then asserts **CSYSPWRUPACK** to acknowledge that it has responded to the request.

It is IMPLEMENTATION DEFINED which components are in the debug power domain that is controlled by **CSYSPWRUPREQ**. This domain might include all debug components in the system, or might be limited, for example, to exclude components that have extra levels of power control, such as processors that implement independent Core Powerup Request controls.

The **CSYSPWRUPREQ** signal indicates that the debugger requires all debug resources of these components to be active. Active means that the debug resource can perform its debug function. An active resource is also communicative.

The power and clock controller must honor **CSYSPWRUPREQ** for as long as it is asserted.

CSYSPWRUPREQ has no effect on debug components that are controlled by **CDBGPWRUPREQ**, because those components have no debug logic in the system power domain. However, for components where some debug resources are in the system power domain that is controlled by **CSYSPWRUPREQ**, the request must be emulated for non-debug logic within that power domain.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

CSYSPWRUPACK is the acknowledge signal for the **CSYSPWRUPREQ** request signal. **CSYSPWRUPACK** must be asserted for as long as **CSYSPWRUPREQ** is asserted. See *Powerup request and acknowledgement timing* on page B2-80.

When **CSYSPWRUPREQ** is asserted by the debugger, **CDBGPWRUPREQ** must also be asserted.

B2.3.2 Power control requirements and operation

This section applies to both the system and the debug domain, and uses the following notation:

- **CxxxPWRUPREQ** refers to either **CSYSPWRUPREQ** or **CDBGPWRUPREQ**.
- **CxxxPWRUPACK** refers to either **CSYSPWRUPACK** or **CDBGPWRUPACK**.

All signals that are described in this section are active-high, so assert denotes taking the signal HIGH, and deassert denotes taking the signal LOW.

The rules for the operation of powerup requests and acknowledgments are:

- The debugger must not set CTRL/STAT.CSYSPWRUPREQ to `0b1` and CTRL/STAT.CDBGPWRUPREQ to `0b0` at the same time. The response to this combination of requests is UNPREDICTABLE.
- To initiate powerup, the DP must assert **CxxxPWRUPREQ**.
  - If the corresponding power domain is powered down or in a low-power retention state, the power controller must power up the domain when it detects that **CxxxPWRUPREQ** is asserted. After the domain is powered up, the controller must assert **CxxxPWRUPACK**.
  - If the corresponding power domain is already powered up when the power controller detects that **CxxxPWRUPREQ** is asserted, the controller must still respond by asserting **CxxxPWRUPACK**, even though it does not affect the power domain.
- Arm strongly recommends that tools only initiate an AP transfer when **CDBGPWRUPREQ** and **CDBGPWRUPACK** are asserted. If **CDBGPWRUPREQ** or **CDBGPWRUPACK** is LOW, any AP transfer might generate a fault response.
- The DP requests removal of power to a domain by deasserting **CxxxPWRUPREQ**.

  The power controller deasserts **CxxxPWRUPACK** when it has accepted the request to power down the domain.

  Note

  The power controller deasserting **CxxxPWRUPACK**, does not indicate that the domain has been powered down, it only indicates that the power controller has recognized and accepted the request to remove power.

- **CxxxPWRUPACK** must default to the LOW state, and only go HIGH on receipt of a **CxxxPWRUPREQ** request.
- After detecting the deassertion of **CxxxPWRUPREQ**, the power controller must gracefully power down the domain, unless removal of power from the domain would affect system operation. For example, the power controller might maintain power to the domain if it has other requests to maintain power.
- After powerdown has been requested through the deassertion of **CxxxPWRUPREQ**, tools must wait until **CxxxPWRUPACK** is LOW before making a new request for powerup.

  This requirement ensures that the power control handshaking mechanism is not violated.

Figure B2-1 on page B2-80 shows the timing of the power control signals.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

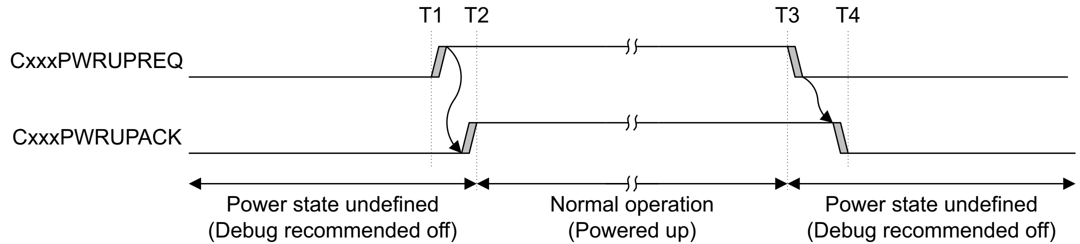


Figure B2-1 Powerup request and acknowledgement timing

——— **Note** ———

Arm strongly recommends that all AP transactions are initiated between times T2 and T3 for **CDBGPWRUPREQ** and **CDBGPWRUPACK**, as shown in Figure B2-1.

B2.3.3 Emulation of powerdown

If a DAP asserts **CxxxPRWUPREQ** for a domain and the power controller receives a conflicting request for the domain from another source, it must emulate the powerdown request for the domain by completing the handshake process as expected, without actually removing power from the domain. This requirement enables debugging a system with power domains that power up and down dynamically.

The following requests cause a conflict when issued by another source after the DAP has asserted **CxxxPRWUPREQ**:

- A powerdown request.
- A request to enter a low-power retention mode, with clocks disabled.

Emulation of powerdown is relevant to application debugging, when the application developer does not care whether the core domain actually powers up and down because this aspect is controlled at the OS level.

B2.3.4 Emulation of power control

If the system to which a DAP is connected does not support the ADIv5 power control model, the required signals must be emulated or generated from other signals:

System power controllers that do not support the ADIv5 power control scheme

To ensure that the DAP receives an immediate acknowledgment after initiating or removing a powerup request, connect **CxxxPRWUPACK** to **CxxxPRWUPREQ**, as shown in Figure B2-2.

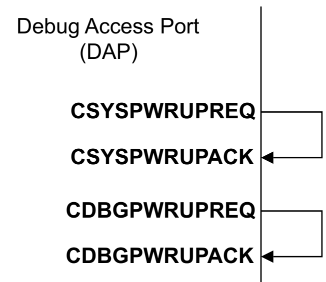


Figure B2-2 Emulation of powerup control

System power controllers that do not support separate power domains

If the debug power domain is part of the system power domain, **CSYSPWRUPREQ** and **CDBGPWRUPREQ** can independently request powerup. To correctly emulate power control:

- To ensure that the DAP receives an immediate acknowledgement of after initiating or removing a system powerup request, connect **CSYSPRWUPACK** to **CSYSPRWUPREQ**.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- Generate appropriate **CxxxPRWUPACK** signals that ensure that the DAP sees the correct response when it asserts **CxxxPRWUPREQ**.

Note

The **CxxxPWRUPACK** signals must be emulated as described. Setting CTRL/STAT.CSYSPWRUPREQ to `0b1` and CTRL/STAT.CDBGPWRUPREQ to `0b0` in the CTRL/STAT register leads to UNPREDICTABLE system behavior.

The connections are shown in Figure B2-3.

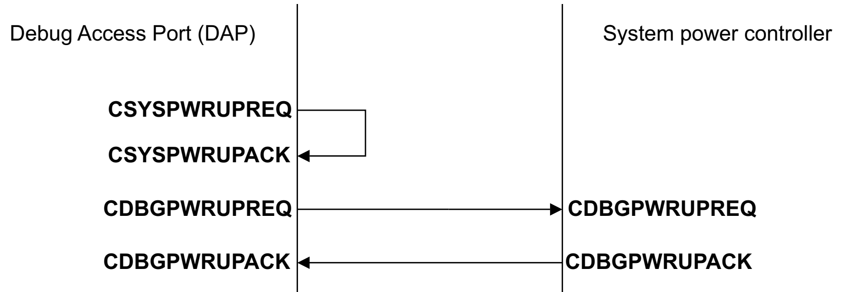


Figure B2-3 Signal generation for a single system and debug power domain

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.4 Debug reset control behavior

The DP register CTRL/STAT provides two fields for reset control of the debug domain:

- CTRL/STAT.CDBGRSTREQ, Debug reset request.
- CTRL/STAT.CDBGRSTACK, Debug reset acknowledge.

The associated signals, **CDBGRSTREQ** and **CDBGRSTACK**, provide a connection to a system reset controller. The debug domain that is controlled by these signals covers the internal DAP and the connection between the DAP and the debug components, for example the debug bus.

The DP registers are in the always-on power domain on the external interface side of the DP. Therefore, the registers can be driven at any time, to generate a reset request to the system reset controller.

Figure B2-4 shows the reset request and acknowledge timing.

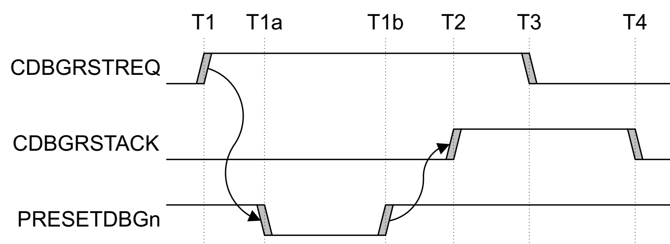


Figure B2-4 Reset request and acknowledge timing

——— Note ———

The use of AMBA APB signal names in the examples do not indicate a requirement that a debug bus must be implemented using an AMBA APB.

———————————

The steps in Figure B2-4 include:

1. At T1, the debugger writes `0b1` to CTRL/STAT.CDBGRSTREQ. This initiates the reset request.

   The debug domain is reset between T1a and T1b. The reset is complete by T2. This operation resets the AP registers and other AP state.

   ——— Note ———

   There is no reset of the DP registers and DP state. These registers are only reset by a powerup reset.

   ———————————

2. At T2, the system reset controller acknowledges that the reset of the debug domain has completed. The **CDBGRSTACK** signal sets the CTRL/STAT.CDBGRSTACK bit to `0b1`.

3. At T3, the debugger checks the DP CTRL/STAT register and finds that the reset has completed. Therefore, it writes `0b0` to CTRL/STAT.CDBGRSTREQ, which removes the reset request signal.

4. At T4, the system reset controller recognizes that **CDBGRSTREQ** is no longer asserted, and deasserts **CDBGRSTACK**.

——— Caution ———

If **CDBGRSTREQ** is removed before the reset controller asserts **CDBGRSTACK**, the behavior is UNPREDICTABLE.

———————————

The AP debug components are also reset on powerup of the debug power domain.

A debug reset request has no effect on devices that are powered down when the request is issued.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.4.1 Emulation of debug reset request

If the debug reset control is not supported then:

- It is IMPLEMENTATION DEFINED whether CTRL/STAT.CDBGRSTREQ is read/write or RAZ/WI.
- CTRL/STAT.CDBGRSTACK is RAZ.

Note

Arm recommends tying **CDBGRSTACK** LOW so that the debugger can use a timeout mechanism to detect whether debug reset is implemented.

B2.4.2 Limitations of CDBGRSTREQ and CDBGRSTACK

Debug reset control behavior on page B2-82 shows how these bits can drive the debug reset signal, **PRESETDBGn**. In an actual system, there might be other reset signals that are associated with other debug buses. For example, in a CoreSight system, **ATRESETn** resets all registers in the AMBA ATB domain.

Note

It is IMPLEMENTATION DEFINED which components are reset by **CDBGRSTREQ**. Figure B2-4 on page B2-82 is an example only. Not only components that use **PRESETDBGn** are reset.

Because debug logic might be accessible by the system, an implementation might have corner cases if **CDBGRSTREQ** is set at the same time as the system is using the debug logic. For example, the reset might occur during a transaction, causing a system or software malfunction.

It is IMPLEMENTATION DEFINED whether **CDBGRSTREQ** can be used when debug power is off.

A system might include IMPLEMENTATION DEFINED conditions which prevent a debug reset from occurring, for example when certain levels of debug are not permitted.

When a debug reset is prevented from occurring, **CDBGRSTREQ** is ignored and **CDBGRSTACK** is held LOW.

Caution

System-level use of debug components must be handled with caution. Arm recommends that such system-level usage is not combined with a reset system that permits those debug components to be reset without the knowledge of the system. Arm recommends that debuggers do not use debug reset requests unless necessary.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B2.5 System reset control behavior

The DP does not provide control bits for requesting a system reset. However, it is common for the physical interface to the debugger to include a system reset pin, **nSRST**. This section describes the recommended behavior of the system when a system reset is requested on the **nSRST** pin.

nSRST is an active-LOW pin that can be asserted and deasserted at any point in time, regardless of the current state of the target system, to return the target system to a known state for booting and for starting a debug session.

While **nSRST** is asserted:

- The target system must be held in the known state.
- The debugger must be able to access the debug domain of the target system.

B2.5.1 Limitations of system reset control

The debugger must ensure that the DAP is not accessing the system when asserting **nSRST**. When **nSRST** is asserted, the debugger can access the debug domain.

The effect of **nSRST** on the debug domain is IMPLEMENTATION DEFINED.

For example, to safely return the target system to a known state, the debug domain might also require to be reset. When **nSRST** is asserted, the entire system must be reset, including the debug domain. However, the debug domain must be released from reset to allow the debugger access. Only the non-debug domain is held in reset while **nSRST** is asserted.

Arm recommends that debuggers set CTRL/STAT.CDBGPWRUPREQ to `0b0` while **nSRST** is initially asserted.

Figure B2-5 shows a system reset timing example.

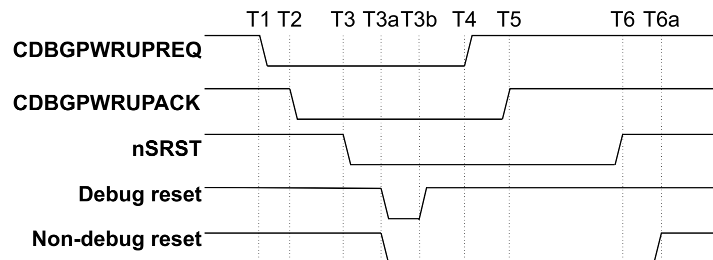


Figure B2-5 Example system reset timing

The steps in Figure B2-5 include:

1. At time T1, the debugger writes `0b0` to CTRL/STAT.CDBGPWRUPREQ.
2. At time T2, the system power controller acknowledges the request and CTRL/STAT.CDBGPWRUPACK is set to `0b0`.
3. At time T3, the debugger asserts **nSRST**. The debug domain and non-debug domain are reset at time T3a. The debug domain reset is complete by time T3b.
4. At time T4, the debugger writes `0b1` to CTRL/STAT.CDBGPWRUPREQ. This might occur before or after the debug reset is complete.
5. At time T5, the system power controller acknowledges this request and signals the debug domain reset is complete by setting CTRL/STAT.CDBGPWRUPACK to `0b1`. The debugger can now program the debug domain.
6. At time T6, the debugger releases **nSRST**. The non-debug domain reset completes at time T6a.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B3
The JTAG Debug Port

This chapter describes the implementation of the JTAG Debug Port (JTAG-DP), and in particular, the *Debug Test Access Port* (DBGTAP), the Debug Test Access Port State Machine (DBGTAPSM), and scan chains.

It is only relevant to ADI implementations that use a JTAG-DP. In this case, the JTAG-DP provides the external connection to the DAP, and all interface accesses are made using the scan chains, which are driven by the DBGTAPSM.

This chapter contains the following sections:

- *The scan chain interface* on page B3-87.
- *IR scan chain and IR instructions* on page B3-91.
- *DR scan chain and DR instructions* on page B3-95.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.1 About the JTAG-DP

The JTAG-DP is based on the IEEE 1149.1 Standard for TAP and Boundary Scan Architecture, widely referred to as JTAG. To emphasize that the JTAG-DP is intended for accessing debug components, the naming convention that is used in this document differs from the IEEE 1149.1 naming convention by adding the prefix DBG, as shown in Table B3-1:

Table B3-1 Comparison of IEEE 1149.1 and JTAG-DP naming

IEEE 1149.1 name	JTAG-DP name	JTAG-DP description
TAP	**DBGTAP**	Debug Test Access Port.
TAPSM	**DBGTAPSM**	Debug Test Access Port State Machine.

The signal naming conventions of IEEE 1149.1 are modified in a similar way, for example the IEEE 1149.1 **TDI** signal is named **DBGTDI** on a JTAG Debug Port. See *Physical connection to the JTAG-DP* on page B3-87 for the complete list of the JTAG-DP signal names.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.2 The scan chain interface

When a DAP is implemented with a JTAG-DP, the wire-level interface accesses the APACC scan chain to access debug resources in the system being debugged, or the DPACC scan chain to access information internal to the DAP.

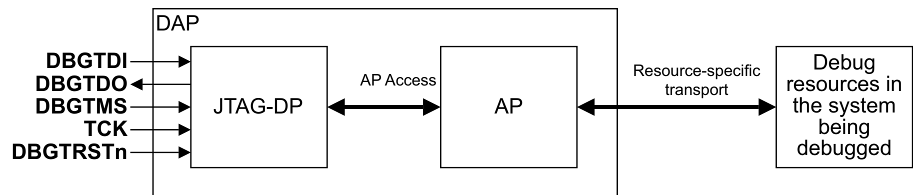
B3.2.1 DAP elements

The DAP requires the following elements to support JTAG scan chains:

- A DBGTAPSM.
- An IR, which selects and controls the available scan chains.
- Various *Data Registers* (DRs), which hold the information that is exposed through the available scan chains, and interface to:
  - — The DP registers in the DAP itself.
  - — The debug registers in the device or debug component being accessed through the DAP.

Figure B3-1 shows how the scan chains provide access to the different levels of the DAP architecture. For more details, see Figure A1-2 on page A1-30.

Physical connection perspective



Scan chains perspective

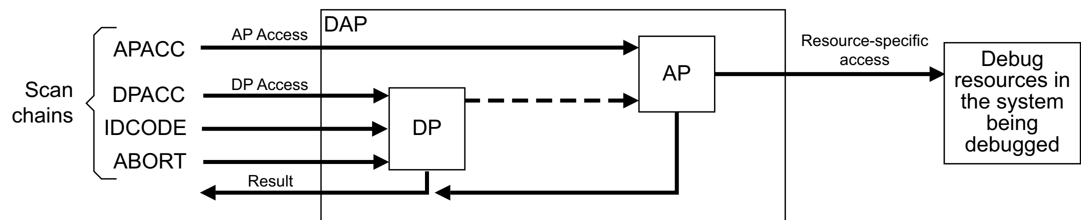


Figure B3-1 JTAG-DP scan chain access to the different levels of the ADI

B3.2.2 Physical connection to the JTAG-DP

The physical connection to the JTAG-DP closely follows the JTAG model. Table B3-2 lists the recommended signals for the JTAG-DP physical connection alongside their equivalent JTAG signal names.

Table B3-2 JTAG-DP signal connections

JTAG-DP signal name	JTAG equivalent signal name	Direction	Required?	Description
DBGTDI	**TDI**	Input	Yes	Debug Data In
DBGTDO	**TDO**	Output	Yes	Debug Data Out

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

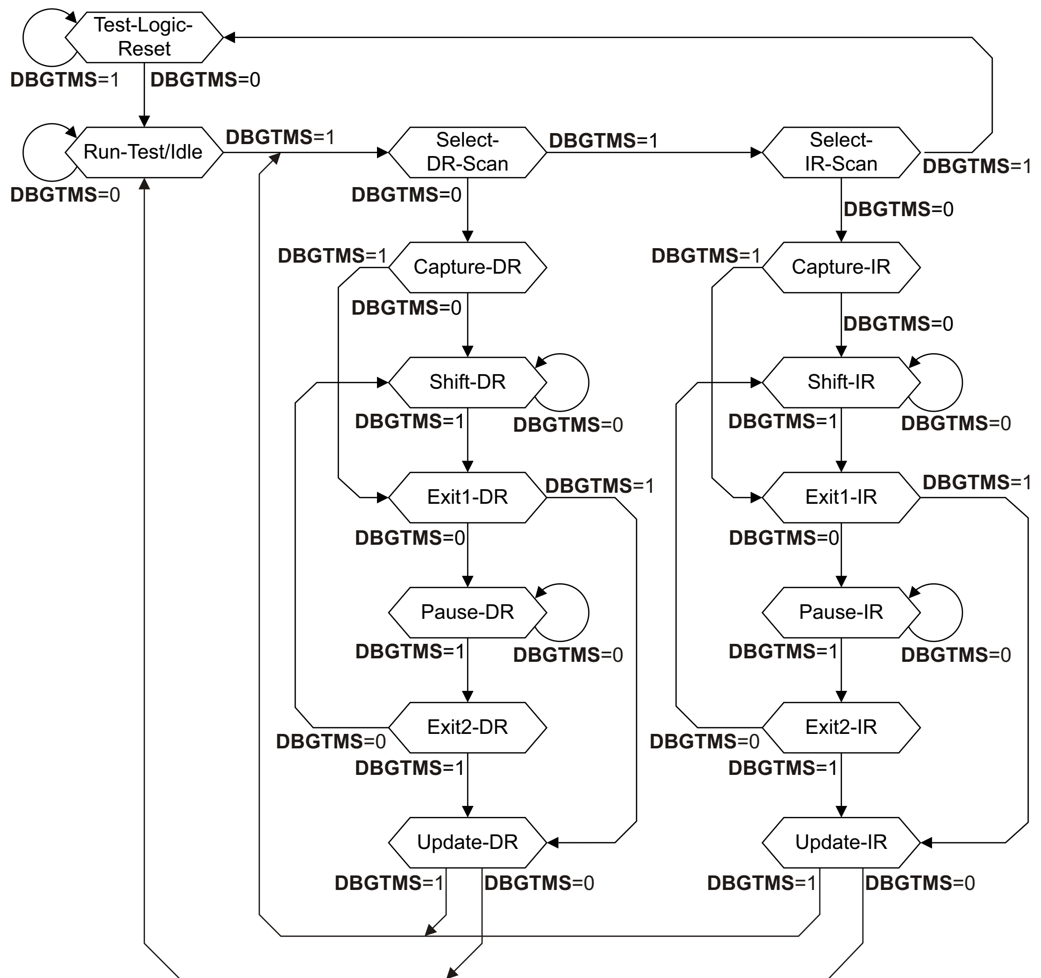
Table B3-2 JTAG-DP signal connections (continued)

JTAG-DP signal name	JTAG equivalent signal name	Direction	Required?	Description
TCK	**TCK**	Input	Yes	Debug Clock
DBGTMS	**TMS**	Input	Yes	Debug Mode Select
DBGTRSTn	**TRST**	Input	Optional	Debug TAP Reset

B3.2.3 The Debug TAP State Machine (DBGTAPSM)

The DBGTAPSM controls the operation of a JTAG-DP. In particular, it controls the scan chain interface that provides the external physical interface to the DAP through the JTAG-DP. It is based closely on the JTAG TAP State Machine, see *IEEE 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture*.

Figure B3-2 shows the state diagram for the DBGTAPSM.



Based on IEEE Std 1149.1-1990. Copyright © 2006 IEEE. All rights reserved.
Note that Arm signal names differ from those used in the IEEE diagram.

Figure B3-2 State diagram for the DBGTAPSM

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

The DBGTAPSM uses the following process:

- The **DBGTDI** signal input is the start of the scan chain and the **DBGTDO** signal output is the end of the scan chain.
- When the DBGTAPSM goes through the Capture-IR state:
  - When using a 4-bit IR, `0b0001` is transferred to the IR scan chain.
  - When using an 8-bit IR, `0b00000001` is transferred to the IR scan chain.
  - The IR scan chain is connected between **DBGTDI** and **DBGTDO**.
- While the DBGTAPSM is in the Shift-IR state, the IR scan chain advances one bit for each rising edge of **TCK**. On the first tick:
  - The LSB of the IR scan chain is output on **DBGTDO**.
  - Bit[1] of the IR scan chain is transferred to bit[0].
  - Bit[2] of the IR scan chain is transferred to bit[1].
  - Similarly, for every other bit *n* of the IR scan chain, bit[*n*] of the scan chain is transferred to bit[*n*-1].
  - The value on **DBGTDI** is transferred to the MSB of the IR scan chain.
- When the DBGTAPSM goes through the Update-IR state, the value that is scanned into the IR scan chain is transferred into the Instruction Register.
- The value that is held in the Instruction Register selects a Data Register, and an associated DR scan chain. When the DBGTAPSM goes through the Capture-DR state, the value of the selected DR is transferred to the selected DR scan chain, which is connected between **DBGTDI** and **DBGTDO**.

  This data is then shifted while the DBGTAPSM is in the Shift-DR state, in the same manner as the IR shift in the Shift-IR state.
- When the DBGTAPSM goes through the Update-DR state, the value that is scanned into the DR scan chain is transferred into the selected Data Register.
- When the DBGTAPSM is in the Run-Test/Idle state, no special actions occur. Debuggers can use this state as a true resting state.

  Note

  This behavior is different from the behavior of previous versions of the ADI that were based on the IEEE JTAG standard. From ADIv5, there is no requirement for debuggers to gate **TCK** to obtain a true rest state.

To ensure that the transfer can be clocked through the JTAG-DP, after going through the Update-DR state the host must do one of the following:

- Start a new JTAG scan operation.
- Put the DBGTAPSM in to Run-Test/Idle, and remain in Run-Test/Idle until a new scan can be started.
- If the host is driving the JTAG clock, continue to clock the JTAG interface for at least eight cycles in Run-Test/Idle. After completing this sequence, the host can stop the clock.

The behavior of the IR and DR scan chains is described in more detail in *IR scan chain and IR instructions* on page B3-91 and *DR scan chain and DR instructions* on page B3-95.

The **DBGTRSTn** signal only resets the DBGTAP state machine and Instruction Register. **DBGTRSTn** asynchronously takes the DBGTAPSM to the Test-Logic-Reset state. As shown in Figure B3-2 on page B3-88, the Test-Logic-Reset state can also be entered synchronously from any state by a sequence of five **TCK** cycles with **DBGTMS** HIGH. However, depending on the initial state of the DBGTAPSM, this transition might take the state machine through one of the Update states, with the resulting side effects.

The reset behavior of the registers is as follows:

- The DP registers are only reset on a powerup reset.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- The AP registers are reset on a powerup reset, and also by the Debug Reset Control described in *Debug reset control behavior* on page B2-82.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.3 IR scan chain and IR instructions

This section describes the JTAG-DP IR, accessed through the IR scan chain.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

B3.3.1 Required IR instructions

As described in *The Debug TAP State Machine (DBGTAPSM)* on page B3-88, the JTAG-DP transfers an instruction into the IR. This instruction determines the DR that the JTAG-DP DR maps onto, as described in *DR scan chain and DR instructions* on page B3-95.

The width of the IR is IMPLEMENTATION DEFINED, and can be 4 or 8 bits.

The standard IR instructions, which are required for all JTAG-DP implementations, are listed in Table B3-3, and recommended IMPLEMENTATION DEFINED extensions to this instruction set are described in *IMPLEMENTATION DEFINED extensions to the IR instruction set* on page B3-93.

Unused IR instruction values are reserved and select the BYPASS register.

Table B3-3 Standard IR instructions

4-bit IR instruction value	8-bit IR instruction value	Data register	DR scan length	Notes
`0b0xxx`	`0b0xxxxxxx`	-	-	*IMPLEMENTATION DEFINED extensions to the IR instruction set* on page B3-93.
-	`0b10000000-0b11110111`	-	-	Reserved.
`0b1000`	`0b11111000`	ABORT	35	-
`0b1001`	`0b11111001`	-	-	Reserved.
`0b1010`	`0b11111010`	DPACC	35	See *DPACC and APACC, JTAG-DP DP, and AP Access registers* on page B3-98.
`0b1011`	`0b11111011`	APACC	35	
`0b110x`	`0b1111110x`	-	-	Reserved.
`0b1110`	`0b11111110`	IDCODE	32	-
`0b1111`	`0b11111111`	BYPASS	1	Required by JTAG specification.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B3.3.2 IMPLEMENTATION DEFINED extensions to the IR instruction set

The 4-bit IR instructions `0b0000` to `0b0111` and the 8-bit instructions `0b00000000` to `0b01111111` are reserved for IMPLEMENTATION DEFINED extensions to the DAP.

These instructions can be used for accessing a boundary scan register, for IEEE 1149.1 compliance, as shown in Table B3-4. All these instructions select the boundary scan data register.

Note

This extension describes only boundary scan instructions that are described by IEEE 1149.1-2001. Later editions of IEEE 1149.1 define additional instructions.

Arm recommends that:

- Separate JTAG TAPs are used for boundary scan and debug.
- The instructions that are listed in Table B3-4 are not implemented.

If the IR register is set to an IR instruction value that is not implemented, or reserved, then the BYPASS register is selected.

Table B3-4 Recommended IMPLEMENTATION DEFINED IR instructions for IEEE 1149.1-compliance

4-bit IR instruction value	8-bit IR instruction value	Instruction	Required by IEEE 1149.1?
`0b0000`	`0b00000000`	EXTEST	See note in main text.
`0b0001`	`0b00000001`	SAMPLE	Yes
`0b0010`	`0b00000010`	PRELOAD	Yes
`0b0100`	`0b00000100`	INTEST	No
`0b0101`	`0b00000101`	CLAMP	No
`0b0110`	`0b00000110`	HIGHZ	No

If you require a boundary scan implementation, you must implement the instructions that are required by IEEE 1149.1. The other IR instruction values that are listed in Table B3-4 are reserved encodings that must be used if that function is implemented in the boundary scan. If implemented, these instructions must behave as required by the IEEE 1149.1 specification. If not implemented, they select the BYPASS register.

Note

EXTEST instruction The original revision of the IEEE 1149.1 specification, 1149.1-1990, requires that instruction `{000..0}` is EXTEST. However, in more recent editions this requirement is removed and the specification recommends that instruction `{000..0}` is reserved. See the IEEE specification for more details.

The IEEE 1149.1 specification also defines the IDCODE and BYPASS instructions, which are included in Table B3-3 on page B3-92.

B3.3.3 IR, JTAG-DP Instruction Register

Purpose

Holds the current DAP Controller instruction.

Configurations

This register is mandatory in the IEEE 1149.1 standard.

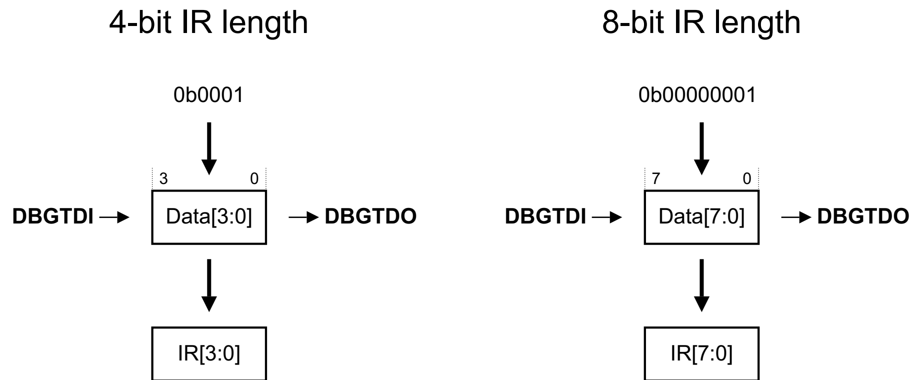
Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Attributes

IMPLEMENTATION DEFINED, a 4-bit or 8-bit register.

Operation

The operation of the IR register is shown in the following figure:



When in Shift-IR state, the shift section of the IR is selected as the serial path between **DBGTDI** and **DBGTDO**. At the Capture-IR state, the binary value `0b0001` for 4-bit instructions, or `0b00000001` for 8-bit instructions, is loaded into this shift section. This value is shifted out, least significant bit first, during Shift-IR, while a new instruction is shifted in, least significant bit first:

- At the Update-IR state, the value in the shift section is loaded into the IR and becomes the current instruction.
- In the Test-Logic-Reset state, IDCODE becomes the current instruction.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.4 DR scan chain and DR instructions

The DR scan chain is associated with the DR registers:

- The BYPASS and IDCODE registers, as defined by the IEEE 1149.1 standard.
- The DPACC and APACC Access registers, xPACC.
- An ABORT register, to abort a transaction.

This section describes each of these registers.

The instruction in the IR register determines which of these scan chains is connected to the **DBGTDI** and **DBGTDO** signals, see *IR scan chain and IR instructions* on page B3-91.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.4.1 ABORT, JTAG-DP Abort register

Purpose

Access the ABORT register in the DP, to force an AP abort.

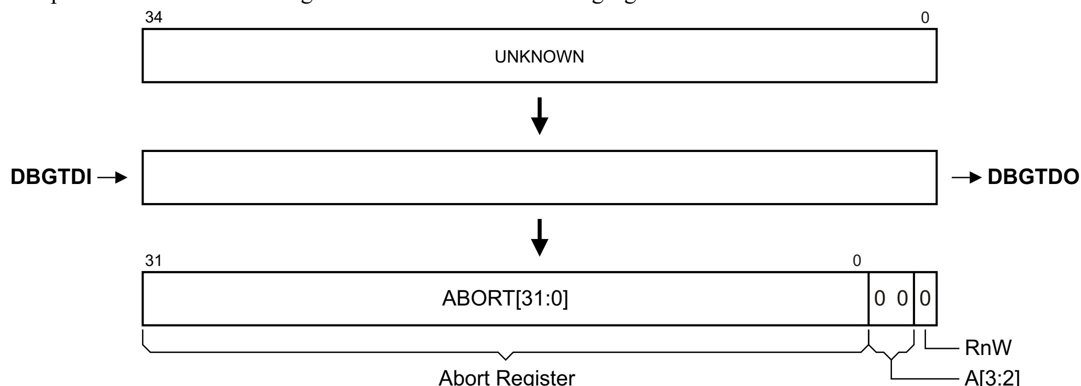
This implementation is the JTAG-DP implementation of the ABORT register.

Attributes

A 35-bit register.

Operation

The operation of the ABORT register is shown in the following figure:



When the ABORT instruction is the current instruction in the IR, the serial path between **DBGTDI** and **DBGTDO** is connected to a 35-bit scan chain that accesses the ABORT register.

In DPv0, the effect of writing a value other than `0x00000001` to the ABORT register is UNPREDICTABLE, which means that, in DPv0, the debugger must scan the value `0x000000008` into this scan chain. For more information, see *ABORT, Abort register* on page B2-53.

B3.4.2 BYPASS, JTAG-DP Bypass register

Purpose

Bypasses the device, by providing a direct path between **DBGTDI** and **DBGTDO**.

Configurations

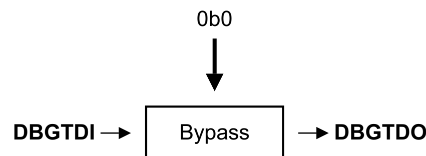
This register is mandatory in the IEEE 1149.1 standard.

Attributes

A 1-bit register.

Operation

The operation of the BYPASS register is shown in the following figure:



When the BYPASS instruction is the current instruction in the IR:

- In the Shift-DR state, data is transferred from **DBGTDI** to **DBGTDO** with a delay of one **TCK** cycle.
- In the Capture-DR state, a logic 0 is loaded into this register.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- In the Update-DR state, nothing happens. The shifted-in data is ignored.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

B3.4.3 DPACC and APACC, JTAG-DP DP, and AP Access registers

The DPACC and APACC scan chains have the same format.

Purpose

DPACC and APACC are used to read from and write to DP or AP registers.

The DPACC scan chain uses A[3:2], SELECT.DPBANKSEL and RnW to determine the address of the DP register to be accessed, as summarized in Table B3-5. For detailed information about addressing JTAG-DP registers, see *DP architecture versions* on page B2-48.

Note

The DP register ABORT is accessed through the ABORT instruction.

Table B3-5 JTAG-DP Register access summary.

Register	Access	Address (A[a],SELECT.DPBANKSEL)		
		DPv0	DPv1	DPv2
CTRL/STAT	RW	`0x4`, -	`0x4`, `0x0`	`0x4`, `0x0`
DLCR	RW	-	`0x4`, `0x1`	`0x4`, `0x1`
DLPIDR	RO	-	-	`0x4`, `0x3`
DPIDR	RO	-	`0x0`, x	`0x0`, x
EVENTSTAT	RO	-	-	`0x4`, `0x4`
RDBUFF	RO	`0xC`, -	`0xC`, x	`0xC`, x
SELECT	WO[b]	`0x8`, -	`0x8`, x	`0x8`, x
TARGETID	RO	-	-	`0x4`, `0x2`

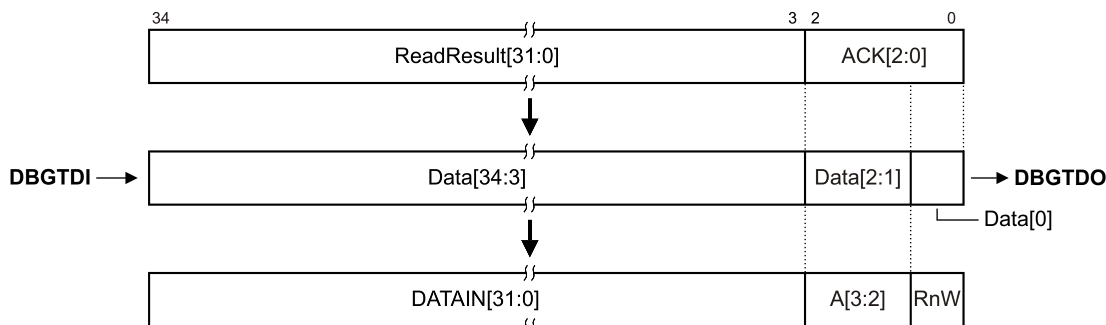
a. Bits [1:0] of the address are always `0b00`.
b. RW for DPv0

- *MEM-AP Programmers' Model* on page C2-167 for details of accessing MEM-AP registers.
- *JTAG-AP register summary* on page C3-205 for details of accessing JTAG-AP registers.

Attributes DPACC and APACC are 35-bit registers.

Operation

The operation of the DPACC and APACC registers is shown in the following figure:



Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

When the DPACC or APACC instruction is the current instruction in the IR, the shift section of the DP Access register or AP Access register is selected as the serial path between **DBGTDI** and **DBGTDO**:

- In the Capture-DR state, the result of the previous transaction, if any, is returned, together with a 3-bit ACK response. The ACK responses that are implemented are summarized in Table B3-6.

  Table B3-6 DPACC and APACC ACK responses

Response	ACK[2:0] encoding	See:
OK/FAULT	`0b010`	*OK/FAULT response to a DPACC or APACC access*
WAIT	`0b001`	*WAIT response to a DPACC or APACC access* on page B3-100

  All other ACK encodings are reserved.

- In the Shift-DR state, this data is shifted out, least significant bit first. The first three bits of data that are shifted out are ACK[2:0].

  As the returned data is shifted out to **DBGTDO**, new data is shifted in from **DBGTDI**, as described in *OK/FAULT response to a DPACC or APACC access*.

- Operation in the Update-DR depends on whether the ACK[2:0] response was OK/FAULT or WAIT. The two cases are described in:
  — *OK/FAULT response to a DPACC or APACC access*.
  — *WAIT response to a DPACC or APACC access* on page B3-100.

OK/FAULT response to a DPACC or APACC access

If the response indicated by ACK[2:0] is OK/FAULT, the previous transaction has completed. The response code does not show whether the transaction completed successfully or failed. You must read the CTRL/STAT register to find whether the transaction was successful:

- If the previous instruction in the IR was not one of DPACC, APACC, or BYPASS, the captured ReadResult[31:0] is UNKNOWN, and if Data[34:3] is shifted out it must be discarded.
- If the previous transaction was a read that completed successfully, the captured ReadResult[31:0] is the requested register value. This result is shifted out as Data[34:3].
- If the previous transaction was a write, or a read that did not complete successfully, the captured ReadResult[31:0] is UNKNOWN, and if Data[34:3] is shifted out it must be discarded.

An OK/FAULT response is followed by an Update-DR operation to fulfill the read or write request that is formed by the values that were shifted into the scan chain:

- **DBGTDI** and **DBGTDO** connect to the scan chain corresponding to the current IR instruction, and the specified address is used to select a register.
- For write requests, corresponding to RnW having a value of `0b0`, the value in DATAIN[31:0] is written to the selected register.
- For read requests, corresponding to RnW having a value of `0b1`, the value in DATAIN[31:0] is IGNORED. Another scan is required to obtain the read data.

Register accesses can be pipelined, because a single DPACC or APACC scan can return the result of the previous read operation at the same time as shifting in a request for another register access. At the end of a sequence of pipelined register reads, you can read the DP RDBUFF register to shift out the result of the final register read.

Reading the DP RDBUFF register has no effect on the operation of the DBGTAPSM. For details about returning the result from a previous DPACC or APACC scan, see section *Target response summary* on page B3-101.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

If the current IR instruction is APACC, an AP access is requested:

- If any sticky flag in the DP CTRL/STAT register is `0b1`, the transaction is discarded. The next scan returns an OK/FAULT response. For more information, see *Sticky flags and DP error responses* on page B1-41.
- If pushed-compare or pushed-verify operations are enabled, the scanned-in value of RnW must be `0b0`, otherwise behavior is UNPREDICTABLE. On Update-DR, a read request is issued, and the returned value is compared against DATAIN[31:0]. The CTRL/STAT.STICKYCMP flag is updated based on this comparison. For more information, see *Pushed-compare and pushed-verify operations* on page B1-44.

  Pushed operations are enabled using the CTRL/STAT.TRNMODE field.
- The AP access does not complete until the AP signals it as completed. For example, if you access a MEM-AP, the AP access might cause an access to a memory system connected to the MEM-AP. In this case, the AP access does not complete until the memory system signals to the MEM-AP that the memory access has completed.

WAIT response to a DPACC or APACC access

A WAIT response indicates that the previous transaction has not completed. Normally, after receiving a WAIT response the host retries the DPACC or APACC access.

Note

The previous transaction might be either a DP or an AP access. DP accesses are stalled, by returning WAIT, until any previous AP transaction has completed.

Normally, if software detects a WAIT response, it retries the same transfer, which enables the protocol to process data as quickly as possible. However, if the software has retried a transfer several times, and has waited long enough for a slow interconnect and memory system to respond, it might write to the ABORT register to cancel the operation. This action signals to the active AP that it must terminate the transfer that it is attempting, to permit access to other parts of the debug system. An AP might not be able to terminate a transfer on its SoC interface. However, on receiving an ABORT, the AP must free its interface to the DP.

No request is generated at the Update-DR state, and the shifted-in data is discarded. The captured value of ReadResult[31:0] is UNKNOWN.

Sticky overrun behavior on DPACC and APACC accesses

At the Capture-DR state, if the previous transaction has not completed, a WAIT response is generated. In this case, if the Overrun Detect flag, CTRL/STAT.ORUNDETECT is `0b1`, the Sticky Overrun flag, CTRL/STAT.STICKYORUN, is set to `0b1`.

As long as the previous transaction is not completed, subsequent scans also receive a WAIT response.

When the previous transaction has completed, any additional APACC transactions are abandoned and scans respond with an OK/FAULT response. However, DP registers can be accessed. In particular, the CTRL/STAT register can be accessed, to confirm that the Sticky Overrun flag is `0b1`, and to clear the flag to `0b0` after gathering any required information about the overrun condition. See *Sticky flags and DP error responses* on page B1-41 for more information.

Minimum response times

As explained in *OK/FAULT response to a DPACC or APACC access* on page B3-99, a DP or AP register access is initiated at the Update-DR state of one DPACC or APACC access, and the result of the access is returned at the Capture-DR state of the following DPACC or APACC access. If the requested register access has not completed, however, the second access generates a WAIT response.

The timing between the Update-DR state and the Capture-DR state is defined in terms of **TCK** cycles. Referring to Figure B3-2 on page B3-88, there are two paths from the Update-DR state, where the register access is initiated, to the Capture-DR state, where the response is captured:

- A direct path through Select-DR-Scan.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- A path through Run-Test/Idle and Select-DR-Scan.

If the second path is followed, the state machine can spend a variable number of **TCK** cycles in the Run-Test/Idle state, which in turn varies the number of **TCK** cycles between the Update-DR and Capture-DR states.

A JTAG-DP implementation might impose an IMPLEMENTATION DEFINED lower limit on the number of **TCK** cycles between the Update-DR and Capture-DR states, and always generate an immediate WAIT response if Capture-DR is entered before this limit has expired. Although any debugger must be able to recover successfully from any WAIT response, Arm recommends that debuggers must be able to adapt to any IMPLEMENTATION DEFINED limit.

Note

Accessing AP registers or debug resources in connected device through an AP can be subjected to other variable response delays in the system. A debugger that can adapt to these delays and avoid wasting WAIT scans operates more efficiently and provides higher maximum data throughput.

Target response summary

As described in *OK/FAULT response to a DPACC or APACC access* on page B3-99, a DP or AP register access is initiated at the Update-DR state of one DPACC or APACC access, and the result of the access is returned at the Capture-DR state of the following DPACC or APACC access. The target responses, at the Capture-DR state, for every possible DPACC and APACC access in the previous scan, are summarized in:

- Table B3-7, for cases where the previous scan was a DPACC access.
- Table B3-8 on page B3-102, for cases where the previous scan was an APACC access.

Note

The target responses that are shown in Table B3-7 are independent of the operation being performed in the current DPACC or APACC scan. In this table, Read result is the data that is shifted out as Data[34:3], and ACK is decoded from the data that is shifted out as Data[2:0].

Table B3-7 JTAG-DP target response summary, when previous scan[a] was a DPACC access

Previous scan[a], at Update-DR state			Current scan, at Capture-DR state			Notes
Access	**A[b]**	**Sticky?[c]**	**AP state[d]**	**Read result**	**ACK**	
X	X	X	Busy	UNKNOWN	WAIT	If the Overrun Detect flag is `0b1`, this access and response sequence causes the Sticky Overrun flag to be set to `0b1`. See *CTRL/STAT, Control/Status register* on page B2-55.
R	`0b00`	X	Not Busy	UNKNOWN	OK/ FAULT	The return value depends on the DP version: **DPv1, DPv2** Returns the value of DPIDR. **DPv0** Returns UNKNOWN value.
	`0b01`			CTRL/STAT		Returns CTRL/STAT value.
	`0b10`			SELECT		Returns SELECT value.
	`0b11`			`0x00000000`		Returns RDBUFF value, always zero.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

Table B3-7 JTAG-DP target response summary, when previous scan[a] was a DPACC access (continued)

Previous scan[a], at Update-DR state			Current scan, at Capture-DR state			Notes
Access	A[b]	Sticky?[c]	AP state[d]	Read result	ACK	
W	0b00	X	Not Busy	UNKNOWN	OK/ FAULT	The result depends on the DP version: **DPv2** Write is ignored, returns RES0. **DPv0, DPv1** Behavior is UNPREDICTABLE.
	0b01					Value has been written to CTRL/STAT.
	0b10					Value has been written to SELECT.
	0b11					Writes to RDBUFF always ignored.

a. The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates that are made following a WAIT response are discarded.
b. A[3:2] in the DPACC access.
c. The *Sticky?* column indicates whether any Sticky flag was 0b1 in the DP CTRL/STAT register.
d. The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.

Table B3-8 JTAG-DP target response summary, when previous scan[a] was an APACC access

Previous scan[a], at Update-DR state			Current scan, at Capture-DR state			Notes
Access	A[b]	Sticky?[c]	AP state[d]	Read result	ACK	
X	X	X	Busy	UNKNOWN	WAIT	If the Overrun Detect flag is 0b1, this access and response sequence causes the Sticky Overrun flag to be set to 0b1. See *CTRL/STAT, Control/Status register* on page B2-55.
R	X	No	Ready	See Notes	OK/ FAULT	Returns the value of the AP register that was addressed on the previous scan. If pushed-verify or pushed-compare is implemented and enabled, the behavior is UNPREDICTABLE.
			Error	UNKNOWN		Sticky Error flag is set to 0b1.
W	X	No	Ready	UNKNOWN	OK/ FAULT	The data that was captured at the previous scan has been written to the requested AP register. If pushed-verify or pushed-compare is implemented and enabled, the previous pushed transaction might have set the Sticky Compare flag to 0b1, see *Pushed-compare and pushed-verify operations* on page B1-44.
			Error	UNKNOWN		Sticky Error flag is set to 0b1.
X	X	Yes	Not busy	UNKNOWN	OK/ FAULT	Previous transaction was discarded.

a. The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates that are made following a WAIT response are discarded.
b. A[3:2] in the APACC access.
c. The *Sticky?* column indicates whether any Sticky flag was 0b1 in the DP CTRL/STAT register.
d. The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Host response summary

The ACK column, for the Current scan, at Capture-DR state section of Table B3-7 on page B3-101 and Table B3-8 on page B3-102, shows the responses that the host might receive after initiating a DPACC or APACC access. Table B3-9 indicates the normal action of a host in response to each of these ACKs.

Table B3-9 Summary of JTAG-DP host responses

Access type	ACK from target	Suggested host action in response to ACK
Read	OK/FAULT	Capture read data.
Write	OK/FAULT	No action required.
Read or Write	WAIT	Repeat the same access until either an OK/FAULT ACK is received or the wait timeout is reached. If necessary, use the ABORT register to enable access to the AP.
Read or Write	Invalid ACK	Assume that a target or line error has occurred and treat as a fatal error.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B3.4.4 IDCODE, the JTAG TAP ID register

Purpose

JTAG-DP TAP identification. The IDCODE value enables a debugger to identify the Debug Port to which it is connected. JTAG-DP implementations have different IDCODE values, so that a debugger can distinguish between them.

When the IDCODE instruction is the current instruction in the IR, the shift section of the Device ID Code register is selected as the serial path between **DBGTDI** and **DBGTDO**:

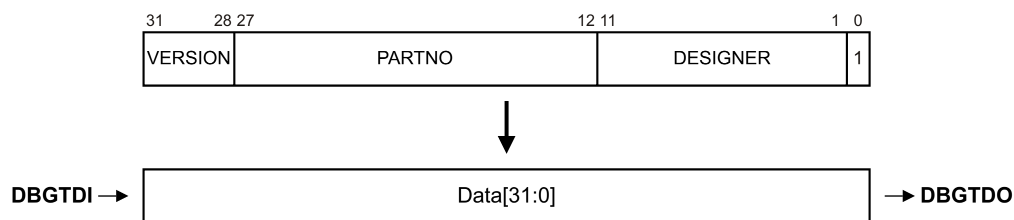
- In the Capture-DR state, the 32-bit device ID code is loaded into this shift section.
- In the Shift-DR state, this data is shifted out, least significant bit first.
- Nothing happens at the Update-DR state. The shifted-in data is ignored.

Attributes

A 32-bit register.

Field descriptions

The IDCODE bit assignments and operating mode are:



VERSION, bits[31:28]

Version code. The meaning of this field is IMPLEMENTATION DEFINED.

PARTNO, bits[27:12]

Part Number for the DP TAP. This value is provided by the designer of the DP TAP and must not be changed.

DESIGNER, bits[11:1]

The Designer ID is an 11-bit JEDEC code that identifies the designer of the JTAG-DP TAP. It is formed from the JEDEC JEP106 continuation code and identity code as shown in Table B3-10.

Table B3-10 JEDEC JEP106 manufacturer ID code, with Arm values

JEP106 field	Width (bits)	Bits in IDCODE	Arm registered value
Continuation code	4	Bits[11:8]	`0b0100` (`0x4`)
Identity code	7	Bits[7:1]	`0b0111011` (`0x3B`)

JEDEC codes are assigned by the JEDEC Solid State Technology Association, see *JEP106, Standard Manufacturer's Identification Code*.

Normally, this field identifies the designer of the ADIv5 implementation, rather than the system architect or the device manufacturer. If the DAP is used for boundary scan, however, the field must conform to the JEDEC Manufacturer ID assigned to the manufacturer of the device.

The Arm default value for this field is `0x23B`. Other designers must use their own JEDEC assigned code.

Bit[0]

RAO.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B4
The Serial Wire Debug Port

This chapter describes the implementation of the *Serial Wire Debug Port* (SW-DP), including the SWD interface. It is only relevant if the ADI implementation uses an SW-DP. In this case, the SW-DP provides the external connection to the debug interface and all interface accesses are made using the SWD protocol summarized in this chapter.

Note

The Arm SWD interface is a synchronous serial interface. This specification does not describe the physical characteristics of the SWD interface, such as signal timings.

This chapter contains the following sections:

- *About the SWD protocol* on page B4-106.
- *SWD protocol operation* on page B4-110.
- *SWD interface* on page B4-122.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B4.1 About the SWD protocol

This section provides general information about the Arm SWD protocol. It contains the following sections:

- *Basic operation*.
- *SWD protocol versions*.
- *Line turnaround*.
- *Idle cycles*.
- *Bit order*.
- *Parity*.
- *Limitations of multi-drop*.

B4.1.1 Basic operation

The Arm SWD interface uses a single bidirectional data connection and a separate clock to transfer data synchronously. An operation on the wire consists of two or three phases:

Packet request

The external host debugger issues a request to the DP. The DP is the target of the request.

Acknowledge response

The target sends an acknowledge response to the host.

Data transfer phase

This phase is only present when either:

- A data read or data write request is followed by a valid (OK) acknowledge response.
- The CTRL/STAT.ORUNDETECT flag is `0b1`.

The data transfer is one of:

- Target to host, following a read request (RDATA).
- Host to target, following a write request (WDATA).

Note

If the CTRL/STAT.ORUNDETECT bit is `0b1`, a data transfer phase is required on all responses, including WAIT and FAULT. For more information, see *Sticky overrun behavior* on page B4-115.

When the SW-DP receives a packet request from the debug host, it must respond immediately by entering the acknowledge phase. There is a turnaround period between these phases, as they are in different directions. If a data phase is required, it follows immediately after the acknowledge phase.

For a write request, there is a turnaround period between the acknowledge phase and the WDATA data transfer phase. Following the WDATA data transfer phase the host continues to drive the wire. There is no additional turnaround period.

For a read request, there is no turnaround period between the acknowledge phase and the data transfer phase. There is a turnaround period following the RDATA data transfer phase, following which the host drives the wire.

To ensure that the transfer can be clocked through the SW-DP, after the data transfer phase the host must do one of the following:

- Immediately start a new SWD operation with the start bit of a new packet request.
- Continue to drive the SWD interface with idle cycles until the host starts a new SWD operation.
- If the host is driving the SWD clock, continue to clock the SWD interface with at least 8 idle cycles. After completing this sequence, the host can stop the clock.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B4.1.2 SWD protocol versions

SWD protocol version 1 is a point-to-point architecture, supporting connection between a single host and a single device. It permits connection to multiple devices by providing extra connections from the host, which has several disadvantages:

- It complicates the physical connection standard, by having variants with different numbers of connections.
- It increases the number of pins that are required for the connector on the device PCB, which is unacceptable where size is a limiting factor.
- It increases the number of pins that are required on a package with multiple dies inside.
- It makes it difficult to integrate multiple platforms that are accessed by the SWD protocol into the same chip.

Techniques to solve this require connections that are shared between multiple Serial Wire devices. These connections are detrimental to the maximum speed of operation, but in many situations they provide an acceptable trade-off.

SWD protocol version 2 is a multi-drop architecture that:

- Enables a two-wire host connection to communicate simultaneously with multiple devices.
- Permits an effectively unlimited number of devices to be connected simultaneously, subject to electrical constraints.
- Is largely backwards-compatible, because provision for multi-drop support in a device does not break point-to-point compatibility with existing host equipment that does not support the multi-drop extensions.
- Permits a device to power down completely, while the device is not selected.
- Prevents multiple devices from driving the wire simultaneously, and continues to support the wire being actively driven both HIGH and LOW, maintaining a high maximum clock speed.
- Permits multi-drop connections incorporating devices that do not implement the SWD protocol.

Note

SWD protocol version 2 requires the implementation of dormant state, which can limit its compatibility with SWD version 1:

- For an SWJ-DP implementation, JTAG is selected on a powerup reset. Selecting SWD bypasses the dormant state, and subsequent operation is compatible with SWD protocol version 1.
- For an SW-DP implementation of SWD protocol version 2, the dormant state is selected on a powerup reset, meaning the start-up state differs from a start-up with SWD protocol version 1. After SWD operation is selected, operation is compatible with SWD protocol version 1.

B4.1.3 Line turnaround

To prevent contention, a turnaround period is required when the device driving the wire changes. For the turnaround period, neither the host nor the target drives the wire, and the state of the wire is undefined. See also *Line pull-up* on page B4-122.

Note

The line turnaround period can provide for pad delays when using a high sample clock frequency.

The length of the turnaround period is controlled by DLCR.TURNROUND. The default setting is a turnaround period of one clock cycle.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved. *Non-Confidential*

B4.1.4 Idle cycles

After completing a transaction, the host must either insert idle cycles or continue immediately with the start bit of a new transaction.

The host clocks the SWD interface with the line LOW to insert idle cycles.

B4.1.5 Bit order

All data values in SWD operations are transferred LSB first.

For example, the OK response of `0b001` appears on the wire as 1, followed by 0, followed by 0, as shown in Figure B4-1 on page B4-111 and Figure B4-2 on page B4-112.

B4.1.6 Parity

A simple parity check is applied to all packet request and data transfer phases. Even parity is used:

Packet requests

- If the number of bits with a value of `0b1` is odd, the parity bit is set to `0b1`.
- If the number of bits with a value of `0b1` is even, the parity bit is set to `0b0`.

Data transfers (WDATA and RDATA)

The parity check is made over the 32 data bits WDATA[0:31] or RDATA[0:31]:

- If the number of bits with a value of `0b1` is odd, the parity bit is set to `0b1`.
- If the number of bits with a value of `0b1` is even, the parity bit is set to `0b0`.

The packet request parity bit is shown in each of the diagrams in this section, from Figure B4-1 on page B4-111 to Figure B4-7 on page B4-116. It appears on the wire immediately after the A[2:3] bits. A parity error in the packet request is detected by the SW-DP, which responds with a protocol error. See *Protocol error response* on page B4-114.

The WDATA parity bit is shown in Figure B4-1 on page B4-111 and in Figure B4-7 on page B4-116. It appears on the wire immediately after the WDATA[31] bit. A parity error in the WDATA data transfer phase is detected by the SW-DP and, other than writes to TARGETSEL, recorded in CTRL/STAT.WDATAERR. If overrun detection is enabled, it is IMPLEMENTATION DEFINED whether CTRL/STAT.STICKYORUN is set to `0b1`. A parity error in a write to TARGETSEL deselects the target.

If a SWD write parity error occurs, the transaction is discarded and the register is not updated. This applies to both DP and AP writes.

The RDATA parity bit is shown in Figure B4-2 on page B4-112. It appears on the wire immediately after the RDATA[31] bit. The debugger must check for parity errors in the RDATA data transfer phase and retry the read if necessary.

Note

The ACK[0:2] bits are never included in the parity calculation. Debuggers must remember this principle when parity checking the data from a read operation, when the debugger receives a continuous stream of 36 bits, as shown in Figure B4-2 on page B4-112:

- Bit 35 is the parity bit.
- Bits 3-34 are RDATA[0:31].
- Bits 0-2 are ACK[0:2].

The parity check must be applied to bits 3-34 of this block of data and the result must be compared with bit[35], the parity bit.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B4.1.7 Limitations of multi-drop

This section describes the configuration and auto-detection limitations of a multi-drop SWD system.

System configuration

Each device must be configured with a unique target ID, which includes a 4-bit instance ID, to differentiate between otherwise identical targets. The 4-bit ID places a limit of 16 such targets in any system. To ensure the target IDs do not conflict, identical devices must be configured before they are connected.

Auto-detection of the target

It is not possible to interrogate a multi-drop SWD system that includes multiple devices to establish which devices are connected. For a target with multiple devices, because all devices are selected on coming out of a line reset, no communication with a device is possible without prior selection of that target using its target ID. Therefore, connection to a multi-drop SWD system that includes multiple devices requires that either:

- The host has prior knowledge of the devices in the system and is configured before target connection.
- The host attempts auto-detection by issuing a target select command for each of the devices it has been configured to support. While auto-detection is likely to involve many target select commands, it must be possible to iterate through all the supported devices in a reasonable time from the viewpoint of a user of the debug tools.

For this reason, debug tools cannot connect seamlessly to targets in a multi-drop SWD system that they have never seen before. If the debug tools can be provided with the target ID of such targets, however, the contents of the target can be auto-detected as normal.

To protect against multiple selected devices all driving the line simultaneously, the SWD protocol version 2 requires:

- For multi-drop SWJ-DP, the JTAG connection is selected out of powerup reset. JTAG does not drive the line. See Chapter B5 *The Serial Wire/JTAG Debug Port*.
- For multi-drop SW-DP, the DP is in the dormant state out of powerup reset. See *Dormant operation* on page B5-131.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B4.2 SWD protocol operation

This section gives an overview of the bidirectional operation of the protocol. This section illustrates each of the possible sequences of operations on the SWD interface data connection.

Note

The diagrams in this section are included to show the operation of the SWD protocol. The diagrams do not show timing diagrams for the protocol.

- *Successful write operation (OK response)* on page B4-111.
- *Successful read operation (OK response)* on page B4-112.
- *WAIT response to read or write operation request* on page B4-113.
- *FAULT response to read or write operation request* on page B4-113.
- *Protocol error response* on page B4-114.
- *Sticky overrun behavior* on page B4-115.
- *SW-DP write buffering* on page B4-116.

The illustrations of the different possible operations use the following terms:

Start A single start bit, with value `0b1`.

APnDP A single bit, indicating whether the DP or the AP Access register is to be accessed. This bit is `0b0` for a DPACC access, or `0b1` for an APACC access.

RnW A single bit, indicating whether the access is a read or a write. This bit is `0b0` for a write access, or `0b1` for a read access.

A[2:3] Two bits, giving the A[3:2] address field for the DP or AP register Address:

- For a DPACC access, the register being addressed depends on the A[3:2] value and, if A[3:2]==`0b01`, the value that is held in SELECT. DPBANKSEL. For details, see:
  - *DP architecture version 1 (DPv1) address map* on page B2-50
  - *DP architecture version 2 (DPv2) address map* on page B2-51.
- For an APACC access, the register being addressed depends on the A[3:2] value and the value that is held in SELECT.{APSEL,APBANKSEL}. For details about addressing, see:
  - *MEM-AP Programmers' Model* on page C2-167 for accesses to a MEM-AP register
  - *JTAG-AP register summary* on page C3-205 for accesses to a JTAG-AP register.

Note

The A[3:2] value is transmitted Least Significant Bit (LSB) first on the wire, which is why it appears as A[2:3] on the diagrams.

Parity A single parity bit for the preceding packet. See *Parity* on page B4-108.

Stop A single stop bit. In the synchronous SWD protocol, this bit is always `0b0`.

Park A single bit. The host must drive the Park bit HIGH to park the line before tristating it for the turnaround period, to ensure that the line is read as HIGH by the target. This is required because the pull-up on the SWD interface is weak. The target reads this bit as `0b1`.

Trn Turnaround. See *Line turnaround* on page B4-107.

Note

All the examples that are given in this chapter show the default turnaround period of one cycle.

ACK[0:2] A 3-bit target-to-host response.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

The ACK value is transmitted LSB-first on the wire and so appears as ACK[0:2] on the diagrams.

Table B4-1 SWD ACK responses summary table

ACK[0:2] encoding	Response	See:
`0b100`	OK	*Successful write operation (OK response)*, *Successful read operation (OK response)* on page B4-112.
`0b010`	WAIT	*WAIT response to read or write operation request* on page B4-113.
`0b001`	FAULT	*FAULT response to read or write operation request* on page B4-113.

WDATA[0:31]

32 bits of write data, from host to target.

Note

The WDATA value is transmitted LSB-first on the wire and so appears as WDATA[0:31] on the diagrams.

RDATA[0:31]

32 bits of read data, from target to host.

Note

The RDATA value is transmitted LSB-first on the wire and so appears as RDATA[0:31] on the diagrams.

B4.2.1 Successful write operation (OK response)

On receiving a write packet request, if the SW-DP is ready for the WDATA data transfer phase, and there is no error condition, it issues an OK response. This response is indicated by a response value of `0b001`.

This response does not apply to writes to TARGETSEL. See *Connection and line reset sequence* on page B4-122.

Therefore, a successful write operation consists of three phases:

1. An 8-bit write packet request, from the host to the target.
2. A 3-bit OK acknowledge response, from the target to the host.
3. A 33-bit WDATA data transfer phase, from the host to the target.

By default, there are single-cycle turnaround periods between each of these phases. See *Line turnaround* on page B4-107 for more information.

A successful write operation is shown in Figure B4-1.

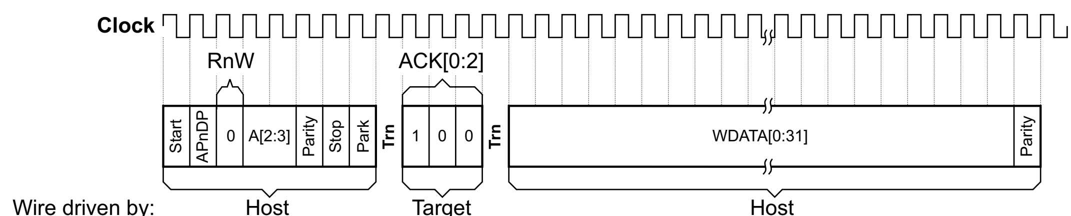


Figure B4-1 SWD successful write operation

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The host must start transferring the write data immediately after receiving the OK acknowledge response from the target. This behavior is the same for writing to the DP or to an AP. The OK response that is shown in Figure B4-1 on page B4-111 indicates that the DP is ready to accept the write data. The DP writes this data after the write phase has completed, and therefore the response to the DP write itself is given on the next operation. However, the SW-DP can buffer AP writes, as described in *SW-DP write buffering* on page B4-116.

There is no turnaround phase after the data phase. The host is driving the line, and can start the next operation immediately.

B4.2.2 Successful read operation (OK response)

On receiving a read packet request, if the SW-DP is ready for the RDATA data transfer phase, and there is no error condition, it issues an OK response. This response is indicated by a response value of `0b001`.

Therefore, a successful read operation consists of three phases:

1. An 8-bit read packet request, from the host to the target.
2. A 3-bit OK acknowledge response, from the target to the host.
3. A 33-bit RDATA data transfer phase, where data is transferred from the target to the host.

By default, there are single-cycle turnaround periods between the first and second of these phases, and after the third phase. See *Line turnaround* on page B4-107 for more information. However, there is no turnaround period between the second and third phases, because the line is driven by the target in both of these phases.

Figure B4-2 shows a successful read operation.

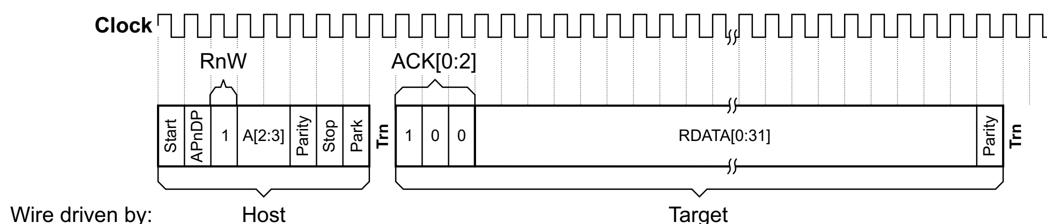


Figure B4-2 SWD successful read operation

If the host requested a read access to the DP, the SW-DP sends the read data immediately after the acknowledgement response.

Read accesses to the AP are posted, which means that the result of the access is returned on the next transfer. This result can be another AP register read, or a DP register read of RDBUFF.

To make a series of AP reads, a debugger only has to insert one read of the RDBUFF register:

- On the first AP read access, the read data that is returned is unknown. The debugger must discard this result.
- The next AP read access, if successful, returns the result of the previous AP read.
- This response can be repeated for any number of AP reads. Issuing the last AP read packet request returns the penultimate AP read result.
- The debugger can then read the DP RDBUFF register to obtain the last AP read result.

So that a debugger can recover from line errors, the next transaction after an AP register read can be any DP register read. If the next transaction is a DP register read other than a read of RDBUFF, then a following AP register read or read of RDBUFF returns the result of the first AP register read.

If the next transaction following an AP register read is an AP register write or a DP register write, the result of the first AP register read is lost because any following AP register read or read of RDBUFF returns an UNKNOWN value.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B4.2.3 WAIT response to read or write operation request

If the SW-DP is not able to process the request from the debugger immediately, it must issue a WAIT response. A WAIT response to a read or write packet request consists of two phases:

1. An 8-bit read or write packet request, from the host to the target.
2. A 3-bit WAIT acknowledge response, from the target to the host.

By default, there are single-cycle turnaround periods between these two phases, and after the second phase. See *Line turnaround* on page B4-107 for more information.

A WAIT response to a read or write packet request is shown in Figure B4-3.

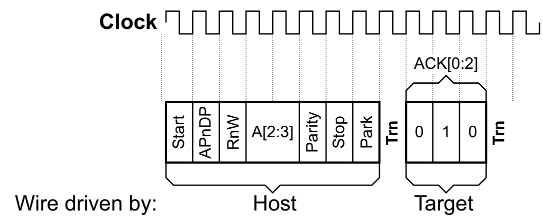


Figure B4-3 SWD WAIT response to a packet request

If overrun detection is enabled, CTRL/STAT.STICKYORUN is set to `0b1` and a data phase is required on a WAIT response. For more information, see *Sticky overrun behavior* on page B4-115.

A WAIT response must not be issued in response to the following requests, which must always be processed immediately:

- Reads of the DPIDR register.
- Reads of the CTRL/STAT register.
- Writes to the ABORT register.

In response to any other request, the DP issues a WAIT response if it cannot process the request, which happens if:

- A previous AP or DP access is outstanding.
- The new request is an AP read request and the result of the previous AP read is not yet available.

Normally, when a debugger receives a WAIT response it retries the same operation, to process data as quickly as possible. However, if several retries have been attempted, with a wait that is long enough for a slow interconnection and memory system to respond, the debugger might write to ABORT.DAPABORT, if appropriate. This value signals to the active AP that it must terminate the transfer that it is attempting. An AP implementation might be unable to terminate a transfer on its SoC interface, but on receiving a DAP abort request the AP must free up the interface to the Debug Port.

Writing to the ABORT register after receiving a WAIT response enables the debugger to access other parts of the debug system.

B4.2.4 FAULT response to read or write operation request

An SW-DP must not issue a FAULT response in response to:

- Reads of the DPIDR register, which is a read-only register.
- Reads of the CTRL/STAT register, which is a read/write register.
- Writes to the ABORT register, which is a write-only register.

For any other access, the SW-DP issues a FAULT response if any sticky flag is set to `0b1` in the CTRL/STAT register.

A FAULT response to a read or write packet request consists of two phases:

1. An 8-bit read or write packet request, from the host to the target.
2. A 3-bit FAULT acknowledge response, from the target to the host.

By default, there are single-cycle turnaround periods between these two phases and after the second phase. See *Line turnaround* on page B4-107 for more information.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

A FAULT response to a read or write packet request is shown in Figure B4-4.

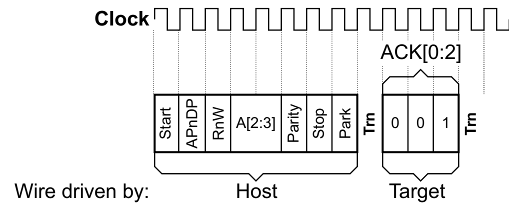


Figure B4-4 SWD FAULT response to a packet request

If overrun detection is enabled, CTRL/STAT.STICKYORUN is set to `0b1` and a data phase is required on a FAULT response. For more information, see *Sticky overrun behavior* on page B4-115.

Use of the FAULT response enables the protocol to remain synchronized. A debugger might stream a block of data and then check the CTRL/STAT register at the end of the block.

In an SW-DP, the sticky error flags are cleared to `0b0` by writing bits in the ABORT register.

B4.2.5 Protocol error response

If any of the following occurs, a protocol error occurs:

- The Parity bit does not match the parity of the packet request.

  For more information about the parity checks in the SWD protocol, see *Parity* on page B4-108.
- The Stop bit is not `0b0`.
- The Park bit is not `0b1`.
- DLCR.TURNROUND indicates an unsupported turnaround period.

Note

A mismatch between the Parity bit in the WDATA transfer phase and the parity of the data does not cause a protocol error response, because the SW-DP has already given its response to the header. For more information, see *Sticky flags and DP error responses* on page B1-41.

Target response to protocol errors

On detecting a protocol error, the target enters the protocol error state.

If overrun detection is enabled, CTRL/STAT.STICKYORUN is set to `0b1` and the target must wait until the data phase of the transaction has completed before entering the protocol error state. Otherwise, it enters the protocol error state immediately.

When a protocol error is detected by the SW-DP, the SW-DP does not reply to the packet request and does not drive the line. This situation is illustrated in Figure B4-5 on page B4-115.

Note

If SWD protocol version 2 is implemented, the SW-DP does also not reply to a TARGETSEL register write packet request.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

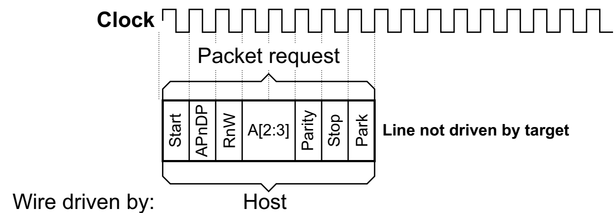


Figure B4-5 SWD protocol error after a packet request

In the protocol error state, the target behaves as follows:

- If the target detects a valid read of the DP DPIDR register, it is IMPLEMENTATION DEFINED whether the target leaves the protocol error state, and gives an OK response.
- If the target detects a valid packet header other than the read of the DP DPIDR register, or the target detects an IMPLEMENTATION DEFINED number of additional protocol errors, it enters the lockout state.

  Arm recommends that the target enters the lockout state after one more protocol error is detected while in the protocol error state.

If the target cannot leave the protocol error state on a read of the DPIDR register, the protocol error and lockout states are equivalent.

The target must leave the protocol error state on a line reset.

The target only leaves the lockout state on a line reset.

If the SW-DP implements SWD protocol version 2, it must enter the lockout state after a single protocol error immediately after a line reset. However, if the first packet request detected by the target following line reset is valid it can then revert to entering the lockout state after an IMPLEMENTATION DEFINED number of protocol errors.

Host response to protocol errors

If the host does not receive an expected response from the target, it must not drive the line for at least the length of any potential data phase and then attempt a line reset. For more information, see *Connection and line reset sequence on page B4-122*.

The host can attempt reads of the DP DPIDR register before attempting a line reset, as the target might respond and leave the protocol error state, but Arm does not recommend this solution.

If the transfer that resulted in the original protocol error response was a write, it can be assumed that no write occurred. If the original transfer was a read, it is possible that the read was issued to an AP. Although this scenario is unlikely, the possibility must be considered because reads are pipelined.

B4.2.6 Sticky overrun behavior

If an SW-DP receives a transaction request when the previous transaction has not completed, it returns a WAIT response. If overrun detection is enabled in the CTRL/STAT register, the CTRL/STAT.STICKYORUN flag is set to `0b1`. Subsequent transactions generate FAULT responses, because a sticky flag is `0b1`. If overrun detection is enabled, CTRL/STAT.STICKYORUN is also set if there is a FAULT response, protocol error, or line reset.

When overrun detection is enabled, WAIT and FAULT responses require a data phase:

- If the transaction is a read, the data in the data phase is UNKNOWN. The target does not drive the line, and the host must not check the parity bit.
- If the transaction is a write, the data phase is ignored.

Figure B4-6 on page B4-116 shows the WAIT or FAULT response to a read operation when overrun detection is enabled.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

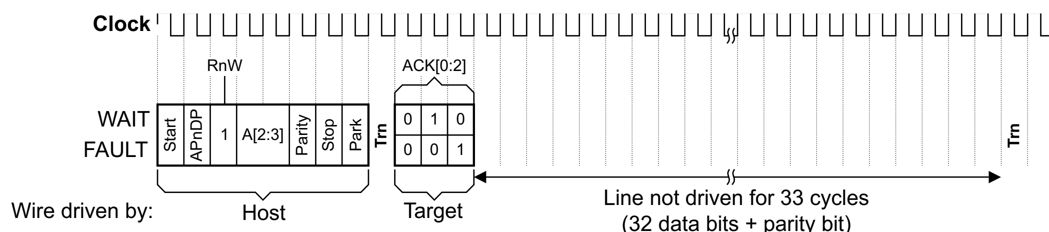


Figure B4-6 SW-DP WAIT or FAULT response to a read operation when overrun detection is enabled

Figure B4-7 shows the response to a write operation when overrun detection is enabled.

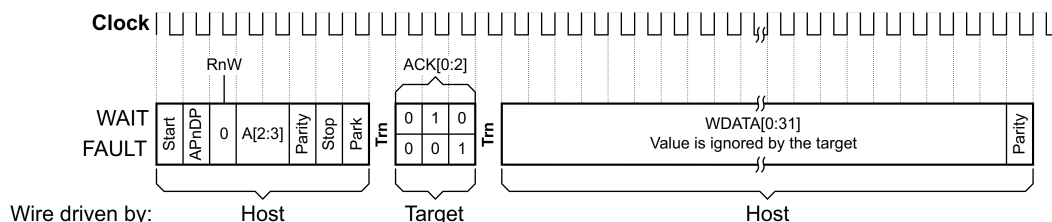


Figure B4-7 SW-DP WAIT or FAULT response to a write operation when overrun detection is enabled

B4.2.7 SW-DP write buffering

The SW-DP can implement a write buffer, enabling it to accept write operations even when other transactions are outstanding. If a DP implements a write buffer, the DP issues an OK response to a write request if it can accept the write into its write buffer. This response means that an OK response to a write request, other than a write to the ABORT register in the DP, indicates only that the write has been accepted by the DP. An OK response does not indicate that all previous transactions have completed.

The maximum number of outstanding transactions, and the types of transactions that might be outstanding, when a write is accepted, are IMPLEMENTATION DEFINED. However, the DP must be implemented so that all accesses occur in order. For example, if a DP only buffers writes to AP registers and it has any buffered writes, the DP must stall on a DP register write access to ensure that the writes are performed in order.

If a write is accepted into the write buffer but later abandoned, the CTRL/STAT.WDATAERR flag is set to `0b1`. A buffered write is abandoned if:

- A sticky flag is set to `0b1` by a previous transaction.
- A DP read of the DPIDR or CTRL/STAT register is made. Because the DP must not stall reads of these registers, the DP must:
  - Perform the DPIDR or CTRL/STAT register access immediately.
  - Discard any buffered writes, because otherwise they would be performed out-of-order.
  - Set the WDATAERR flag to `0b1`.
- A DP write of the ABORT register is made. The DP must not stall an ABORT register access.

The flag being set means that if software makes a series of AP write transactions, it might not be possible to determine which transaction failed from examining the ACK responses, but it might be possible to use other inquiries to find which write failed. For example, if when using the *auto-address increment* (AddrInc) feature of a MEM-AP, software can read the TAR to find the address of the last successful write transaction.

The write buffer must be emptied before the following operations can be performed:

- Any AP read operation.
- Any DP operation other than a read of the DPIDR or CTRL/STAT register, or a write of the ABORT register.

If the write buffer is not empty, attempting these operations causes a WAIT response from the DP.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

If pushed-verify or pushed-compare is enabled, AP write transactions are converted into AP reads. These transactions are then treated in the same way as other AP read operations. See *Pushed-compare and pushed-verify operations* on page B1-44 for details of these operations.

If a DP read of the DPIDR or CTRL/STAT register, or a DP write to the ABORT register, is required immediately after a sequence of AP writes, the software must first perform an access that the DP is able to stall. This ensures that the write buffer is emptied before the DP register access is performed. If this access is not done, WDATAERR might be set to `0b1`, causing the buffered writes to be lost.

Note

There is no requirement to insert an extra instruction to terminate the sequence of AP writes if the sequence of writes is followed by one of:

- An AP read operation.
- A write operation that can be stalled, such as a write to the SELECT register.

Often the requirement for an extra instruction can be avoided.

B4.2.8 Summary of target responses

The following subsections show the target SW-DP responses for different transaction requests:

- *Target SW-DP responses to DP transaction requests*.
- *Target SW-DP responses to AP transaction requests* on page B4-119.

Target SW-DP responses to DP transaction requests

For DP transaction requests, the register that is accessed is determined by:

- The value of A[3:2].
- In DPv1 or later, when A[3:2] is `0b01`, the value of SELECT.DPBANKSEL.

The behavior of some read transaction requests depends on the register that is accessed, as Table B4-2 shows.

Table B4-2 shows the target SW-DP response to all possible debugger DP read operation requests.

Table B4-3 on page B4-118 shows the target SW-DP response to all possible debugger DP write operation requests, assuming the WDATA parity check is good.

Table B4-2 Target response summary for DP read transaction requests

A[3:2]	SELECT. DPBANKSEL	Sticky flag value `0b1`?	AP Ready?	SW-DP (target) response	
				ACK	Action
`0b00`	x	x[a]	x[a]	OK	Respond with register value.
`0b01`	`0x0`	x[a]	x[a]	OK	Respond with register value.
	Not `0x0`	No	Yes	OK	Respond with register value.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential

Table B4-2 Target response summary for DP read transaction requests (continued)

A[3:2]	SELECT.DPBANKSEL	Sticky flag value 0b1?	AP Ready?	SW-DP (target) response	
				ACK	Action
0b10	x	No	Yes	OK	Respond by resending the last read value that is sent to the host. This value is the result of one of: • The most recent AP read • The most recent DP RDBUFF read.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].
0b11	x	No	Yes	OK	Respond with the value from the previous AP read, and set CTRL/STAT.READOK bit to 0b1.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b]. Set CTRL/STAT.READOK bit to 0b0.
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b]. Set CTRL/STAT.READOK bit to 0b0.

a. The SW-DP must always give an OK response to a read of the DPIDR or CTRL/STAT register.

b. See *Sticky overrun behavior* on page B4-115 for details about the data phase when overrun detection is enabled.

Table B4-3 Target response summary for DP write transaction requests

A[3:2]	Protocol version	Sticky flag value 0b1?	AP Ready?	SW-DP (target) response	
				ACK	Action
0b00	x	x	x	OK	Write WDATA value to ABORT register.
0b01 or 0b10	x	No	Yes[a]	OK	Write WDATA value to the selected DP register.
			No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table B4-3 Target response summary for DP write transaction requests (continued)

A[3:2]	Protocol version	Sticky flag value `0b1`?	AP Ready?	SW-DP (target) response	
				ACK	Action
`0b11`	v1	No	Yes[a]	OK	Register is reserved, SBZ. Write is ignored.
			No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].
	v2	x	x	None	Write WDATA to TARGETSEL register[c].

a. Writes might be accepted when other transactions are still outstanding. These writes might be abandoned later. See *SW-DP write buffering* on page B4-116 for more information.

b. See *Sticky overrun behavior* on page B4-115 for details about the data phase when overrun detection is enabled.

c. Target does not respond. See *Connection and line reset sequence* on page B4-122.

Fault conditions that are not shown in these tables are described in *Fault conditions not included in the target response tables* on page B4-120.

Target SW-DP responses to AP transaction requests

For AP transaction requests, the register that is accessed is determined by the value of A[3:2] combined with the values of SELECT.{APSEL,APBANKSEL}. For more information, see *Using the AP to access debug resources* on page A1-31.

Table B4-4 summarizes the target SW-DP response to all possible debugger AP read operation requests.

Table B4-5 on page B4-120 summarizes the target SW-DP response to all possible debugger AP write operation requests, assuming the WDATA parity check is good.

Table B4-4 Target response summary for AP read transaction requests

A[3:2]	Sticky flag value `0b1`?	AP Ready?	SW-DP (target) response	
			ACK	Action
`0bxx`	No	Yes	OK	Normally[a], return value from previous AP read[b] and set CTRL/STAT.READOK bit to `0b1`. Initiate AP read of addressed register.
		No	WAIT	No data phase, unless overrun detection is enabled[c]. Set CTRL/STAT.READOK bit to `0b0`.
	Yes	x	FAULT	No data phase, unless overrun detection is enabled[c]. Set CTRL/STAT.READOK bit to `0b0`.

a. If pushed-verify or pushed-compare is enabled, behavior is UNPREDICTABLE.

b. On the first of a sequence of AP reads, the value that is returned in the data phase is UNKNOWN.

c. See *Sticky overrun behavior* on page B4-115 for details of data phase when overrun detection is enabled.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table B4-5 Target response summary for AP write transaction requests

A[3:2]	Sticky flag value `0b1`?	AP Ready?	SW-DP (target) response	
			ACK	Action
`0bxx`	No	Yes[a]	OK	Normally[b], write WDATA value to the indicated AP register.
		No	WAIT	No data phase, unless overrun detection is enabled[c].
	Yes	x	FAULT	No data phase, unless overrun detection is enabled[c].

a. Writes might be accepted when other transactions are still outstanding. These writes might be abandoned later. See *SW-DP write buffering* on page B4-116 for more information.

b. If pushed-verify or pushed-compare is enabled, the write is converted to a read of the addressed AP register, and the value that is returned by this read is compared with the supplied WDATA value, see *Pushed-compare and pushed-verify operations* on page B1-44 for more information.

c. See *Sticky overrun behavior* on page B4-115 for details of data phase when overrun detection is enabled.

Fault conditions that are not shown in these tables are described in *Fault conditions not included in the target response tables*.

Fault conditions not included in the target response tables

There are two fault conditions that are not included in possible operation requests listed in Table B4-2 on page B4-117 to Table B4-5:

Protocol error

If there is a protocol error, the target does not respond to the request at all, which means that when the host expects an ACK response, it finds that the line is not driven. See *Protocol error response* on page B4-114.

WDATA fails parity check (write operations only)

The ACK response of the DP is sent before the parity check is performed, and is shown in Table B4-3 on page B4-118. When the parity check is performed and fails, the CTRL/STAT.WDATAERR flag is set to `0b1`.

B4.2.9 Summary of host responses

Every access by a debugger to an SW-DP starts with an operation request. *Summary of target responses* on page B4-117 listed all possible requests from a debugger, and summarized how the SW-DP responds to each request.

Whenever a debugger issues an operation request to an SW-DP, it expects to receive a 3-bit acknowledgment, as listed in the ACK columns of Table B4-2 on page B4-117 to Table B4-5. Table B4-6 on page B4-121 summarizes how the debugger must respond to this acknowledgment, for all possible cases.

Note

For SWD protocol version 2, Table B4-6 on page B4-121 does not apply to writes to TARGETSEL. See *Connection and line reset sequence*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table B4-6 Summary of host (debugger) responses to the SW-DP acknowledge

Operation requested	ACK received	Host response: Data phase	Host response: Additional action
R	OK	Capture RDATA from target and check for valid parity[a] and protocol.	If a parity or protocol fault occurs and it is not possible to flag the data as invalid, the host might have to repeat the original read request or use the RESEND register[b].
	Invalid ACK	Back off because of possible data phase.	The host can check CTRL/STAT register to see if the response sent was OK.
W	OK	Send WDATA.	Validity of this transfer is confirmed on next access.
	Invalid ACK	Back off to ensure that target does not capture next header as WDATA.	Repeat the write access. A FAULT response is possible if the first response was sent as OK but not recognized as valid by the debugger. The subsequent write is not affected by the first, misread, response.
x	WAIT	No data phase, unless overrun detection is enabled[c].	Normally, repeat the original operation request. See *WAIT response to read or write operation request* on page B4-113 for more information.
	FAULT	No data phase, unless overrun detection is enabled[c].	Can send new headers, but only an access to DP register addresses `0b0X` gives a valid response.
	No ACK	Back off because of possible data phase.	Can attempt IDCODE register read. Otherwise reset connection and retrain. See *Protocol error response* on page B4-114.

a. See *Parity* on page B4-108 for details of the parity checking.
b. The host debugger might support corrupted reads, or it might have to retry the transfer.
c. If overrun detection is enabled, a data phase is required. See *Sticky overrun behavior* on page B4-115 for a description of the behavior on read and write operations.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B4.3 SWD interface

The SWD protocol uses a synchronous serial interface, which comprises a single bidirectional data signal, and a clock signal.

This section gives an overview of the physical SWD interface.

B4.3.1 Line interface

The SWD interface uses a single bidirectional data pin, **SWDIO**. The same signal is used for both host and target sourced signals.

The SWD interface is synchronous, and requires a clock pin, **SWCLK**.

When the target samples **SWDIO**, sampling is performed on the rising edge of **SWCLK**. When the target drives **SWDIO**, or stops driving it, signal changes are performed on the rising edge of **SWCLK**.

The clock can be sourced from the target and exported, or provided by the host. This clock is then used by the host as a reference for generation and sampling of data so that the target is not required to perform any over-sampling.

Both the target and host can drive the bus HIGH and LOW or tristate it. The ports must be able to tolerate short periods of contention that might occur because of a loss of synchronization.

The clock can be asynchronous to any system clock, including the debug logic clock. The SWD interface clock can be stopped when the debug port is idle, see *About the SWD protocol* on page B4-106.

B4.3.2 Line pull-up

To make sure that the line is in a known state when neither host nor target is driving the line, a 100KΩ pull-up is required at the target. This pull-up can only be relied on to maintain the state of the wire. If the wire is driven LOW and released, the pull-up resistor eventually returns the line to the HIGH state, but this process takes many clock periods.

The pull-up is intended to prevent false detection of signals when no host is connected, and must be of a suitably high value to reduce current consumption from the target when the host actively pulls down the line.

Note

A small current drains from the target whenever the line is driven LOW. If the interface is left connected for extended periods when the target has to use a low-power mode, the line must be held HIGH, or reset, by the host until the interface is activated.

B4.3.3 Connection and line reset sequence

A debugger must use a line reset sequence to ensure that hot-plugging the serial connection does not result in unintentional transfers. The line reset sequence ensures that the SW-DP is synchronized correctly to the header that signals a connection.

The SWD interface does not include a reset signal. A line reset is achieved by holding the data signal HIGH for at least 50 clock cycles, followed by at least two idle cycles. Figure B4-8 on page B4-123 shows the interface timing for a line reset followed by a DP DPIDR register read.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

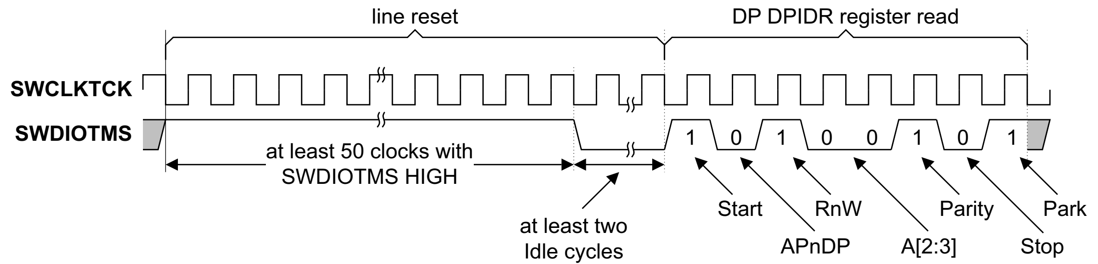


Figure B4-8 Line reset sequence followed by a DP DPIDR register read

A line reset is required when first connecting to the target. A line reset might be required following a protocol error. See *Protocol error response* on page B4-114.

A line reset resets DLCR.

Note

Other SW-DP registers are reset only by a powerup reset.

When waiting for a packet header, if the target detects a sequence of 50 clock cycles with the data signal held HIGH, followed by at least two idle cycles, it must enter the reset state. It is IMPLEMENTATION DEFINED whether a sequence of 50 clock cycles with the data signal held HIGH that is detected at any other time causes the interface to enter the reset state.

The only valid transactions in reset state are:

- A read of the DPIDR register. This transaction takes the connection out of reset state.
- One of the switching sequences defined by *Switching between SWD and JTAG* on page B5-128, if implemented.
- A write to the TARGETSEL register, if SWD protocol version 2 is implemented. If this transaction selects the target, the interface remains in reset state.

  Note

  Only writes to TARGETSEL immediately after entry to the reset state can select or deselect the target. See *Target selection protocol, SWD protocol version 2*.

Any of these sequences can be aborted by a second line reset. The behavior of the target is UNPREDICTABLE if any other transaction is made in reset state.

If the host does not see an expected response when reading the DPIDR register, it must retry the reset sequence, because the target might have been in a state where, for example, it treated the initial line reset as a data phase of a transaction and therefore did not detect it as a valid line reset. If so, the target detects the line reset as a protocol error and requires a second line reset to respond correctly.

If overrun detection is enabled, then the line reset sets CTRL/STAT.STICKYORUN to `0b1`.

B4.3.4 Target selection protocol, SWD protocol version 2

1. Perform a line reset. See Figure B4-9 on page B4-124.
2. Write to DP register `0xC`, TARGETSEL, where the data indicates the selected target. The target response must be ignored. See Figure B4-9 on page B4-124.
3. Read from the DP register `0x0`, DPIDR, to verify that the target has been successfully selected.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

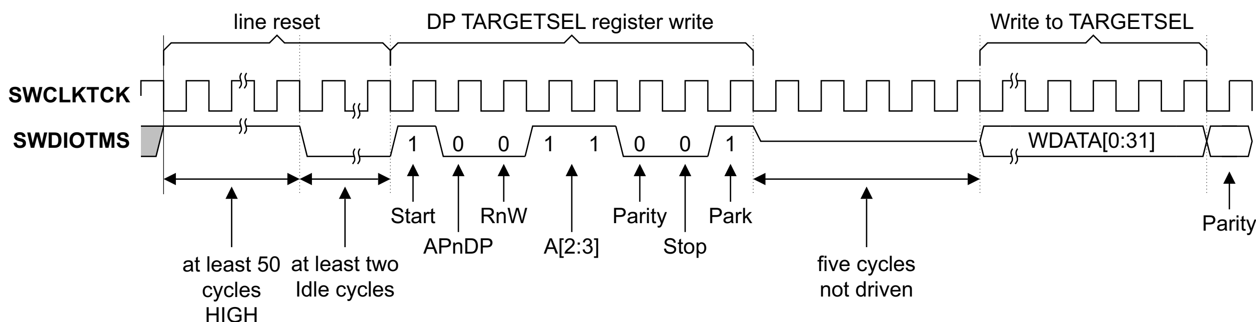


Figure B4-9 Line reset sequence followed by a DP TARGETSEL write

A write to the TARGETSEL register must always be followed by a read of the DPIDR register or a line reset. If the response to the DPIDR read is incorrect, or there is no response, the host must start the sequence again.

The target is selected on receiving a line reset sequence.

After receiving a line reset sequence, if the target receives a write request to TARGETSEL that does not select the same target, the target is deselected.

When deselected, the target ignores all accesses and must not drive the line. To select or deselect the target, a write to TARGETSEL must immediately follow a line reset sequence. Writes to TARGETSEL at any other time are UNPREDICTABLE.

If the target encounters a protocol error, it becomes deselected. Specifically, it does not respond to a read of the DPIDR register.

For more information, including the required behavior of the target during the response phase of the write to the TARGETSEL register, see *Sticky flags and DP error responses* on page B1-41.

A parity error in the data phase of a write to the TARGETSEL register does not set the CTRL/STAT.WDATAERR bit to `0b1`. A parity error in the data phase of a write to the TARGETSEL register is treated as a protocol error.

Accesses to the TARGETSEL register are not affected by the state of the CTRL/STAT.{WDATAERR, STICKYERR, STICKYCMP, STICKYORUN} bits.

Implementations of SWD protocol version 2 must also support dormant operation. See *Dormant operation* on page B5-131.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Chapter B5
The Serial Wire/JTAG Debug Port

This chapter describes multiple protocol interoperability as implemented in the Serial Wire/JTAG Debug Port (SWJ-DP) CoreSight component. It contains the following sections:

- *About the SWJ-DP* on page B5-126.
- *Switching between SWD and JTAG* on page B5-128.
- *Dormant operation* on page B5-131.
- *Restrictions on switching between operating modes* on page B5-138.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B5.1 About the SWJ-DP

The SWJ-DP interface provides a mechanism to select between SWD and JTAG Data Link protocols. This mechanism enables the DP to provide JTAG and SWD connections while making efficient use of package pins through sharing, or overlaying, pins.

Implementing an SWJ-DP enables an SoC to connect to legacy JTAG equipment without the need to change to the DP design. If an SWD tool is available, the JTAG interface is not needed and only two pins are required, instead of the four or five used for JTAG.This frees up some pins for alternative use. See also *Limitations when reusing pins*.

B5.1.1 SWJ-DP structure

The SWJ-DP comprises both an SW-DP and a JTAG-DP. It selects either the SW-DP or the JTAG-DP as the interface to the DAP, and switches between the SWD and JTAG Data Link protocols. Switching is achieved by routing the shared pins as shown in Table B5-1.

Table B5-1 Routing of SWJ-DP pins

SWJ-DP pin	SW-DP pin	JTAG-DP pin
SWDIOTMS	**SWDIO**	**TMS**
SWDCLKTCK	**CLK**	**TCK**
TDO	-	**TDO**
TDI	-	**TDI**
TRSTn	-	**TRSTn** (optional)

The mechanism for switching between SWD and JTAG is described in *Switching from JTAG to SWD operation* on page B5-129.

Note

While the DP is in SWD mode, the JTAG pins **TDI**, **TDO**, and **nTRST** are expected to be reused.

An SWJ-DP can be implemented in a package where the JTAG pins **TDI**, **TDO**, and **nTRST** are not connected because an SWJ-DP does not need these JTAG pins to switch the DP to SWD mode.

The following rules apply to SWJ-DP implementations:

- There is no requirement to implement separate SW-DP and JTAG-DP blocks within the SWJ-DP.
- The number, type, and location of APs accessed by the SWJ-DP must not depend on whether the SW-DP or the JTAG-DP is selected. Each DP type must access the same debug resources. There is no requirement to implement these APs as shared APs.

  For this reason, tools must not rely on the state of a DP or any AP it accesses to persist when the other DP is selected. After switching DPs, the debugger must re-initialize the DAP, including setting the CTRL/STAT.{CDBGPWRUPREQ, CSYSPWRUPREQ} bits correctly.
- The JTAG-DP and SW-DP programmers' models do not have to implement the same DP architecture version. See Chapter B1 *About the DP*.
- If the JTAG protocol is never used, a pull-down on TDI at the target is required.

B5.1.2 Limitations when reusing pins

If the JTAG pins on the SWJ-DP interface are not used, they can be reused. However, there is a trade-off between the number of pins that are used and compatibility with existing hardware and test equipment.

In the following situations, the use of a JTAG debug interface must be maintained:

- The DP is included in an existing scan chain. This is often true for on-chip TAPs used for testing or other purposes.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- The device must be enabled to be cascaded with legacy devices which use JTAG for debug, although this requirement can also be supported using a JTAG-AP.
- There is a requirement to use existing debug hardware with the corresponding test TAPs, for example, in *Automatic Test Equipment* (ATE).

The following must be observed:

- When reusing pins, there must be no conflict with their use in JTAG operation.
- To support use of SWJ-DP in a scan chain with other JTAG devices, the default behavior after a DP reset must be to transition any reused pins from their alternative function to their JTAG function, if the direction of the alternative function is compatible with being driven by a JTAG debug device. The transition of the JTAG TAP to the Shift-DR or Shift-IR state can be used for this transition.
- The alternate function of reused pins cannot be used while the SoC is being used in JTAG operation.
- The switching scheme must enable a JTAG debugger to connect by sending a specific sequence, provided there is no conflict on the **TDI** and **TDO** pins.
- The connection sequence that is used for SWD must be safe when applied to the JTAG interface, even when hot-plugged, to enable the debugger to continually retry its access sequence.
- A sequence with **TMS** HIGH must ensure that all parts of the SWJ-DP are in a known reset state.
- The pattern that selects SWD must have no effect on JTAG devices.
- An SWJ-DP implementation must be compatible with a free-running **TCK**, or a gated clock, that is supplied by the external tools.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B5.2 Switching between SWD and JTAG

SWJ-DP enables either an SWD or JTAG protocol to be used on the debug port. This section describes in detail how the switching mechanism is implemented, and how to switch between the two interfaces.

Note

On devices where the dormant state of operation is implemented, Arm deprecates the mechanism that is described in *The Switching Mechanism*, and recommends using a transition through dormant state instead. For more information about dormant state, see *Dormant operation* on page B5-131.

B5.2.1 The Switching Mechanism

The implementation uses a watcher circuit that detects a specific 16-bit select sequence on **SWDIOTMS**:

- A 16-bit sequence to switch from JTAG to SWD operation.
- A 16-bit sequence to switch from SWD to JTAG.

Note

Arm deprecates use of these sequences on devices where the dormant state of operation is implemented.

Arm recommends using a transition through dormant state instead. For more information, see *Dormant operation* on page B5-131.

SWJ-DP defaults to JTAG operation on powerup reset and therefore the JTAG protocol can be used from reset without sending a select sequence.

Switching from one protocol to the other can only occur when the selected interface is in its reset state. The JTAG TAP state machine must be in its Test-Logic-Reset (TLR) state and the SWD interface must be in line-reset. The powerup reset state for a JTAG TAP state machine is the Test-Logic-Reset state.

Having detected a switching sequence, SWJ-DP does not detect more sequences until after a reset condition. If JTAG is selected, the JTAG TAP state machine being in the TLR state is the reset condition. If SWD is selected, a line reset is the reset condition.

Figure B5-1 on page B5-129 is a simplified state diagram that shows how SWJ-DP transitions between selected, detecting, and selection states.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

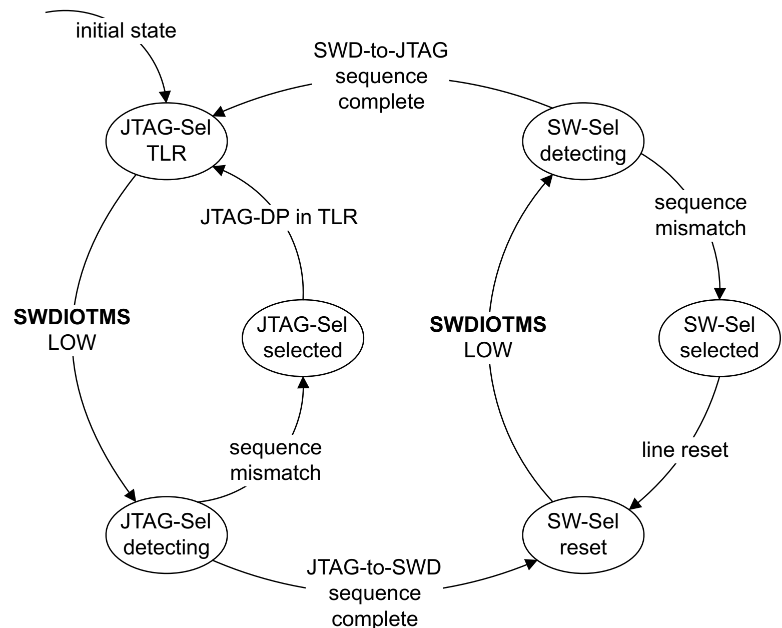


Figure B5-1 SWD and JTAG select state diagram

Note

For Figure B5-1:

- The JTAG-to-SWD sequence terminates in the SW-Sel reset state.
- The SWD-to-JTAG sequence terminates in the JTAG-Sel TLR state.

The recommended sequences end with a reset sequence for the selected state, to ensure that the target is in the relevant reset state.

B5.2.2 Switching from JTAG to SWD operation

To switch SWJ-DP from JTAG to SWD operation:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that the current interface is in its reset state. The JTAG interface only detects the 16-bit JTAG-to-SWD sequence starting from the TLR state.
2. Send the 16-bit JTAG-to-SWD select sequence on **SWDIOTMS**.
3. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that if SWJ-DP was already in SWD operation before sending the select sequence, the SWD interface enters line reset state.

The 16-bit JTAG-to-SWD select sequence is `0b0111 1001 1110 0111`, *most-significant-bit* (MSB) first. This sequence can be represented as one of the following:

- `0x79E7`, transmitted MSB first.
- `0xE79E`, transmitted *least-significant-bit* (LSB) first.

Figure B5-2 on page B5-130 shows the interface timing. The sequence that is shown in the figure has been chosen to ensure that the SWJ-DP switches to SWD, independent of whether it was previously expecting JTAG or SWD. As long as the 50 cycles with **SWDIOTMS** HIGH sequence are sent first, the JTAG-to-SWD select sequence does not affect SW-DP or the SWD and JTAG protocols that are used in the SWJ-DP, or any other TAP Controllers that might be connected to **SWDIOTMS**.

On selecting SWD operation, the SWD interface returns to its reset state. See *Connection and line reset sequence on page B4-122*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Figure B5-2 shows that JTAG-to-SWD sequence begins immediately after the 50 cycles with **SWDIOTMS** HIGH. Unlike a normal line reset, the two cycles with **SWDIOTMS** LOW are not present.

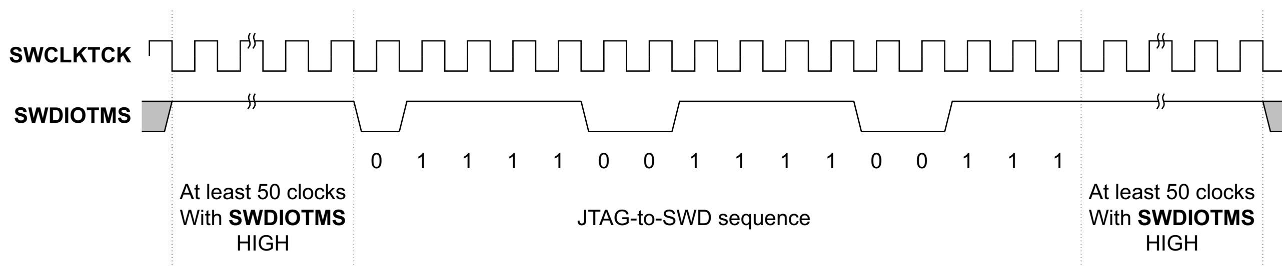


Figure B5-2 JTAG-to-SWD sequence timing

B5.2.3 Switching from SWD to JTAG operation

To switch SWJ-DP from SWD to JTAG operation:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that the current interface is in its reset state. The SWD interface only detects the 16-bit SWD-to-JTAG sequence when it is in the reset state.
2. Send the 16-bit SWD-to-JTAG select sequence on **SWDIOTMS**.
3. Send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that if SWJ-DP was already in JTAG operation before sending the select sequence, the JTAG TAP enters the Test-Logic-Reset state.

The 16-bit SWD-to-JTAG select sequence is `0b0011 1100 1110 0111`, MSB first. This sequence can be represented as either of the following:

- `0x3CE7`, transmitted MSB first
- `0xE73C`, transmitted LSB first.

Figure B5-3 shows the SWD-to-JTAG sequence timing. The sequence that is shown in the figure has been chosen to ensure that the SWJ-DP switches to JTAG, independent of whether it was previously expecting JTAG or SWD. If the **SWDIOTMS** HIGH sequence is sent first, the SWD-to-JTAG select sequence does not affect SW-DP or the SWD and JTAG protocols that are used in the SWJ-DP, or any other TAP Controllers that might be connected to **SWDIOTMS**.

Figure B5-3 shows that the SWD-to-JTAG sequence begins immediately after the 50 cycles with **SWDIOTMS** HIGH.Unlike a normal line reset, the two cycles with **SWDIOTMS** LOW cycles are not present.

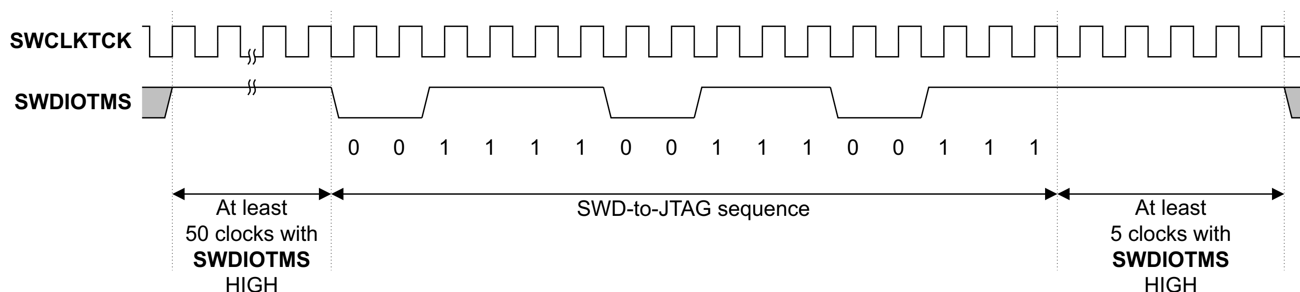


Figure B5-3 SWD-to-JTAG sequence timing

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B5.3 Dormant operation

An alternative to the selection mechanism for switching between JTAG and SWD operation that is described in *Switching between SWD and JTAG* on page B5-128 is the dormant state of operation.

To switch between JTAG and SWD operation, a debugger must first place the target into dormant state and then transition to the required operating state.

Using dormant state allows the target to be placed into a quiescent mode, allowing devices to inter-operate with other devices implementing other protocols. Those other protocols must also implement a quiescent state, with a mechanism for entering and leaving that state that is compatible, but not necessarily compliant, with the SWJ-DP and SW-DP protocols.

Dormant operation is required by SWJ-DP and SW-DP implementations that implement SWD protocol version 2. SWD protocol version 2 is described in Chapter B4 *The Serial Wire Debug Port*. Otherwise, support for dormant state is IMPLEMENTATION DEFINED. In the dormant state, the target must ignore any stimulus, with any timing, other than a defined Selection Alert sequence.

The Selection Alert sequence must be followed by a protocol-specific Activation code.

Selection of dormant state is possible when either JTAG or SWD operation is selected. Figure B5-4 extends the state diagram of Figure B5-1 on page B5-129 to include selection of dormant state, for an SWJ-DP implementation.

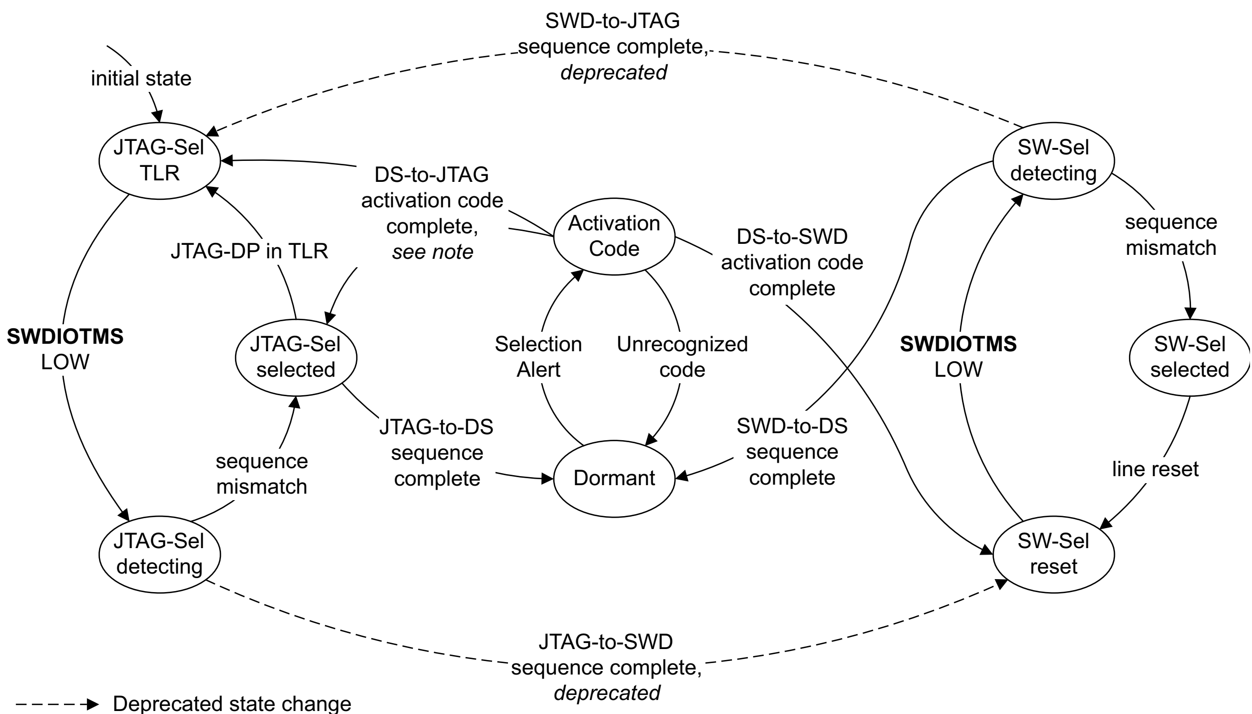


Figure B5-4 SWJ-DP selection of JTAG, SWD, and dormant states

Note

Following the DS-to-JTAG activation code, the JTAG TAP is in either the Test-Logic-Reset state or Run-Test/Idle state, and therefore this state machine is in either the JTAG-Sel TLR state or the JTAG-Sel selected state. Normally, the TAP state that the state machine returns to is the TAP state it left from, but it is also possible to reset the JTAG TAP state machine when JTAG is not the selected protocol. To ensure that the TAP is in the Run-Test/Idle state, Arm recommends that the DS-to-JTAG sequence is followed by a single clock with **SWDIOTMS** LOW.

The DS-to-SWD sequence is shown terminating in the SW-Sel reset state. The recommended sequence ends with a line reset to ensure that the target is in the reset state.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B5.3.1 Using the dormant state outside of SWJ-DP

An SWD device that does not implement JTAG can nevertheless implement dormant state, and inter-operate with SWJ-DP and other JTAG devices that also implement dormant state. In this case:

- The operating mode selection state machine is simplified.
- The initial state, entered on a powerup reset, is the dormant state.

Figure B5-5 shows the state diagram for an SW-DP that implements protocol version 2 illustrating it supports dormant operation.

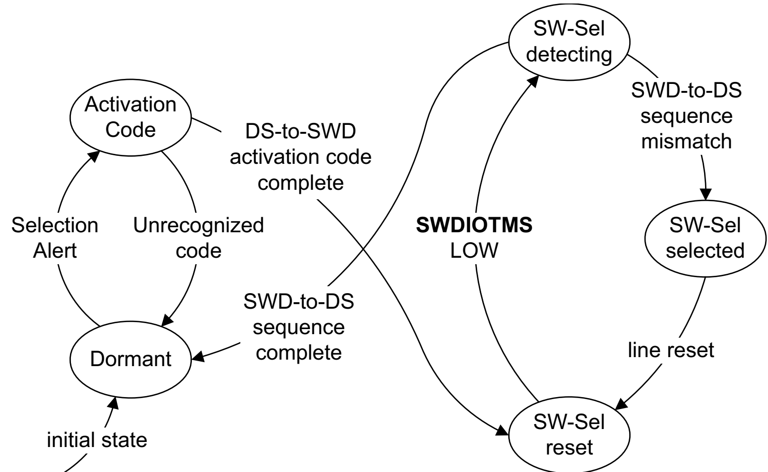


Figure B5-5 SW-DP selection of SWD, and dormant states

The dormant state enables multi-drop SWJ-DP, SW-DP, and JTAG TAPs to share a physical connection to a host, as shown in Figure B5-6. These different devices can be in different physical packages, on different dies in a single package, or on a single die.

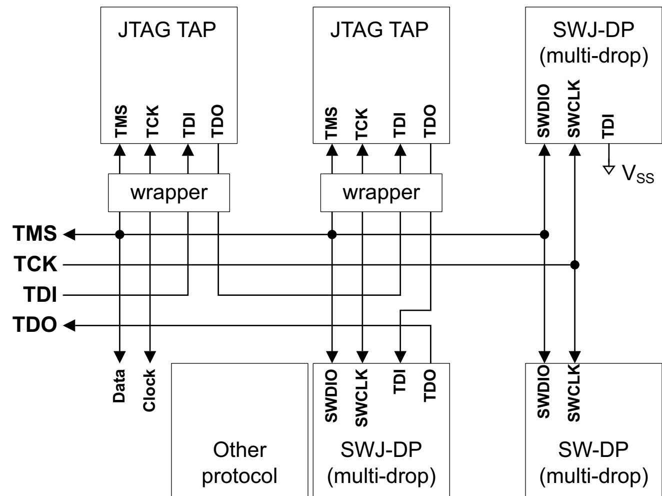


Figure B5-6 Multiple JTAG, SW, SWJ (multi-drop), and other protocol devices on shared connection

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B5.3.2 Switching from JTAG to dormant state

To switch from JTAG to dormant state, a debugger must:

1. Send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence places the JTAG TAP state machine into the TLR state, and selects the IDCODE instruction.
2. Send the recommended 31-bit JTAG-to-DS select sequence on **SWDIOTMS**.

The recommended 31-bit JTAG-to-DS select sequence is `0b010_1110_1110_1110_1110_1110_1110_0110`, MSB first. This sequence can be represented as either:

- `0x2EEEEEE6` transmitted MSB first, that is, starting from bit 30.
- `0x33BBBBBA` transmitted LSB first.

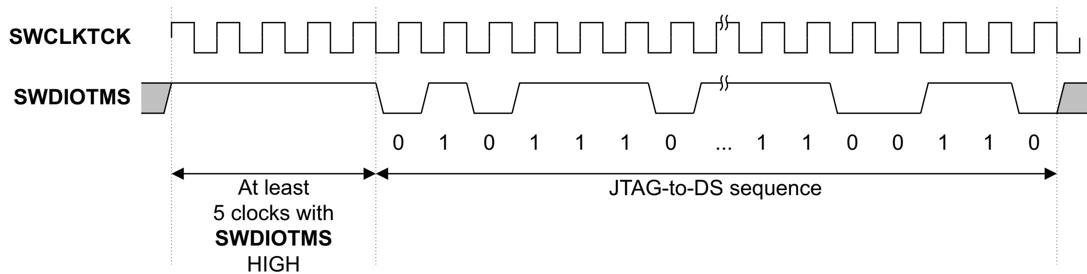


Figure B5-7 Recommended JTAG-to-DS sequence timing

Requirements for implementations

The JTAG-to-DS sequence is the shortest sequence that switches from JTAG-to-DS. For compatibility with other standards, all JTAG devices that implement dormant state must recognize other sequences as valid JTAG-to-DS select sequences.

The full sequence is defined around the concept of a *zero-bit-DR-scan* (ZBS or ZBS scan) which is in turn defined by transitions of the JTAG TAP state machine. A ZBS is defined as any JTAG TAP state machine sequence that starts at Capture-DR and ends in Update-DR without passing through Shift-DR.

Examples of a ZBS are:

- **Capture-DR → Exit1-DR → Update-DR**
- **Capture-DR → Exit1-DR → Pause-DR → … → Pause-DR → Exit2-DR → Update-DR**

The sequence also uses the ZBS count, which is defined as follows:

- If the TAP state machine enters either the Select-IR-Scan or TLR state, the ZBS count is unlocked and reset to zero, which includes asynchronously entering Test-Logic-Reset following assertion of nTRST. At reset, the ZBS count is unlocked and reset to zero.
- On entering Update-DR at the end of a ZBS scan, if the ZBS count is unlocked and less than seven, it is incremented by one.
- The counter does not increment past seven. On entering Update-DR at the end of a ZBS scan, if the ZBS count is unlocked and equal to seven, it is not incremented. The count does not wrap to zero.
- The ZBS count is locked if the TAP state machine enters the Shift-DR state and the ZBS count is not zero.

The JTAG-to-DS sequence is defined as any sequence of TAP state machine transitions that terminates in the Run-Test/Idle state with a locked ZBS count of six. On entering Run-Test/Idle, the target is placed into *Dormant State* (DS).

The behavior of the target on entering Run-Test/Idle with other locked ZBS counts is IMPLEMENTATION DEFINED.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Although the recommended JTAG-to-DS sequence starts by placing the JTAG TAP state machine in the Test-Logic-Reset state, this transition is not required for recognizing the JTAG-to-DS sequence. However, tools must ensure that the IR is loaded with either the BYPASS or IDCODE instruction before placing the target into the dormant state. If the IR is not loaded with either of these instructions when the target is put into dormant state, the behavior is unpredictable.

The pseudocode function `EnterDormantState` describes the function of the JTAG-to-DS sequence detector. It is notionally called on every TAP state machine transition. The function takes the state being entered as an argument, and returns a Boolean that indicates whether dormant state must be entered.

For details of the pseudocode language, see Appendix E3 *Pseudocode Definition*.

```
enumeration TAPState {
    TestLogicReset, RunTestIdle,
    SelectDRScan, CaptureDR, ShiftDR, Exit1DR, PauseDR, Exit2DR, UpdateDR,
    SelectIRScan, CaptureIR, ShiftIR, Exit1IR, PauseIR, Exit2IR, UpdateIR};

boolean shiftDRflag = FALSE;
integer ZBScount = 0;
boolean ZBSlocked = FALSE;

// EnterDormantState()
// ===================

boolean EnterDormantState(TAPState state)
    case state of
        when CaptureDR
            shiftDRflag = FALSE;
        when ShiftDR
            shiftDRflag = TRUE;

            if ZBScount != 0 then ZBSlocked = TRUE;
        when UpdateDR
            if !ZBSlocked && !shiftDRflag && ZBScount < 7 then
                ZBScount = ZBScount + 1;
        when SelectIRScan, TestLogicReset
            ZBScount = 0;  ZBSlocked = FALSE;
        return state == RunTestIdle && ZBSlocked && ZBScount == 6;
```

Note

If the JTAG-to-DS sequence is terminated by entering the TLR state, an SWJ-DP can immediately detect a JTAG-to-SWD sequence.

B5.3.3 Switching from SWD to dormant state

To switch from SWD to dormant state:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that the SWD interface is in the reset state. The target only detects the SWD-to-DS sequence when it is in the reset state.
2. Send the 16-bit SWD-to-DS select sequence on **SWDIOTMS**.

The 16-bit SWD-to-DS select sequence is `0b0011_1101_1100_0111`, MSB first. This sequence can be represented as either:

- `0x3DC7` transmitted MSB first.
- `0xE3BC` transmitted LSB first.

Figure B5-8 on page B5-135 shows that the SWD-to-DS sequence begins immediately after the 50 cycles with **SWDIOTMS** HIGH. Unlike a normal line reset, the two cycles with **SWDIOTMS** LOW are not present.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

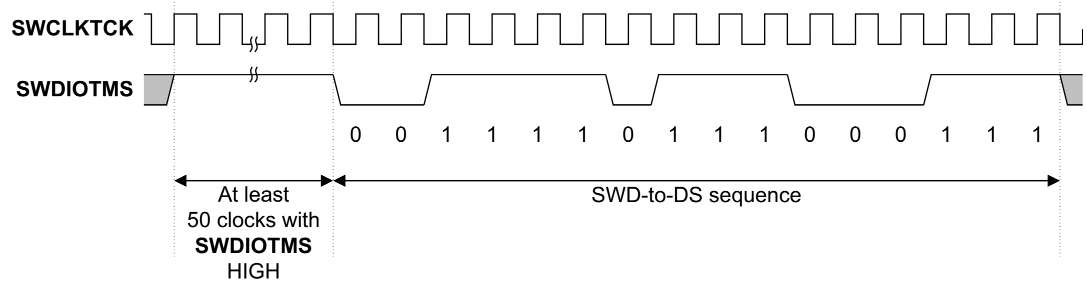


Figure B5-8 SWD-to-DS sequence timing

B5.3.4 Leaving dormant state

To ensure that the probability for any protocol being used to accidentally signal the DP to leave the dormant state is low, the sequence for leaving the dormant state is considerably longer than the sequence for entering it.

To signal the DP to leave the dormant state:

1. Send at least eight **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that the target is not in the middle of detecting a Selection Alert sequence. The target is permitted to detect the Selection Alert sequence even if this eight cycle sequence is not present.
2. Send the 128-bit Selection Alert sequence on **SWDIOTMS**.
3. Send four **SWCLKTCK** cycles with **SWDIOTMS** LOW. The target must ignore the value on **SWDIOTMS** during these cycles.
4. Send the required activation code sequence on **SWDIOTMS**.
5. Send a sequence to place the target into a known state
   - If selecting JTAG, the target is in either the Run/Test Idle or TLR states, see the *Note* that follows Figure B5-4 on page B5-131 for more information. Arm recommends that the debugger sends one **SWCLKTCK** cycle with **SWDIOTMS** LOW, to ensure that the TAP state machine is in the Run-Test/Idle state. Alternatively, send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH to ensure that the TAP state machine is in the Test-Logic/Reset state.
   - If selecting SWD, the target is in the protocol error state. The debugger must send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This sequence ensures that the SWD interface is in the line reset state.

Note

The Activation code selects a protocol, not a target. In a multi-drop SWD system with multiple SW-DPs, a target must then be selected. For more information, see *Target selection protocol, SWD protocol version 2* on page B4-123.

The Selection Alert sequence, in binary, is

```
0100_1001_1100_1111_1001_0000_0100_0110_1010_1001_1011_0100_1010_0001_0110_0001_
1001_0111_1111_0101_1011_1011_1100_0111_0100_0101_0111_0000_0011_1101_1001_1000
```

This sequence is sent MSB first. This sequence can be represented as either:

- `0x49CF9046 A9B4A161 97F5BBC7 45703D98` transmitted MSB first.
- `0x19BC0EA2 E3DDAFE9 86852D95 6209F392` transmitted LSB first.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential

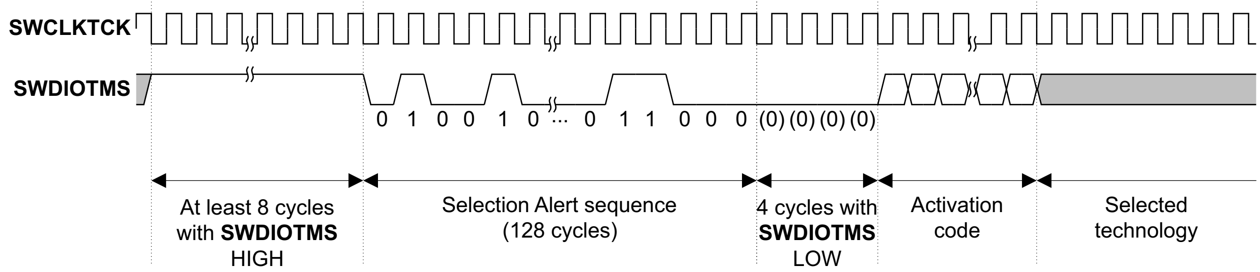


Figure B5-9 Selection Alert sequence

Note

The Selection Alert sequence can be generated by implementing a Linear Feedback Shift Register (LFSR) implementing feedback on bits 6, 5, 3 and 0, starting in the state `0b1001001` and shifting out one bit from bit 0 each cycle. The sequence starts with a zero start bit and continues with the output of the LFSR.

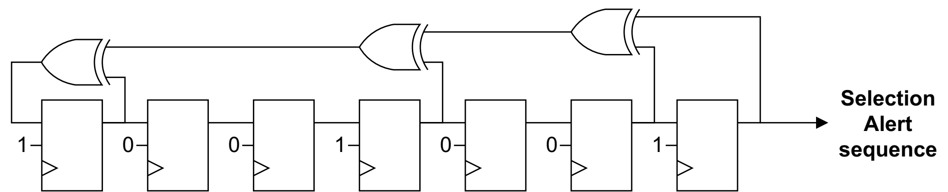


Figure B5-10 LSFR for generating Selection Alert sequence

The value of the activation code depends on whether SWD or JTAG operation is to be requested. Table B5-2 defines the activation codes a debugger must use for JTAG devices, SW-DP devices, and SWJ-DP devices. These sequences are sent MSB first.

Table B5-2 Activation codes

Activation code	Value, MSB first	Devices activated				Protocol selected
		Other JTAG	ADIv5 Debug Ports			
			JTAG	SW	SWJ	
JTAG-Serial	`0b0000_0000_0000`	Yes	Yes	No	Yes	JTAG
Arm CoreSight SW-DP	`0b0101_1000`	No	No	Yes	Yes	SWD
Arm CoreSight JTAG-DP	`0b0101_0000`	No	Yes	No	Yes	JTAG

JTAG online activation codes

For compatibility with other standards, all JTAG devices that implement dormant state using the ADIv5-defined selection alert sequence, must recognize other sequences as valid JTAG-Serial activation codes.

Figure B5-11 on page B5-137 shows the sequence that a JTAG device must recognize, as a state diagram.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

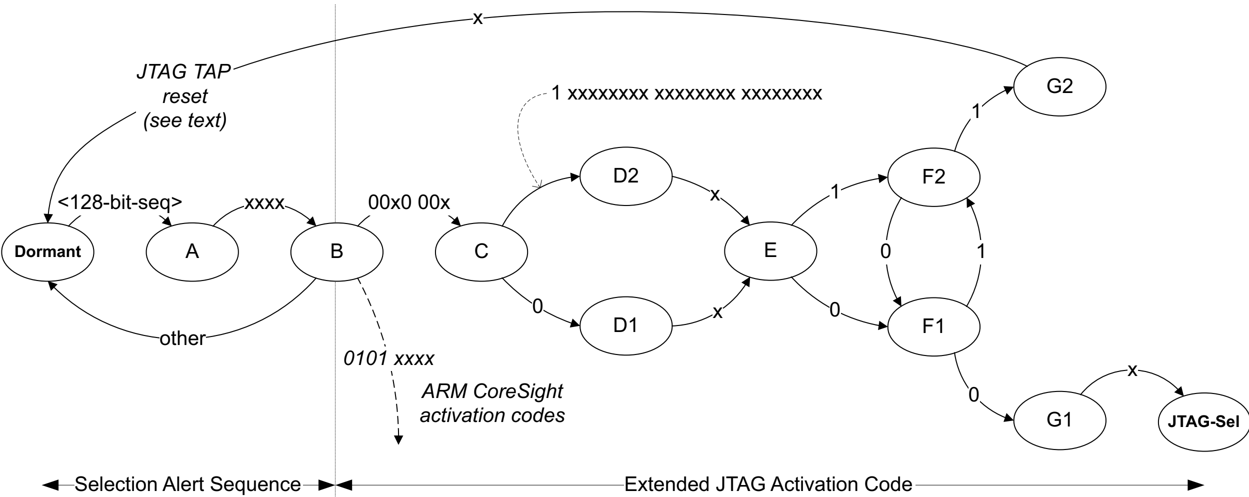


Figure B5-11 Dormant to JTAG state diagram

Each of the bit-strings that are shown in Figure B5-11 are received MSB first. The transition out of state G2 requires a reset of the JTAG TAP, but otherwise returns to dormant state. For more information on this sequence, contact Arm.

Note

ADIv5 does not define any other activation codes, but also does not prohibit an implementation from recognizing other activation codes for compatibility with other standards. Implementations can also use alternative selection alert mechanisms. Debuggers can generate multiple selection alert sequences to alert multiple devices, and then use the common activation codes to select which devices to activate.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

B5.4 Restrictions on switching between operating modes

A debugger must not mix JTAG-DP and SW-DP reads and writes of DAP registers in a single debug session. A single debug session is defined as from when a debugger connection is made with the system in a reset state through to the debugger connection being broken. At the start of a debug session, the state of the target is UNKNOWN.

Attempting to mix JTAG-DP and SW-DP reads and writes of DAP registers while any component of the DAP is active can have unpredictable results.

A powerup reset cycle might be required to reset the DAP before a change in active Data Link protocol. However, this cycle is not required when switching between the active protocol and dormant state.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part C

The Access Port

Chapter C1
About the AP

A DAP can include multiple APs.

This chapter gives an overview of APs, and describes the features that must be implemented by every AP. It contains the following sections:

- *AP requirements* on page C1-142.
- *Selecting and accessing an AP* on page C1-143.
- *AP Programmers' Model* on page C1-144.

The following chapters provide two AP definitions:

- Chapter C2 *The Memory Access Port*.
- Chapter C3 *The JTAG Access Port*.

Designers can use the ADI architecture specification to implement other APs.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.1 AP requirements

A DAP can implement multiple APs, and use a mixture of AP types.

Note

This specification permits a DAP to include AP types that are not defined in the specification, even if such an AP is the only AP in the DAP. A debugger must be able to detect any AP and must ignore any AP that it does not recognize.

All APs must observe the following requirements:

- Every AP must implement an IR as described in *AP Programmers' Model* on page C1-144. This identification model is required for implementations of the MEM-AP and JTAG-AP implementations that are defined by Arm, by any future Arm AP implementations, and by any APs that might be implemented by any third party.
- Any AP must support accesses by the implemented DP, as described in *Using the AP to access debug resources* on page A1-31. A summary of how to access an AP is given in *Selecting and accessing an AP* on page C1-143.
- For all APs, reserved registers must be RES0. This requirement applies to all APs, including any implemented by companies other than Arm.

There are no other requirements for APs in the ADIv5 specifications. All features that are provided by an AP can be IMPLEMENTATION DEFINED.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2 Selecting and accessing an AP

Any APACC request to a MEM-AP, a JTAG-AP, or an AP not defined by this specification must be answered by the DP according to the following addressing scheme:

- The value of the SELECT.APSEL field must be used to select which AP is accessed.
- The value of the SELECT.APBANKSEL field must be used to select the four-register bank of AP registers is accessed. This information is passed as the A[7:4] field of the AP access.
- The A[3:2] field that is passed in the APACC access must be used to select the AP register within the selected four-register bank.
- The RnW field for the APACC access must be used to determine whether the AP register access is a read access or a write access.

For detailed information about DP support for APACC accesses, see Chapter B3 *The JTAG Debug Port* and Chapter B4 *The Serial Wire Debug Port*.

Examples of implementations of APACC accesses are shown in Figure C2-1 on page C2-149 for a MEM-AP, and Figure C3-1 on page C3-191 for a JTAG-AP.

C1.2.1 Stalling accesses

AP interfaces can support stalling accesses, which enable the AP to be connected to slow devices, such as a memory system or a long JTAG scan chain. As a result, the DAP can put an AP access into a pending state, and the access does not have to complete within a fixed number of cycles. This is important because often an AP access cannot complete until the associated memory access or JTAG scan has completed. For more information, see:

- *Stalling accesses* on page C2-154, for stalling accesses to a MEM-AP.
- *Stalling accesses* on page C3-196, for stalling accesses to a JTAG-AP.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C1.3 AP Programmers' Model

This section describes the AP programmers' model, which must be implemented by all APs.

C1.3.1 Summary

ADIv5 requires every AP to implement an AP Identification Register, IDR, at offset `0xFC`. IDR is the last register in the AP register space, and is described in *IDR, Identification Register*.

The IDR is the only register that must be implemented by all Access Ports, see Table C1-1.

For information about the programmers' model for specific AP implementations, see Chapter C2 *The Memory Access Port* and Chapter C3 *The JTAG Access Port*.

Table C1-1 Common AP programmers' model

Offset	Type	Name	Description
`0xFC`	RO	IDR	Identification Register.

C1.3.2 IDR, Identification Register

Purpose

IDR identifies the Access Port. An IDR value of zero indicates that there is no AP present.

Usage constraints

The value of IDR after a reset is IMPLEMENTATION DEFINED.

IDR is accessible as follows:

Default
RO

Configurations

IDR is implemented in DPv0, DPv1, and DPv2.

DAPs that comply with the ADIv5 specification must implement the JEP106 code and provide a value in the REVISION and CLASS fields.

Attributes

~~IDR is~~ A 32-bit read-only register.

Field descriptions

The IDR bit assignments are:

31 28	27 17	16 13	12 8	7 4	3 0
REVISION	DESIGNER	CLASS	RES0	VARIANT	TYPE

REVISION, bits[31:28]

Starts at `0x0` for the first implementation of an AP design~~,~~ and increments by 1 on each major or minor revision of the design. Major design revisions introduce functionality changes, minor revisions are bug fixes.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

DESIGNER, bits[27:17]

Code that identifies the designer of the AP.

This field indicates the designer of the AP 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 encoding that is used in the IDR is as follows:

- The JEP106 continuation code, IDR bits[27:24], is the number of times that `0x7F` appears before the final number. For example, for Arm Limited this field is `0x4`.
- The JEP106 identification code, IDR bits[23:17], equals bits[6:0] of the final number of the JEDEC code. For example, for Arm Limited this field is `0x3B`.

Note

The JEP106 codes are assigned by JEDEC to identify the manufacturer of a device. However, in the AP Identification register they identify the designer of the AP.

An implementer of an Arm MEM-AP or JTAG-AP must not change these AP Identification Register values.

Note

- For backwards compatibility, debuggers must treat an AP return a JEP106 field of zero as an AP designed by Arm. This encoding was used in early implementations of the DAP. In such an implementation, the REVISION and CLASS fields are also RAZ.
- DAPs that comply with the ADIv5 specification must use the JEP106 code and provide a value in the REVISION and CLASS fields.

CLASS, bits[16:13]

Defines the class of AP. An AP belongs to a class if it follows a programmers' model that is defined as part of the ADIv5 specification or extensions to it. This field can have the following values:

`0b0000` No defined class.

`0b0001` COM Access Port. See Chapter C4 *COM-AP programmers' model*.

`0b1000` Memory Access Port. See Chapter C2 *The Memory Access Port*.

Bits[12:8] Reserved, RES0. This field is reserved for future ID register fields. If a debugger reads a non-zero value in this field, it must treat the AP as unidentifiable.

VARIANT, bits[7:4]

Together with the TYPE field, this field identifies the AP implementation. VARIANT differentiates AP implementations that have the same value of TYPE.

Each AP designer must maintain their own list of implementations and associated AP Identification codes.

TYPE, bits[3:0]

Indicates the type of bus, or other connection, that connects to the AP. Table C1-2 on page C1-146 lists the possible values of the Type field for an AP designed by Arm. It also shows the value of the CLASS field, which corresponds to bits[16:13] of the IDR, for each value of TYPE.

Together with the VARIANT field, this field identifies the AP implementation. AP implementations that have the same value of TYPE are differentiated by their VARIANT value.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Each AP designer must maintain their own list of implementations and associated AP Identification codes.

Table C1-2 AP Identification types for an AP designed by Arm

TYPE	Connection to AP	CLASS	Notes
0x0	JTAG connection	0b0000	VARIANT field, bits [7:4] of IDR, must be non-zero.
0x0	COM-AP	0b0001	-
0x1	AMBA AHB3 bus	0b1000	-
0x2	AMBA APB2 or APB3 bus	0b1000	-
0x4	AMBA AXI3 or AXI4 bus, with optional ACE-Lite support	0b1000	Not defined in Issue A of this document.
0x5	AMBA AHB5 bus	0b1000	-
0x6	AMBA APB4 and APB5 bus	0b1000	-
0x7	AMBA AXI5 bus	0b1000	-
0x8	AMBA AHB5 with enhanced HPROT	0b1000	-
Other	Reserved	-	-

Accessing IDR

IDR can be accessed at the following address:

Offset
0xFC

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Chapter C2
The Memory Access Port

This chapter describes the implementation of the Memory Access Port (MEM-AP) and how a MEM-AP connects the DP to a debug component.

This chapter contains the following sections:

- *About the MEM-AP* on page C2-148.
- *MEM-AP functions* on page C2-152.
- *Implementing a MEM-AP* on page C2-162.
- *MEM-AP examples of pushed-verify and pushed-compare* on page C2-165.
- *MEM-AP Programmers' Model* on page C2-167.
- *MEM-AP register descriptions* on page C2-168.

For information that applies to all APs, see Chapter C1 *About the AP*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.1 About the MEM-AP

A MEM-AP provides a DAP with AP access to a memory-mapped abstraction of a set of debug resources in the system being debugged.

Note

Access to a MEM-AP might only access a register within the MEM-AP without generating a memory access to the system being debugged.

C2.1.1 The programmers' model for debug register access

The programmers' model for debug registers is a memory map. Although use of a memory bus system is not required, this abstraction enables the same programming model to be used for accessing debug registers and system memory. With this model, the debug registers might be implemented as a peripheral within the system memory space.

The debug registers in a debug component occupy one or more 4KB blocks of address space, and a system might contain several such debug components.

Although this specification permits a debug component to implement multiple 4KB blocks, most components implement a single block.

Note

Although a component can occupy only 4KB of address space, Arm recommends that the base address of each component is aligned to the largest translation granule supported by any processor that can access the component. For an Armv8 or Armv9 processor, the granule size can be up to 64KB.

Debug register files

A 4KB block of address space accessible from an AP can be referred to as a debug register file. A single AP can access multiple debug register files. There is a base standard for debug register file identification and debuggers must be able to recognize and ignore register files that they do not support.

A single MEM-AP can access a mixture of system memory and debug register files.

ROM Tables

A ROM Table is a special case of a debug register file. It is a 4KB memory block that identifies a system.

If there is only one debug component in the system to which the MEM-AP is connected, the ROM Table is optional. However, because the ROM Table contains a unique system identifier that identifies the complete SoC to the debugger, an implementation might choose to include a ROM Table even if there is only one other debug component in the system.

When a system includes more than one debug component it must include a ROM Table.

For more information, Chapter D1 *About ROM Tables* describes ROM Tables.

C2.1.2 Selecting and accessing the MEM-AP

Figure C2-1 on page C2-149 shows the implementation of a MEM-AP, and how the MEM-AP connects the DP to the debug components. Two example debug components are shown, a processor core and an ETM, together with a ROM Table. APACC accesses to the DP are passed to the MEM-AP.

The method of selecting an AP, and selecting a specific register within the selected AP, is the same for MEM-APs and JTAG-APs. See also *Selecting and accessing an AP* on page C1-143.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

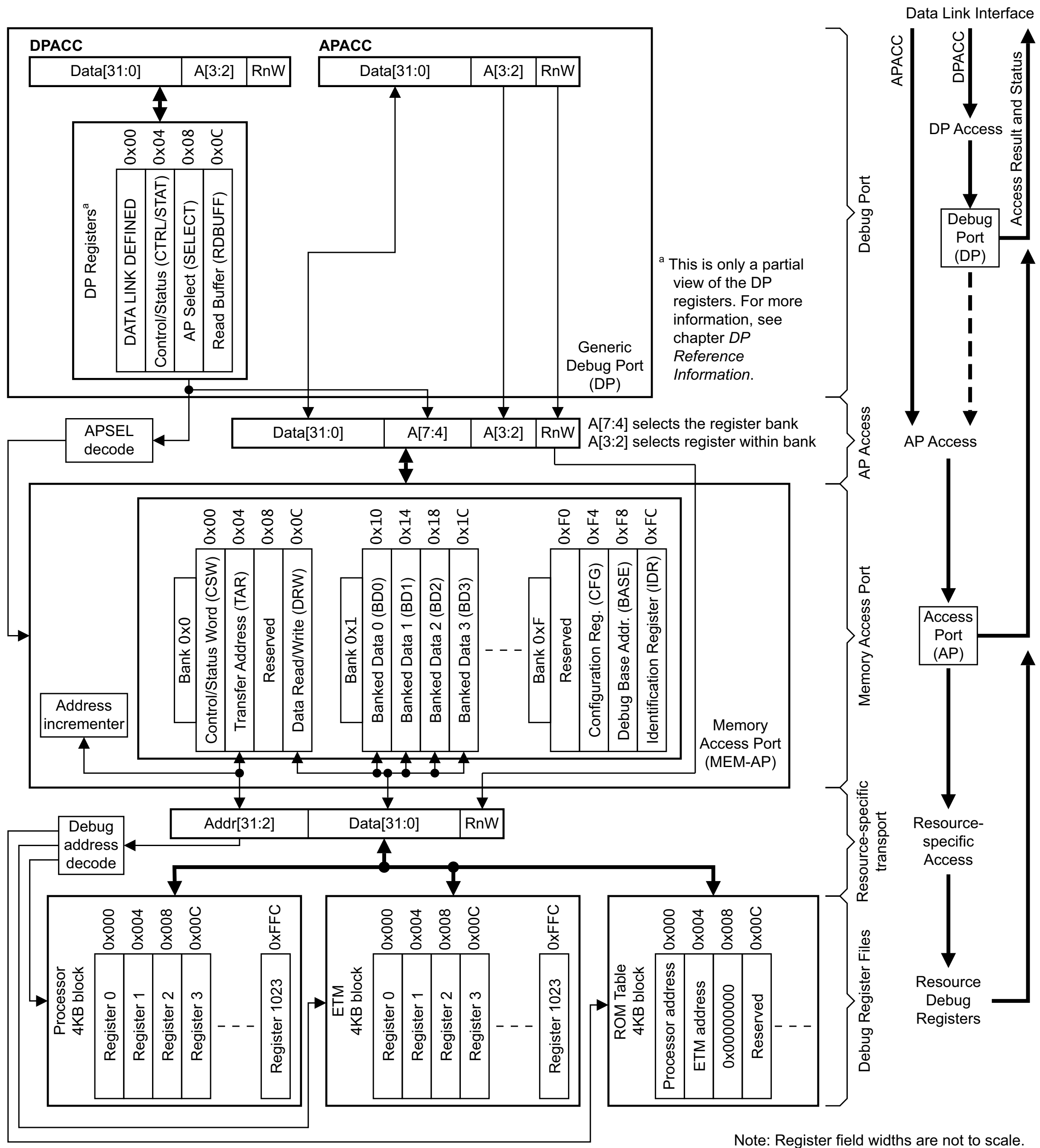


Figure C2-1 MEM-AP connecting the DP to debug components

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C2.1.3 The MEM-AP registers

The MEM-AP registers, and the memory map of the MEM-AP, are described in detail in *MEM-AP register descriptions* on page C2-168. However, a basic knowledge of the functions of these registers is required to understand the operation of the MEM-AP. The MEM-AP registers are shown in Figure C2-1 on page C2-149:

Control/Status Word register, CSW

The CSW holds control and status information for the MEM-AP.

Transfer Address Register, TAR

The TAR holds the address for the next access to the memory system, or set of debug resources that are connected to the MEM-AP. The MEM-AP can be configured so that the TAR is incremented automatically after each memory access. Reading or writing to the TAR does not cause a memory access.

Data Read/Write register, DRW

The DRW is used for memory accesses:

- Writing to the DRW initiates a write to the address specified by the TAR.
- Reading from the DRW initiates a read from the address that is specified by the TAR. When the read access completes, the value is returned from the DRW.

Banked Data Registers, BD0 to BD3

The Banked Data Registers, BD0-BD3, provide direct read or write access to a block of four words of memory, starting at the address that is specified in the TAR:

- Accessing BD0 accesses (TAR[31:4] << 4) in memory.
- Accessing BD1 accesses ((TAR[31:4] << 4) + `0x4`) in memory.
- Accessing BD2 accesses ((TAR[31:4] << 4) + `0x8`) in memory.
- Accessing BD3 accesses ((TAR[31:4] << 4) + `0xC`) in memory.

The value in TAR[3:0] is ignored in constructing the access address:

- The values of bits[3:2] of the access address depend solely on which of the four banked data registers is being accessed.
- Bits[1:0] of the access are always zero.

Configuration register, CFG

The CFG register hold information about the configuration of the MEM-AP.

Debug Base Address register, BASE

The BASE register is a pointer into the connected memory system. It points to one of:

- The start of a set of debug registers for the single connected debug component.
- The start of a ROM Table that describes the connected debug components.

Identification Register, IDR

The IDR register identifies the MEM-AP.

Note

This brief summary of the MEM-AP registers does not include cross-references to the detailed register descriptions. For more information about these registers, see *MEM-AP register descriptions* on page C2-168.

C2.1.4 MEM-AP register accesses and memory accesses

Note

In this section, an access to the debug resources is described as a memory access.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

This section summarizes all the possible APACC accesses to a MEM-AP, and covers accesses to each of the MEM-AP registers. These accesses are summarized in the following sections:

- *Accesses that do not initiate a memory access*.
- *Accesses that initiate a memory access*.
- *Accesses that support pushed transactions and the transaction counter*.

Accesses that do not initiate a memory access

APACC accesses to the following MEM-AP registers do not cause a memory access:

- The Control/Status Word register, CSW.
- The Transfer Address Register, TAR.
- The Configuration register, CFG.
- The Debug Base Address register, BASE.
- The Identification Register, IDR.

Accesses that initiate a memory access

This section introduces the APACC accesses to MEM-AP registers that initiate one or more memory accesses. These APACC accesses are:

- Accesses to the DRW register. A memory access is initiated, using the address that is held in the TAR.
- Accesses to one of the Banked Data Registers, BD0-BD3.

  The address that is used for the memory access depends on which Banked Data Register is accessed.
- Accesses to the Memory Barrier Transfer register, MBT.

However, if the MEM-AP implementation includes the Large Data Extension, and CSW.Size specifies a transfer size that is larger than a word, some DRW and BD0-BD3 accesses do not initiate a memory access, see *DRW, Data Read/Write register* on page C2-181 and *Accessing BD0-BD3* on page C2-172.

Sometimes, a single AP transaction initiates more than one memory access:

- When the transaction counter is set. See *The transaction counter* on page B1-43.
- When packed transfers are supported and enabled and the transfer size is smaller than word. See *Packed transfers* on page C2-158.

For more information, see *Packed transfers* on page C2-158.

If an AP transaction initiates one or more memory accesses, the AP transaction does not complete until one of the following occurs:

- All the memory accesses complete successfully.
- A memory access terminates with an error response. In this case, any outstanding accesses to the debug component are abandoned.
- The AP accesses are aborted using the ABORT register.

Accesses that support pushed transactions and the transaction counter

A MEM-AP supports pushed transactions and sequences of transactions to the following registers only:

- DRW, Data Read/Write register.
- BD0-BD3, Banked Data registers.

For more information, see:

- *Pushed-compare and pushed-verify operations* on page B1-44.
- *The transaction counter* on page B1-43.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.2 MEM-AP functions

This section describes the functions of a MEM-AP. These functions are controlled by the MEM-AP registers, as described in *MEM-AP register descriptions* on page C2-168.

The following sections describe functions that a MEM-AP must support:

- *Enabling access to the connected debug device or memory system*.
- *Auto-incrementing the Transfer Address Register (TAR)*.
- *Stalling accesses* on page C2-154.
- *Response to debug component errors* on page C2-155.

The following sections describe functions for which it is IMPLEMENTATION DEFINED whether a particular MEM-AP supports them:

- *Variable access size for memory accesses* on page C2-156.
- *Byte lanes* on page C2-157.
- *Packed transfers* on page C2-158.
- *Software access control* on page C2-161.
- *Implementing a MEM-AP* on page C2-162.

Note

Some of the IMPLEMENTATION DEFINED functions are inter-dependent. Their dependencies are summarized in *MEM-AP implementation requirements* on page C2-162.

C2.2.1 Enabling access to the connected debug device or memory system

Access to the debug device or memory system is controlled by Device Enable signal, **DEVICEEN**. This signal is an input to the DAP. **DEVICEEN** is normally tied HIGH, so that it is asserted even when the Debug Enable signal, **DBGEN**, is LOW, allowing the MEM-AP to be programmed even when debug is disabled.

The current value of the **DEVICEEN** signal is shown by the read-only CSW.DeviceEn flag that indicates whether the MEM-AP is able to issue transactions to the memory system to which it is connected.

When CSW.DeviceEn is `0b0`, no transactions can be issued to any address, and any access to the Data Read/Write Register or to any of BD0-BD3 immediately causes the CTRL/STAT.STICKYERR bit to be set to `0b1`. The access does not cause a MEM-AP transaction.

If there is no **DEVICEEN** signal for a device, the DeviceEn flag must Read-As-One.

C2.2.2 Auto-incrementing the Transfer Address Register (TAR)

As indicated in *The MEM-AP registers* on page C2-150, the TAR holds an address in the address map of the debug resource that is connected to the MEM-AP. This address is used as:

- The address in the debug component memory map of read or write accesses that are initiated by a read or write of the DRW.
- The base address determines the address in the debug component memory map of read or write accesses that are initiated by a read or write of one of BD0-BD3, as described in *Accesses that initiate a memory access* on page C2-151.

Software can configure the MEM-AP to auto-increment the TAR on every read or write access to the DRW. Auto-incrementing is controlled by the CSW.AddrInc field.

When auto address incrementing is enabled, the address in the TAR is updated whenever an access to the DRW is successful. However, if the DRW transaction completes with an error response, or the transaction is aborted, the TAR is not incremented.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

Accesses to BD0-BD3 never cause the TAR to auto-increment. The AddrInc field has no effect on accesses to BD0-BD3.

The permitted values of the AddrInc field are summarized in Table C2-1.

Table C2-1 Summary of AddrInc field values

AddrInc value	Description	Support required?
`0b00`	Auto-increment off	Always.
`0b01`	Increment single	Always.
`0b10`	Increment packed	If Packed transfers are supported. See *Packed transfers* on page C2-158. If Packed transfers are not supported, the value `0b10` selects the *Auto-increment off* mode and reading the AddrInc value returns `0b00`.
`0b11`	Reserved	-

The modes of operation that is associated with each of the possible settings of this field are:

Auto-increment off

The address in the TAR is not automatically incremented, and remains unchanged after any Data Read/Write Register access.

Increment single

After a successful DRW access, the address in the TAR is incremented by the size of the access. For information about different access sizes, see *Variable access size for memory accesses* on page C2-156.

Note

It is IMPLEMENTATION DEFINED whether a MEM-AP supports transfer sizes other than Word. If a MEM-AP only supports word transfers and Increment single is selected, the TAR always increments by four after a successful DRW transaction.

Increment packed

Setting AddrInc to `0b10`, Increment packed, enables packed transfers, which pack multiple halfword or byte memory accesses into a single word APACC access. Packed transfers are described in more detail in *Packed transfers* on page C2-158.

It is IMPLEMENTATION DEFINED whether a MEM-AP supports packed transfers, but:

- An implementation that supports transfers smaller than a word must support packed transfers.
- Packed transfers cannot be supported on a MEM-AP that only supports word transfers.

When packed transfer operation is enabled and the transfer size is smaller than a word, each DRW access causes multiple memory accesses, and the value in the TAR is auto-incremented correctly after each memory access. For example:

- For packed accesses with a CSW.Size value of `0b001`, denoting halfword (16-bits) transfers, each DRW read access generates two data bus transfers. The value in the TAR is incremented by `0x2` after each successful data bus transfer. As described in *Packed transfers* on page C2-158, the two halfword values from the two reads are packed into a single 32-bit word that is returned through the APACC.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- With packed accesses enabled, and a CSW.Size value of `0b000`, denoting byte transfers, a single DRW write operation generates four 8-bit data bus transfers, and the TAR is incremented by `0x1` after each of these transfers.

Automatic address increment is only guaranteed to operate on the 10 least significant bits of the address that is held in the TAR. Whether it is possible to auto increment bit [10] and beyond is IMPLEMENTATION DEFINED, which means that auto address incrementing at a 1KB boundary is IMPLEMENTATION DEFINED. For example, if the TAR is `0x14A4`, and the access size is word, successive accesses to the DRW increment TAR to `0x14A8`, `0x14AC`, and in steps of 4 bytes up to the end of the 1KB range at `0x17FC`. The auto-increment behavior on the next DRW access is IMPLEMENTATION DEFINED.

C2.2.3 Stalling accesses

A MEM-AP access by the DP to the DRW register or one of BD0-BD3 might not complete until the required memory access is completed. Therefore, to be able to support slow connections, a MEM-AP must support stalling accesses, which do not have to be completed within a fixed number of cycles.

An example of the importance of stalling accesses can be found in the Armv7 Debug Architecture, which specifies a mode of operation where accesses to the *Data Transfer Registers* (DTRs) and *Instruction Transfer Register* (ITR) do not complete until the processor is ready to accept new data. The following sequence describes how a processor that complies with the Armv7 Debug Architecture, and an ADIv5 DAP that comprises a MEM-AP and a JTAG-DP, might co-operate to inform the debugger that it has to retry an access because of such a condition.

1. The initial conditions are:
   - The processor core is idle and configured to stall accesses to its ITR and DTRs when it is not ready to accept new data.
   - The DP SELECT register addresses a MEM-AP with a connection to the processor.
   - The AP TAR addresses the ITR of the processor core.
2. The debugger writes a first instruction to the ITR:
   a. The debugger performs an AP write to DRW with the first instruction to execute:
      - The AP is ready, so the DP returns an OK/FAULT ACK response.
      - In the Update-DR state, the DP initiates a transfer to the AP.

   b. The TAR addresses the ITR on the processor, and the AP access consists of a write to the DRW. Therefore, the AP initiates a write to the ITR through its connection to the processor.

   c. The core accepts the transfer, because the processor is idle and the instruction complete flag, InstrCompl, is `0b1`.

   d. The transfer completes.

   e. The core starts to execute the instruction from the ITR. InstrCompl is set to `0b0`.

   Note

   The ACK value OK/FAULT is issued before the transfer is accepted by the core.

3. The debugger writes a second instruction to the ITR:
   a. The debugger performs an AP write to DRW with the next instruction to execute:
      - The AP is ready, so the DP returns an OK/FAULT ACK response.
      - In the Update-DR state, the DP initiates a transfer to the AP.

   b. The TAR has not changed, and the AP initiates a second write to the ITR through its connection to the processor.

   c. The core is still executing the first instruction (InstrCompl is `0b0`) and cannot accept the transfer.

   d. The transfer cannot complete, and the AP remains busy.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

ACK returns the value OK/FAULT because the AP is ready to accept a new transfer. The AP does not know that the core is not able to accept the transfer until it attempts the transfer.

4. The debugger writes a third instruction to the ITR:
   a. The debugger performs an AP write to DRW with the next instruction to execute:
      - The AP is not ready, so the DP returns a WAIT ACK response.
      - In the Update-DR state, the DP discards the AP access request, because the AP was not ready at Capture-DR.
   b. The debugger might retry the AP write until the DP returns the ACK value OK/FAULT instead of WAIT in the Capture-DR state to signal that the first instruction has completed.
5. When the core completes the first instruction, the following happens:
   a. InstrCompl is set to `0b1`.
   b. The external debug interface on the core is now ready to accept the second instruction.
   c. The AP transfer from stage 3 is accepted by the core, and the second instruction is written to the ITR.
   d. The core starts to execute the second instruction. InstrCompl is set to `0b0` again.
   e. Because the AP transfer is complete, the AP returns to the ready state.
6. The debugger retries writing the third instruction to the ITR:
   a. The debugger performs an AP write to DRW with the third instruction:
      - The AP is ready, so the DP returns an OK/FAULT ACK response.
      - In the Update-DR state, the DP initiates a transfer to the AP.
   b. The TAR has not changed. The AP initiates another write to the ITR through its connection to the processor.
   c. The response to the AP write attempt depends on whether the processor has finished processing the last instruction that was written to the ITR:
      - If the processor is idle (InstrCompl is `0b1`), the AP transfer completes, writing a new instruction to the ITR. The core starts to execute the new instruction, and the AP returns to the ready state. This stage, stage 6, of the debug session is repeated for the next instruction from the debugger.
      - If the processor is still processing the previous instruction, InstrCompl is `0b0`. The processor cannot accept the transfer and the AP remains busy. The debug session repeats stage 4.

C2.2.4 Response to debug component errors

If the MEM-AP receives an error response from a debug component, it returns an error to the DP. As a result of this error, the DP sets the STICKYERR flag. For more information about error handling flags, see *Sticky flags and DP error responses* on page B1-41.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.2.5 Variable access size for memory accesses

It is IMPLEMENTATION DEFINED whether a MEM-AP supports memory access sizes other than word (32-bit).

If a MEM-AP implementation does not support the Large Data Extension, but does support various access sizes, it must support word, halfword, and byte accesses.

Note

The ADI specification does not require a MEM-AP to support access sizes other than word. If a MEM-AP can access other memory, such as system memory. However, Arm recommends that it supports other access sizes as well.

For more information, see *MEM-AP implementation requirements* on page C2-162.

The access size is controlled by the CSW.Size field. Table C2-2 shows the access size options.

Table C2-2 Size field values when the MEM-AP supports different access sizes

Size value, CSW.Size	Memory access size	Support required?
`0b000`	Byte (8-bits)	No
`0b001`	Halfword (16-bits)	No
`0b010`	Word (32-bits)	Yes[a]
`0b011`[b]	Doubleword (64-bits)	No
`0b100`[b]	128-bits	No
`0b101`[b]	256-bits	No
`0b110` - `0b111`	Reserved	-

a. On a MEM-AP implementation that does not support access sizes other than word, the Size field is read-only, and always returns the value `0b010`.

b. Supported by the MEM-AP Large Data Extension, see *MEM-AP Large Data Extension* on page C2-163. If the extension is not implemented, this value is reserved.

When a CSW.Size specifies a size that is smaller than a word, the resulting data access is returned in byte lanes. See *Byte lanes* on page C2-157 for more information.

Caution

If BD0-BD3 are accessed with CSW.Size set to any size other than word or doubleword, behavior is UNPREDICTABLE.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.2.6 Byte lanes

A MEM-AP that supports memory transfers of less than 32-bits uses byte lanes for the data transfers between the DRW and the debug component. Which byte lanes are used depends on:

- The memory transfer size, which is specified by the CSW.Size field, see *Variable access size for memory accesses* on page C2-156.
- The two least significant bits of the TAR, TAR[1:0].

If supported, packed transfers also use byte lanes for byte and halfword transfers, as described in *Packed transfers* on page C2-158.

Table C2-3 shows how byte lanes are used in DRW.

Table C2-3 Byte-laning of memory accesses from DRW

CSW[2:0], Size	TAR[1:0]	Access data
`0b000`, byte	`0b00`	DRW[7:0]
	`0b01`	DRW[15:8]
	`0b10`	DRW[23:16]
	`0b11`	DRW[31:24]
`0b001`, halfword	`0b00`	DRW[15:0]
	`0b10`	DRW[31:16]
	`0bX1`	IMPLEMENTATION DEFINED[a]
`0b010`, word	`0b00`	DRW[31:0]
	`0b1X`, `0bX1`	IMPLEMENTATION DEFINED[a]

a. IMPLEMENTATION DEFINED behavior is one of the following:

Unaligned portions of the address are ignored. For example, an unaligned word access to `0x8003` accesses the 32-bit value at `0x8000`.

The access is faulted, and the MEM-AP returns an error response.

The access is made to the unaligned address specified in TAR[31:0], and the result is packed as if packed transfers were enabled, see *Packed transfers* on page C2-158. The data transfer might be split into more than one memory access across the connection to the debug component.

For example, an unaligned word access to `0x8003` accesses the bytes at `0x8003`, `0x8004`, `0x8005`, and `0x8006`. This word access might generate four byte-wide accesses to memory, or the accesses to bytes `0x8004` and `0x8005` might be performed as a single halfword (16-bit) access.

Big-endian support

The byte lane with the lowest address corresponds to the least significant byte of DRW or BD0-BD3, and can be described as little-endian.

Previous versions of this manual described a variant of the MEM-AP which supported an alternative byte-lane scheme, where the byte lane with the lowest address corresponded to the most significant byte of DRW or BD0-BD3, or big-endian. bit[0] of the CFG register was used to describe whether the MEM-AP was little-endian or big-endian. ADIv5.2 obsoletes this scheme.

If the target uses a big-endian memory arrangement, the external debugger must treat the values that are passed through the MEM-AP accordingly.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C2.2.7 Packed transfers

Whether a MEM-AP supports packed transfers is IMPLEMENTATION DEFINED. If packed transfers are supported, they are enabled by setting the auto address increment field, CSW.AddrInc, to `0b10` (Increment packed). See *Auto-incrementing the Transfer Address Register (TAR)* on page C2-152.

When packed transfers are enabled, each access to the DRW results in one of the following actions, depending on the value of the CSW.Size field, see *Variable access size for memory accesses* on page C2-156:

- If CSW.Size = `0b010` (word), there is a single word (32-bit) access.
- If CSW.Size = `0b001` (halfword), there are 2 halfword (16-bit) accesses.
- If CSW.Size = `0b000` (byte), there are 4 byte (8-bit) accesses.

Use of packed transfers with CSW.Size set to a transfer size larger than word is UNPREDICTABLE.

Note

Packing occurs before any pushed comparisons are made. Pushed comparisons are made and the STICKYCMP flag is set to `0b1`, if necessary, only when a complete word of data has been packed into the DRW. See *Pushed-compare and pushed-verify operations* on page B1-44 for a description of pushed comparisons.

When packed transfers are enabled, after each successful memory access the address held in the Transfer Address Register is automatically updated by the access size.

Accesses are always made in increasing memory address order:

- For write accesses to memory, data is unpacked from the DRW in byte-lanes that depend on the memory address of each write access.
- For read accesses, data is packed into the DRW in byte-lanes that depend on the memory address of each read access.

The byte lanes for data packing and unpacking are the same as the byte lanes that are described in Table C2-3 on page C2-157, as shown in the following examples:

- Example C2-1, *Halfword packed write operation*.
- Example C2-2 on page C2-159, *Byte packed write operation* on page C2-159.
- Example C2-3 on page C2-159, *Halfword packed read operation* on page C2-159.

Note

The descriptions in these examples assume that each memory access completes successfully. If any access terminates with an error response, the sequence is halted at that point, and the MEM-AP returns an error to the DP.

Example C2-1 Halfword packed write operation

This example describes a single word (32-bit) write access to DRW on a MEM-AP with the following settings:

- CSW.Size = `0b001`, specifying halfword (16-bit) memory accesses.
- CSW.AddrInc = `0b10`, specifying packed transfer operation.
- TAR[31:0] = `0x00000000`, the base address of the access.

Two write transfers are made. The halfword entries in Table C2-3 on page C2-157 define the byte lanes for these accesses. The accesses are made in the following order:

1. TAR[1] == `0b0`, so DRW[15:0] is written to address `0x00000000`.

   After this transfer, the value in the TAR is increased by the transfer size of 2, and becomes `0x00000002`.

2. TAR[1] == `0b1`, so DRW[31:16] is written to address `0x00000002`.

   After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes `0x00000004`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Example C2-2 Byte packed write operation

This example describes a single word (32-bit) write access to DRW on a MEM-AP with the following settings:

- CSW.Size = `0b000`, specifying byte (8-bit) memory accesses
- CSW.AddrInc = `0b10`, specifying packed transfer operation
- TAR[31:0] = `0x00000002`, the base address of the access.

Four write transfers are made. The *byte* entries in Table C2-3 on page C2-157 define the byte lanes for these accesses. The accesses are made in the following order:

1. TAR[1:0] == `0b10`, so DRW[23:16] is written to address `0x00000002`.

   After this transfer, the value in the TAR is increased by the transfer size, 1, and becomes `0x00000003`.

2. TAR[1:0] == `0b11`, so write DRW[31:24] is written to address `0x00000003`.

   After this transfer, the value in the TAR is increased by the transfer size, 1, and becomes `0x00000004`.

3. TAR[1:0] == `0b00`, so write DRW[7:0] is written to address `0x00000004`.

   After this transfer, the value in the TAR is increased by the transfer size, 1, and becomes `0x00000005`.

4. TAR[1:0] == `0b01`, so write DRW[15:8] is written to address `0x00000005`.

   After this transfer, the value in the TAR is increased by the transfer size, 1, and becomes `0x00000006`.

Example C2-3 Halfword packed read operation

This example describes a single word (32-bit) read access to DRW on a MEM-AP with the following settings:

- CSW.Size = `0b001`, specifying halfword (16-bit) memory accesses.
- CSW.AddrInc = `0b10`, to give packed transfer operation.
- TAR[31:0] = `0x00000002`, to define the base address of the access.

Two read transfers are made. The little-endian *halfword* entries in Table C2-3 on page C2-157 define the byte lanes for these accesses. The accesses are made in the following order:

- TAR[1] == `0b1`, so read a halfword from address `0x00000002`, and pack this value into DRW[31:16].

  After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes `0x00000004`.

- TAR[1] == `0b0`, so read a halfword from address `0x00000004`, and pack this value into DRW[15:0].

  After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes `0x00000006`.

- The complete word has been read into the DRW, and the APACC read access completes.

The optional DP transaction counter, described in *The transaction counter* on page B1-43, enables an external debugger to make a single AP transaction request that generates multiple AP transactions. Each of these transactions transfers a single word (32-bits) of data, and the TAR is incremented automatically between the transactions. If the MEM-AP supports memory accesses smaller than word and packed transfers and packed transfer operation is enabled, each of the AP transactions that are driven by the transaction counter is split into multiple memory accesses. For example, if the transaction counter is programmed to generate eight word accesses, and the MEM-AP is programmed to make packed byte transfers, a total of 32 memory accesses of 1 byte are made.

C2.2.8 Completer Memory Ports

A MEM-AP can include a Completer memory port, which can be used by an external bus transaction Requester to access the area of memory that is requested by the MEM-AP. For example, the external bus transaction Requester can be permitted to access the debug registers of the system to which the MEM-AP is connected.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

If a MEM-AP implements a Completer memory port, Completer memory port accesses are multiplexed with DAP accesses. Completer memory port accesses have bit[31] of the access address forced to zero. A debug component can use the value of this address bit to distinguish between Completer memory port accesses and DAP accesses.

For more information about MEM-AP memory addressing, see *BASE, Debug Base Address register* on page C2-168.

Note

The DAP can emulate a Completer memory port access by setting bit [31] of the TAR to `0b0`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.2.9 Software access control

It is IMPLEMENTATION DEFINED whether the CSW register includes the debug software access enable flag CSW.DbgSwEnable.

The CSW.DbgSwEnable flag can be applied as follows:

Using DbgSwEnable to control a Completer memory port

If a MEM-AP implements a Completer memory port, the DbgSwEnable flag can be used to enable or disable the port as shown in Table C2-4. For information about Completer memory ports, see *Completer Memory Ports* on page C2-159.

Table C2-4 Using DbgSwEnable to control a Completer memory port

Value of DbgSwEnable	Effect on Completer memory port
`0b0`	Disabled.
`0b1`	Enabled. This value is the value after a reset.

Using DbgSwEnable to control software access to debug resources

The DbgSwEnable flag can drive a system-level signal, **DBGSWENABLE**. This signal gates software access to debug resources. For example, in a processor that complies with the Armv7 Debug Architecture, some CP14 registers are not accessible when **DBGSWENABLE** is LOW. For more information, see the *Arm Architecture Reference Manual, Armv7-A and Armv7-R edition*.

Table C2-5 Using DbgSwEnable to control software access to debug resources

Value of DbgSwEnable	Corresponding value of the DBGSWENABLE signal
`0b0`	LOW.
`0b1`	HIGH. This value is the value after a reset.

If neither of these applications is implemented, CSW.DbgSwEnable is RAZ.

If CSW.DbgSwEnable is implemented and the MEM-AP is disabled, CSW.DbgSwEnable must be treated as one.

Caution

Arm strongly recommends not setting CSW.DbgSwEnable to zero. If CSW.DbgSwEnable is implemented, setting it to zero can cause software that is executing on the target to malfunction.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.3 Implementing a MEM-AP

This section gives information about the implementation of a MEM-AP and contains the following:

- *IMPLEMENTATION DEFINED features of a MEM-AP implementation*.
- *MEM-AP implementation requirements*.
- *MEM-AP Extensions* on page C2-163.

C2.3.1 IMPLEMENTATION DEFINED features of a MEM-AP implementation

The following features of a MEM-AP implementation are IMPLEMENTATION DEFINED:

- Whether the MEM-AP supports data bus access sizes other than word size.
- Whether the MEM-AP supports packed transfers.
- Whether the MEM-AP includes the *MEM-AP Large Physical Address Extension* on page C2-163, which implements support for addresses larger than 32 bits.
- Whether the MEM-AP includes *MEM-AP Barrier Operation Extension* on page C2-163, which implements support for barrier operations.
- Whether the MEM-AP supports the features that are described in *Software access control* on page C2-161.

These implementation choices affect the following register fields:

- CSW.{DbgSwEnable, Mode, AddrInc, Size}.
- CFG.{LD, LA, BE}.

In addition, the CSW register can include the following optional fields that are not described elsewhere in this chapter:

CSW.Prot and CSW.Type, bits[30:24] and bits[15:12]

These fields can be implemented to provide a bus access control mechanism. If implemented, it enables a debugger to specify flags for a memory access. The permitted values and their significance are IMPLEMENTATION DEFINED because they relate to the underlying bus architecture. These bits must reset to a valid access type and Arm strongly recommends that these bits are reset to a useful access type. This reset value might not be zero. For example:

- If the bus supports privileged and non-privileged accesses, the reset value of this field must select privileged accesses.
- If the bus supports code and data accesses, the reset value must select data accesses.
- If the bus supports both Secure and Non-secure address spaces, CSW.Prot and CSW.Type must reset to select Non-secure addresses.

CSW.SPIDEN, bit[23]

This field can be implemented to indicate whether the MEM-AP can generate secure accesses.

Several reference implementation options for implementers and users of MEM-APs when connecting to standard memory interfaces are defined in Appendix E1 *Standard Memory Access Port Definitions*.

C2.3.2 MEM-AP implementation requirements

The descriptions that are given in the section *MEM-AP functions* on page C2-152 indicate several areas where the MEM-AP functionality is IMPLEMENTATION DEFINED. However, the IMPLEMENTATION DEFINED features are inter-dependent. These dependencies are summarized here.

In a MEM-AP:

- The options for the size of data bus accesses are:
  - Support word (32-bit) accesses only.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

— Support word (32-bit), halfword (16-bit), and byte (8-bit) accesses, and optionally support larger access sizes.

No other combinations of supported access sizes are permitted. For more information, see *Variable access size for memory accesses* on page C2-156.

- If access sizes smaller than word are not supported, packed transfers are not supported. Otherwise, it is IMPLEMENTATION DEFINED whether packed transfers are supported. For more information, see *Packed transfers* on page C2-158.
- It is IMPLEMENTATION DEFINED whether access sizes larger than 32-bit are supported. If larger access sizes are not supported, CFG.LD is RAZ. For more information, see *MEM-AP Large Data Extension*.
- It is IMPLEMENTATION DEFINED whether addresses larger than 32-bit are supported. If larger addresses are not supported, CFG.LA is RAZ. For more information, see *MEM-AP Large Physical Address Extension*.
- It is IMPLEMENTATION DEFINED whether barrier operations are supported. If barrier operations are not supported, CSW.Mode is RAZ. For more information, see *MEM-AP Barrier Operation Extension*.

C2.3.3 MEM-AP Extensions

The following subsections summarize the effects of the optional MEM-AP Extensions.

MEM-AP Large Physical Address Extension

The MEM-AP Large Physical Address Extension provides address spaces of up to 64-bits.

Implementing this extension changes the format of the following MEM-AP registers:
- *BASE, Debug Base Address register* on page C2-168.
- *CFG, Configuration register* on page C2-175.
- *CSW, Control/Status Word register* on page C2-178.
- *DRW, Data Read/Write register* on page C2-181.
- *TAR, Transfer Address Register* on page C2-183.

MEM-AP Large Data Extension

The MEM-AP Large Data Extension can support 32-bit, 64-bit, 128-bit, or 256-bit accesses, in addition to optional 8-bit and 16-bit accesses.

The following registers have different formats to support this extension:
- *CSW, Control/Status Word register* on page C2-178.
- *DRW, Data Read/Write register* on page C2-181.
- *BD0-BD3, Banked Data registers* on page C2-171.
- *CFG, Configuration register* on page C2-175.

Although the extension can support 64-bit, 128-bit, and 256-bit accesses, it does not require an implementation to support all these access sizes. If the CSW.Size field is written with a value corresponding to a size that is not supported, or with a reserved value:
- A read of the field returns a value corresponding to a supported size.
- MEM-AP behavior corresponds to the value returned by the read of the CSW.Size field.

MEM-AP Barrier Operation Extension

The MEM-AP Barrier Operation Extension provides support for barrier operations. If the bus supports a weak memory ordering model, then barrier operations must create order.

The following registers are new or have different formats to support this extension:
- *CSW, Control/Status Word register* on page C2-178.
- *MBT, Memory Barrier Transfer register* on page C2-183.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

MEM-AP Memory Tagging Extension

The MEM-AP Memory Tagging Extension updates the following in the MEM-AP programmers model:

- CFG1.
- T0TR.
- CSW.MTE is added.

When memory tagging of accesses is disabled, system read accesses do not request the Allocation tag from the memory system.

When memory tagging of accesses is disabled, system write accesses do not update the Allocation tag.

When memory tagging of accesses is enabled, system read accesses request the Allocation tag from the memory system and store the tag received in T0TR, in the position defined by the following equations:

- ADDR_LSB = CFG1.TAG0GRAN
- ADDR_MSB = CFG1.TAG0GRAN + (log2(32/CFG1.TAG0SIZE) - 1)
- ADDR_OFFSET = Address_accessed[ADDR_MSB:ADDR_LSB]
- T0TR_LSB = ADDR_OFFSET * CFG1.TAG0SIZE
- T0TR_MSB = (ADDR_OFFSET * CFG1.TAG0SIZE) + (CFG1.TAG0SIZE - 1)
- T0TR[T0TR_MSB:T0TR_LSB] = Allocation_tag

Other bits of T0TR are unchanged.

When memory tagging of accesses is enabled, system write accesses Update the Allocation tag from the memory system using data from T0TR, from the position defined by the following equations:

- ADDR_LSB = CFG1.TAG0GRAN
- ADDR_MSB = CFG1.TAG0GRAN + (log2(32/CFG1.TAG0SIZE) - 1)
- ADDR_OFFSET = Address_accessed[ADDR_MSB:ADDR_LSB]
- T0TR_LSB = ADDR_OFFSET * CFG1.TAG0SIZE
- T0TR_MSB = (ADDR_OFFSET * CFG1.TAG0SIZE) + (CFG1.TAG0SIZE - 1)
- Allocation_tag = T0TR[T0TR_MSB:T0TR_LSB]

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.4 MEM-AP examples of pushed-verify and pushed-compare

A DP might support pushed operations, as described in *Pushed-compare and pushed-verify operations* on page B1-44. However, these operations involve interaction between the DP and an AP, because each pushed operation requires an AP read, which, in the case of a MEM-AP, requires a read from the connected debug memory system. This section gives some examples of pushed operations on a DP that is connected to a MEM-AP.

C2.4.1 Example of using a pushed-verify operation on a MEM-AP

The following pushed-verify mechanism verifies the contents of system memory:

1. Make sure that the MEM-AP CSW register is set up to increment the TAR after each access.
2. Write the start address of the memory region that is to be verified to the TAR.
3. Write a series of expected values as AP transactions. On each write transaction, the DP issues an AP read access, compares the result against the value that is supplied in the AP write transaction, and sets the CTRL/STAT.STICKYCMP bit if the values do not match.

   The TAR is incremented on each transaction.

In this way, the series of values that are supplied is compared against the contents of the memory region, and STICKYCMP is set to `0b1` if they do not match.

C2.4.2 Example of using a pushed-find operation on a MEM-AP

The following pushed-find mechanism searches system memory for a particular word:

1. Make sure that the MEM-AP CSW register is set up to increment the TAR after each access.
2. Write the start address of the debug register region that is to be searched to theTAR.
3. Repeatedly write the value to be searched for as an AP write transaction to the DRW. On each transaction, the MEM-AP reads the location indicated by the TAR.

   The return value is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to `0b1`. If they do not match, the TAR is incremented.

Pushed-find can be combined with byte lane masking to search for specific bytes.

For an example of how the transaction counter can refine this search operation, see *Example of using the transaction counter for a pushed-compare operation on a MEM-AP* on page C2-166.

Pushed-find without address incrementing can be used to poll a single location, for example to test the value of a flag after completion of an operation.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.4.3 Example of using the transaction counter for a pushed-compare operation on a MEM-AP

The transaction counter can refine the pushed-compare search operation that is described in *Example of using a pushed-find operation on a MEM-AP* on page C2-165. Pushed-compare enables searching system memory for a particular word, or, when used with byte lane masking, specific bytes. The transaction counter enables using a single AP write transaction to search an area of memory.

To perform a search under the control of the transaction counter:

1. Make sure that the MEM-AP CSW register is set up to increment the TAR after each access.
2. Write the start address of the debug register region that is to be searched to theTAR.
3. Write to the transaction counter field, CTRL/STAT.TRNCNT to indicate the required number of repeat accesses. This value defines the size of the region to be searched.
4. Write the search value as an AP write to the DRW. The MEM-AP repeatedly reads the location indicated by the TAR. The value that is returned by each read is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to `0b1` and the operation completes.
   - The TAR is incremented.
   - If the transaction counter is non-zero, it is decremented.

   The operation completes when either the STICKYCMP flag is set to `0b1` or after the final read when the transaction counter was zero.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.5 MEM-AP Programmers' Model

Table C2-6 shows a memory map of the MEM-AP registers, and indicates where they are described in detail.

All registers that are listed in Table C2-6 are required in every MEM-AP implementation.

Reserved addresses in the register memory map are RES0.

Using the Debug Port to access Access Ports on page A1-28 explains how to access AP registers.

Table C2-6 MEM-AP programmers' model

Offset	Type	Name	Description
0x00	RW	CSW	See *CSW, Control/Status Word register* on page C2-178.
0x04 - 0x08	RW	TAR	See *TAR, Transfer Address Register* on page C2-183. If the implementation includes the Large Physical Address Extension, the word at offset 0x04 represents the least significant word of the transfer address, and the word at offset 0x08 represents the most significant word. If the implementation does not include the Large Physical Address Extension, the word at offset 0x04 represents the transfer address, and the word at offset 0x08 is RES0.
0x0C	RW	DRW	See *DRW, Data Read/Write register* on page C2-181.
0x10 - 0x1C	RW	BD0-BD3	See *BD0-BD3, Banked Data registers* on page C2-171.
0x20	IMP DEF	MBT	See *MBT, Memory Barrier Transfer register* on page C2-183. The entries in this row only apply if the implementation includes the Barrier Operation Extension. Otherwise, this register is reserved, RES0.
0x24 - 0x2C	-	RES0	Reserved for future use.
0x30	RW	T0TR	See *T0TR, Tag 0 Transfer register* on page C2-186.
0x34 - 0xDC	-	-	Reserved, RES0.
0xE0	RO	CFG1	See *CFG1, Configuration register 1* on page C2-177.
0xE4 - 0xEC	-	-	Reserved, RES0.
0xF0	RO	BASE	See *BASE, Debug Base Address register* on page C2-168. If the implementation includes the Large Physical Address Extension, the word at this offset represents the most significant word of the debug base address. If the implementation does not include the Large Physical Address Extension, the word at this offset is RES0.
0xF4	RO	CFG	See *CFG, Configuration register* on page C2-175.
0xF8	RO	BASE	See *BASE, Debug Base Address register* on page C2-168. If the implementation includes the Large Physical Address Extension, the word at this offset represents the least significant word of the debug base address. If the implementation does not include the Large Physical Address Extension, the word at this offset represents the entire debug base address.
0xFC	RO	IDR	See *IDR, Identification Register* on page C1-144.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C2.6 MEM-AP register descriptions

This section describes the MEM-AP registers:

- *BASE, Debug Base Address register*.
- *BD0-BD3, Banked Data registers* on page C2-171.
- *CFG, Configuration register* on page C2-175.
- *CSW, Control/Status Word register* on page C2-178.
- *DRW, Data Read/Write register* on page C2-181.
- *MBT, Memory Barrier Transfer register* on page C2-183.
- *TAR, Transfer Address Register* on page C2-183.

C2.6.1 BASE, Debug Base Address register

The BASE characteristics are:

Purpose

BASE provides an index into the connected memory-mapped resource. This index value points to one of the following:

- The start of a set of debug registers.
- A ROM Table that describes the connected debug components.

To discover information about the debug components that are connected to the MEM-AP, a debugger can examine the four Component ID registers CIDR0-CIDR3 in the *Arm® CoreSight™ Architecture Specification*, which are at offset `0xFF0` from the base address. To examine CIDRn, the debugger writes its address, base address + `0xFF0` + n×4, to the TAR and reads the DRW register. The return value allows the debugger to determine the component type of the connected component, which is one of the following:

- ROM Table.
- Debug component.
- Other.

The ADIv5 architecture specification does not specify requirements for the type of component pointed to by BASE.

For more information about CIDR0-CIDR3, see the *Arm® CoreSight™ Architecture Specification*.

Usage constraints

The following constraints apply:

- If the bus supports both Secure and Non-secure address spaces, the BASE register is defined to be a Non-secure address. Whether the ROM Tables are also accessible in the Secure address space is IMPLEMENTATION DEFINED.
- A debugger must handle the following situations as non-fatal errors:
  - The base address that is specified by BASEADDR is a faulting location.
  - The four words starting at (base address + `0xFF0`) are not valid Component ID registers.
  - An entry in the ROM Table points to a faulting location.
  - An entry in the ROM Table points to a memory block that does not have a set of four valid Component ID registers at offset `0xFF0`.

  Typically, a debugger issues a warning if it encounters one of these situations. However, Arm recommends that it continues operating. An example of an implementation that might cause errors of this type is a system with static base address or ROM Table entries that enable entire subsystems to be disabled, for example by a tie-off input, packaging choice, fuse, or similar.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

BASE is accessible as follows:

Default
RO

Configurations

In the 64-bit register implementation, the two words making up BASE are not contiguous in the MEM-AP programmers' model.

Early implementations of DAPs had different implementations of the Debug Base Address register, as described in *Legacy format of the BASE register* on page C2-171. The legacy format is a 32-bit register at offset `0xF8`.

When BASE is implemented as a 64-bit register, it can specify any address in a 64-bit physical address space. However:

- ROM Tables support only 32-bit signed offset values.
- Armv7-A processors with the MMU disabled, and Armv7-R, Armv6-M, Armv7-M, and Armv8-M processors can access only a 32-bit physical address space.

Arm recommends that all debug components:

- Are located in the bottom 4GB of the physical address space.
- Are located in one 2GB half of the physical address space.

When BASE is implemented as a 32-bit register and the MEM-AP implements a dedicated connection between the DAP and a set of debug registers, the address map of the connection must be aliased into two logical 2GB segments. Segmentation enables the device to distinguish two types of access:

- Debugger-initiated accesses, which address the logical segment with TAR[31]=`0b1`.
- System-initiated accesses, which, if permitted, address the logical segment with TAR[31]=`0b0`.

With such an implementation, BASEADDR[31] must be `0b1`.

Note

- Even where system-initiated accesses are not permitted, Arm recommends that the debug component address space is segmented in this way, and that debugger-initiated accesses have TAR[31]=`0b1`.
- Other systems might include IMPLEMENTATION DEFINED methods for signaling debugger accesses to system components.

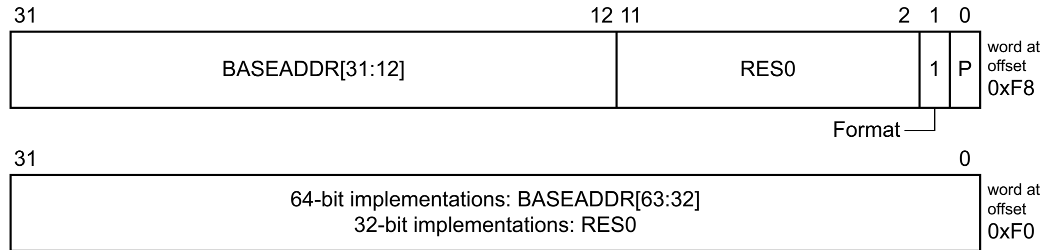
Attributes

A 32-bit or 64-bit read-only register.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Field descriptions

The BASE bit assignments are:



BASEADDR[31:12], bits[31:12] of word at offset 0xF8
BASEADDR[63:32], bits[31:0] of word at offset 0xF0, 64-bit register only

The 20 or 52 most significant bits of the base address, which is the address offset of the start of the debug register space in the memory-mapped resource, or a ROM Table address. BASEADDR is padded with the 12-bit value `0x000` to complete the 32-bit or 64-bit base address.

If BASE is implemented as a 32-bit register, the word at offset `0xF0` is RES0.

The details of the memory area pointed to by the base address depend on the number of debug components that are connected to the ADI:

- If the ADI is connected to a single debug component, as in the system that is shown in Figure A1-3 on page A1-32, the base address is the start of the debug registers for that component.

  If a debug component occupies more than one 4KB page of memory, the base address is the address of the 4KB block which contains the Peripheral ID and Component ID registers of the component.

- If the ADI is connected to more than one debug component, as in the system that is shown in Figure A1-6 on page A1-34, the base address is the address of a ROM Table, which contains the addresses of the other debug components that are connected to the interface. For information about ROM Tables, see Chapter D1 *About ROM Tables*.

  A system that contains only a single debug component might be implemented with a separate ROM Table, as shown in Figure A1-5 on page A1-33. In this case, the base address is the address of the ROM Table.

Bits[11:2] of word at offset 0xF8

Reserved, RES0.

Format, bit[1] of word at offset 0xF8

Base address register format.

This field is RAO, indicating the ADIv5 format.

Note

This bit is RAZ in one of the legacy Debug Base Address register formats, see *Legacy format of the BASE register* on page C2-171.

P, bit[0] of word at offset 0xF8

This field indicates whether a debug entry for this MEM-AP is present:

`0b0` No debug entry is present.

`0b1` Debug entry is present.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

Legacy format of the BASE register includes a description of the legacy format of BASE when there is no debug entry present.

Accessing BASE

BASE can be accessed from the MEM-AP register space:

Offset if Large Physical Address extension is not implemented	Offset if Large Physical Address extension is implemented	
	Least significant word	Most significant word
`0xF8`	`0xF8`	`0xF0`

Legacy format of the BASE register

The legacy format of the BASE register is as follows:

Legacy format when no debug entries are present

The BASE bit assignment when there are no debug entries present is:

NOTPRESENT, bits[31:0]

This field has the value `0xFFFFFFFF`, indicating that there are no debug entries.

Legacy format for specifying BASEADDR

When bit[1] of the BASE register is `0b0`, the legacy format of the register holds the base address value. In this case, the BASE bit assignments are:

BASEADDR, bits[31:12]

Bits[31:12] of the base address. Bits[11:0] of the base address are zero.

Bits[11:2]

Reserved, RAZ.

FORMAT, bit[1]

RAZ, indicating that the BASE register uses the legacy 32-bit BASE register format.

Bit[0]

Reserved, RAZ.

The legacy format is defined only for 32-bit addresses and not permitted for a MEM-AP that implements the Large Physical Address Extension.

The legacy format must not be used for new ADI designs.

C2.6.2 BD0-BD3, Banked Data registers

The BD0-BD3 register characteristics are:

Purpose

BD0-BD3 map AP accesses directly to memory accesses, without having to change the value in the TAR. Together, the four BD0-BD3 give access to four words of the memory space, starting at the address that is specified in the TAR.

Each Banked Data register holds a 32-bit data value:

- In write mode, a Banked Data register holds a value to write to memory.
- In read mode, a Banked Data register holds a value that is read from memory.

Usage Constraints

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Auto address incrementing is not performed when a Banked Data register is accessed. The value of CSW.AddrInc has no effect on Banked Data register accesses.

The Large Data Extension supports memory access size values that are greater than word size, as described in *Variable access size for memory accesses*.

- If the large data extension is implemented, accesses other than word or doubleword are UNPREDICTABLE.
- If the large data extension is not implemented, accesses other than word are UNPREDICTABLE.

The registers are accessible as follows:

Default
RW

Configurations

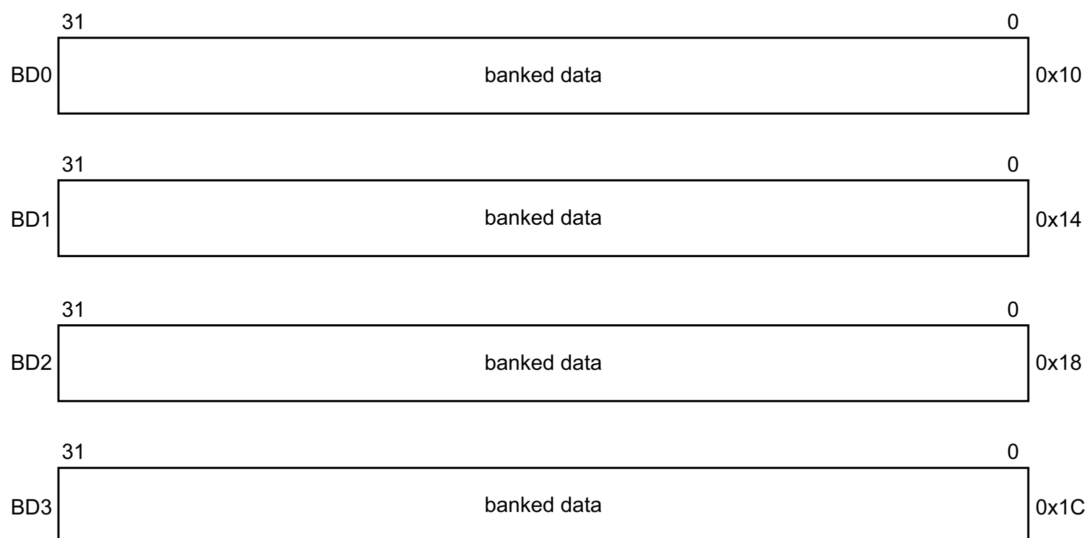
Included in all implementations.

Attributes

BD0-BD3 are four 32-bit read/write registers.

Field descriptions

TheBD0-BD3 bit assignments are:



Banked data, BD0-BD3 bits[31:0]

Data values for the current transfer.

See *Accessing BD0-BD3* for more information about BD register accesses.

Accessing BD0-BD3

BD0-BD3 can be accessed from the MEM-AP register space.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

If the Large Physical Address Extension is not implemented, BD0-BD3 can be accessed at the following offsets:

Register	Offset	Memory Address that is accessed	
		Word Access[a]	**Doubleword Access**[b]
BD0	0x10	TAR[31:4] << 4	TAR[31:4] << 4, accessed first
BD1	0x14	(TAR[31:4] << 4) + 0x4	TAR[31:4] << 4, accessed second
BD2	0x18	(TAR[31:4] << 4) + 0x8	(TAR[31:4] << 4) + 0x8, accessed first
BD3	0x1C	(TAR[31:4] << 4) + 0xC	(TAR[31:4] << 4) + 0x8, accessed second

If the Large Physical Address Extension is implemented, BD0-BD3 can be accessed at the following offsets:

Register	Offset	Memory Address that is accessed	
		Word Access[a]	**Doubleword Access**[b]
BD0	0x10	TAR[63:4] << 4	TAR[63:4] << 4, accessed first
BD1	0x14	(TAR[63:4] << 4) + 0x4	TAR[63:4] << 4, accessed second
BD2	0x18	(TAR[63:4] << 4) + 0x8	(TAR[63:4] << 4) + 0x8, accessed first
BD3	0x1C	(TAR[63:4] << 4) + 0xC	(TAR[63:4] << 4) + 0x8, accessed second

a. Bits[1:0] of the address are always 0b00.
b. Bits[2:0] of the address are always 0b000.

An access to a Banked Data register initiates an access to the memory address shown in the table. The AP access does not complete until the memory access has completed.

If the access size specified in CSW.Size is doubleword, the lower-numbered register holds the least significant word, and the higher-numbered register holds the most significant word. To access a value, a debugger must access both registers of the pair making up the doubleword, where the lower-numbered register is accessed first.

For example, if the Large Physical Address Extension is implemented, to read the doubleword value at (TAR[63:4] << 4), a debugger must:

1. Read BD0, to obtain bits[31:0] of the doubleword.
2. Read BD1, to obtain bits[63:32] of the doubleword.

When CSW.Size specifies doubleword access size, the following restrictions apply to the two required Banked Data register accesses:

- The effect of mixing reads and writes in the sequence is UNPREDICTABLE.
- If CSW is accessed in the middle of the sequence, the following behavior is IMPLEMENTATION DEFINED:
  - Whether the CSW access is successful.
  - Whether the CSW access results in an error response from the AP.
- If CSW is accessed in the middle of the sequence, that sequence is terminated, and the next access to a Banked Data register is the first access of a new sequence.

  If a write sequence is terminated, no memory write is initiated.
- The effect of not accessing the appropriate register first is UNPREDICTABLE.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

- After accessing the first Banked Data register of a pair, the effect of accessing any MEM-AP register other than CSW or the second Banked Data register of the pair is UNPREDICTABLE. Examples of sequences that lead to an UNPREDICTABLE result include:
  - Accessing BD1 and then accessing BD2.
  - Two consecutive accesses to the same Banked Data register.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.6.3 CFG, Configuration register

The CFG characteristics are:

Purpose

CFG indicates whether the MEM-AP implementation includes the Large Data and Large Physical Address Extensions.

Usage constraints

CFG is accessible as follows:

Default
RO

Configurations

CFG is included in all implementations.

Attributes

A 32-bit read-only register.

Field descriptions

The CFG bit assignments are:

Bits[31:3] Reserved, RES0.

LD, bit[2] Large Data. This bit indicates whether the MEM-AP implementation includes the Large Data Extension, which provides support for data items larger than 32-bits. LD has one of the following values

`0b0` The implementation does not support data items that are larger than 32 bits.

`0b1` The implementation includes the Large Data Extension, and supports data items larger than 32 bits.

For more information, see *MEM-AP Large Data Extension* on page C2-163.

Regardless of the value of the LD field, the MEM-AP must support word-size data items, and might support smaller data items. See also CSW.Size.

LA, bit[1] Long Address. This field indicates whether the MEM-AP implementation includes the Large Physical Address Extension, which supports physical addresses of more than 32-bits. LA has one of the following values:

`0b0` The implementation support only physical addresses of 32 bits or smaller.

Memory locations for the TAR and BASE registers, which are at offsets `0x08` and `0xF0` in the MEM-AP register map, are reserved.

`0b1` The implementation supports physical addresses with more than 32 bits:

- The TAR is a 64-bit register, at offsets `0x04` and `0x08` in the MEM-AP register map.
- The BASE register is a 64-bit register, at offsets `0xF8` and `0xF0` in the MEM-AP register map.

For more information, see *MEM-AP Large Physical Address Extension* on page C2-163.

BE, bit[0] Big-Endian. ADIv5.2 obsoletes support for big-endian MEM-AP, and this bit must RAZ. For more information, see *Big-endian support* on page C2-157.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Accessing CFG

CFG can be accessed from the MEM-AP register space:

Offset
`0xF4`

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.6.4 CFG1, Configuration register 1

The CFG1 characteristics are:

Purpose

CFG1 indicates the features of the implementation of the MEM-AP.

Usage constraints

CFG1 is accessible as follows:

Default
RO

Configurations

Included in all implementations.

Attributes

A 32-bit read-only register.

The CFG1 bit assignments are:

Bits[31:9]

Reserved, RES0.

TAG0GRAN, bits[8:4]

Memory tagging granule.

When Memory Tagging Extension implemented

`0x04` Memory tagging granule is 16 bytes.

All other values are reserved.

Otherwise

Reserved, RES0.

TAG0SIZE, bits[3:0]

Memory tagging support. The defined values of this field are:

`0x0` Memory Tagging Extension not implemented. T0TR not implemented. CSW.MTE is not implemented.

`0x4` Memory Tagging Extension implemented. Tag size is 4-bits. T0TR is implemented. CSW.MTE is implemented.

All other values are reserved.

Accessing CFG1

CFG1 can be accessed from the MEM-AP register space:

Offset
`0xE0`

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.6.5 CSW, Control/Status Word register

The CSW characteristics are:

Purpose

CSW configures and controls accesses through the MEM-AP to or from a connected memory system.

Usage constraints

Some of the fields are read-only or IMPLEMENTATION DEFINED.

The register as a whole is accessible as follows:

Default
RW

Configurations

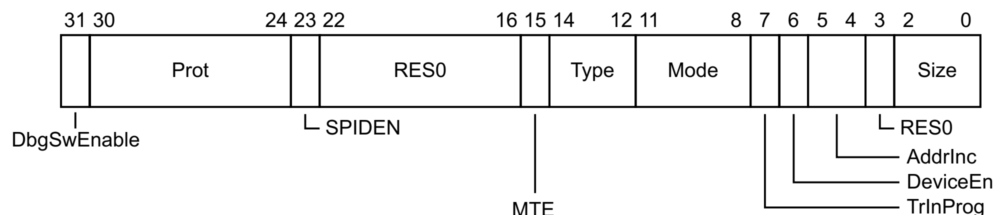
Included in all implementations.

Attributes

A 32-bit read/write register.

Field Descriptions

The CSW bit assignments are:



DbgSwEnable, bit[31]

Debug software access enable.

This field is optional. If not implemented, it is RAZ.

If implemented, it has one of the following values:

`0b0` Debug software access is disabled. If DeviceEn is `0b0`, DbgSwEnable must be ignored and treated as one.

`0b1` Debug software access is enabled.

The use of this field is IMPLEMENTATION DEFINED, see *Software access control* on page C2-161.

Prot, bits[30:24]

Used with the Type field to define the bus access protection control.

A debugger can use these fields to specify flags for a debug access. The permitted values and their significance are IMPLEMENTATION DEFINED, and depend on the underlying bus architecture. For more information, see *Implementing a MEM-AP* on page C2-162.

These fields are OPTIONAL. If not implemented, they are RES0.

SPIDEN, bit[23]

Secure Debug Enabled. This field has one of the following values:

`0b0` Secure access is disabled.

`0b1` Secure access is enabled.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

This field is optional, and read-only. If not implemented, the bit is RES0.

If CSW.DeviceEn is `0b0`, SPIDEN is ignored and the effective value of SPIDEN is `0b1`.

For more information, see *Enabling access to the connected debug device or memory system* on page C2-152.

Bits[22:16]

Reserved, RES0.

Type, bits[15:12], when Memory tagging control is not implemented.

Used with the Prot field to define the bus access protection control.

A debugger can use these fields to specify flags for a debug access. The permitted values and their significance are IMPLEMENTATION DEFINED, and depend on the underlying bus architecture. For more information, see *Implementing a MEM-AP* on page C2-162.

This field is OPTIONAL. If not implemented, it is RES0.

MTE, bit [15], when Memory tagging control is implemented.

Memory Tagging control. The possible values of this bit are:

`0b0` Memory tagging accesses disabled.

`0b1` Memory tagging accesses enabled.

When memory tagging accesses are enabled, system read and write accesses via DRW, BDx, and DARx, use T0TR for transferring tag information.

Type, bits[14:12] , when Memory tagging control is implemented.

Used with the Prot field to define the bus access protection control.

A debugger can use these fields to specify flags for a debug access. The permitted values and their significance are IMPLEMENTATION DEFINED, and depend on the underlying bus architecture. For more information, see *Implementing a MEM-AP* on page C2-162.

This field is OPTIONAL. If not implemented, it is RES0.

Mode, bits[11:8]

Mode of operation of the MEM-AP. This field has one of the following values:

`0b0000` Basic mode.

`0b0001` Barrier support enabled. For more information, see *MEM-AP Barrier Operation Extension* on page C2-163.

Other Reserved.

The set of supported modes is IMPLEMENTATION DEFINED. If the implementation supports only one mode, this field can be RO.

If this field is RW, the reset value of this field is UNKNOWN.

TrInProg, bit[7]

Transfer in progress. This field has one of the following values:

`0b0` The connection to the memory system is idle.

`0b1` A transfer is in progress on the connection to the memory system.

After an ABORT operation, debug software can read this bit to check whether the aborted transaction completed.

DeviceEn, bit[6]

Device enabled.

This field has one of the following values:

`0b0` The MEM-AP is not enabled.

`0b1` Transactions can be issued through the MEM-AP.

See *Enabling access to the connected debug device or memory system* on page C2-152.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

This field is read-only.

AddrInc, bits[5:4]

Address auto-increment and packing mode. The possible values are of this field are:

`0b00` Address auto-increment disabled.

`0b01` Address increment-single enabled.

`0b10` Address increment-packed enabled.

All other values are reserved.

The reset value of this field is UNKNOWN.

Bit[3]

Reserved, RES0.

Size, bits[2:0]

The size of the data type that is used to access the MEM-AP, as shown in Table C2-7.

It is IMPLEMENTATION DEFINED whether a MEM-AP supports access sizes other than 32-bits, and whether the Size field is RW or RO:

- If other sizes are supported, the Size field is RW, and the field indicates the size of the accesses to perform. When this field is RW, its reset value is UNKNOWN.
- If other sizes are not supported, this field is RO and it reads as `0b010` to indicate that only 32-bit accesses are supported.

Table C2-7 Size field values

Size Field	Data Type	Supported
`0b000`	Byte (8-bits)	IMPLEMENTATION DEFINED
`0b001`	Halfword (16-bits)	IMPLEMENTATION DEFINED
`0b010`	Word (32-bits)	Yes[a]
`0b011`[b]	Doubleword (64-bits)	IMPLEMENTATION DEFINED
`0b100`[b]	128-bits	IMPLEMENTATION DEFINED
`0b101`[b]	256-bits	IMPLEMENTATION DEFINED
`0b110` - `0b111`	Reserved	-

a. On a MEM-AP implementation that does not support access sizes other than word, the Size field is read-only, and always returns the value `0b010`.

b. Supported by the MEM-AP Large Data Extension, see *MEM-AP Large Data Extension* on page C2-163. The following usage constraints apply:

If the extension is not implemented, this value is reserved.

If a reserved value, or a value corresponding to an unsupported access size, is written to this field, reading the field returns the value corresponding to a supported size, and the MEM-AP behaves according to the return value.

Accessing CSW

CSW can be accessed from the MEM-AP register space:

Offset
`0x00`

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.6.6 DRW, Data Read/Write register

The DRW characteristics are:

Purpose

DRW maps the value that is passed in an AP access directly to one or more memory accesses at the address that is specified in the TAR.

The value depends on the access mode:

- In write mode, DRW holds the value to write for the current transfer to the address specified in the TAR.
- In read mode, DRW holds the value that is read in the current transfer from the address that is specified in the TAR.

The AP access does not complete until the memory access, or accesses, complete.

Usage constraints

MEM-AP implementations that include the Large Data Extension enable accessing values with a data type that is larger than the size of DRW, which requires multiple access to DRW to complete a single memory access, as shown in Table C2-8.

Memory accesses that involve multiple DRW accesses have the following limitations:

- The effect of mixing reads and writes in the sequence is UNPREDICTABLE.
- An access to CSW in the middle of a sequence terminates that sequence. The next access to DRW is the first access of a new sequence. If a write sequence is terminated, no memory write is initiated.
- After the first DRW access of the sequence, the effect of accessing any MEM-AP register other than the CSW or DRW is UNPREDICTABLE.
- Depending on the value of CSW.AddrInc, the TAR might be incremented after each DRW access. See *Auto-incrementing the Transfer Address Register (TAR)* on page C2-152.

Table C2-8 DRW access behavior for different data type sizes

Size of data type	CSW.Size	Required number of DRW accesses	Read behavior	Write behavior
8 bits[a,b]	`0b000`	1	Each read initiates a memory access and returns the value to be read using byte lanes.	Each write initiates a memory access and writes the value to be written using byte lanes.
16 bits[a,b]	`0b001`	1		
32 bits[c]	`0b010`	1	Each read initiates a memory access and returns the value to be read.	Each write initiates a memory access and writes the value to be written.
64 bits[d]	`0b011`	2	On first read: • Initiate a memory access. • Return the least significant 32-bit word of the value being read. On subsequent reads: • Do not initiate another memory access. • Return the next 32-bit word of the value being read. On the first read, the AP access does not complete until the memory access completes.	On writes before the last write: • Specifies the next 32-bit word of the value to be written, starting from the least significant word. • Do not initiate a memory access. On last write: • Specify the most significant 32-bit word of the value to be written. • Initiate a memory access. On the last write, the AP access does not complete until the memory access completes.
128 bits[d]	`0b100`	4		
256 bits[d]	`0b101`	8		

a. Support is IMPLEMENTATION DEFINED.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

b. A single access to the DRW register might result in multiple memory accesses, depending on the values of CSW.AddrInc. See *Packed transfers* on page C2-158.
c. Supported by all MEM-AP implementations.
d. Might be supported by MEM-AP applications that include the Large Data Extension. Support is IMPLEMENTATION DEFINED.

DRW is accessible as follows:

Default
RW

Configurations

The MEM-AP Large Data Extension, described in *MEM-AP Large Data Extension* on page C2-163, modifies the behavior of this register for accesses with CSW.Size set to a value larger than `0b010`.

Attributes

DRW is a 32-bit MEM-AP register.

Field descriptions

The DRW bit assignments are:

Data, bits[31:0]

Data value of the current transfer.

Accessing DRW

DRW can be accessed from the MEM-AP register space:

Offset
`0x0C`

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C2.6.7 MBT, Memory Barrier Transfer register

The MBT register characteristics are:

Purpose

MBT generates a barrier operation on the bus.

Usage constraints

If CSW.Mode has a value other than `0b0001`, writes to MBT are ignored.

MBT is accessible as follows:

Default
IMPLEMENTATION DEFINED

Configurations

If the Barrier Operation Extension is not implemented, MBT is RES0.

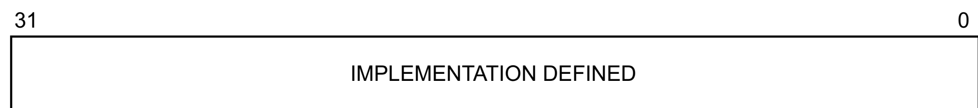
MBT is implemented only if the MEM-AP implementation includes the MEM-AP Barrier Operation Extension, see *MEM-AP Barrier Operation Extension* on page C2-163.

Attributes MBT is a 32-bit MEM-AP register.

Other properties of the register are IMPLEMENTATION DEFINED.

Field descriptions

The MBT bit assignments are:



Bits[31:0]

IMPLEMENTATION DEFINED.

Accessing MBT

MBT can be accessed from the MEM-AP register space:

Offset
`0x20`

C2.6.8 TAR, Transfer Address Register

The TAR characteristics are:

Purpose

TAR holds the memory address to be accessed through AP accesses.

Note

The address that is held in TAR represents an address in the memory system to which the MEM-AP is connected, not an address within the MEM-AP itself.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Configurations and Usage constraints

When using the DRW, TAR specifies the memory address to access:

- If the MEM-AP does not support accesses smaller than word, TAR[1:0] is RES0.
- The contents of TAR can be incremented automatically on a successful DRW access, see *Auto-incrementing the Transfer Address Register (TAR)* on page C2-152.

When accessing memory through *BD0-BD3, Banked Data registers* on page C2-171, bits [3:0] of TAR are ignored, and TAR[63:4] or TAR[31:4] specifies the base address of the 16-byte block of memory that can be accessed.

The size and reset value of TAR are as follows:

Large Physical Address Extension	TAR Size	Reset Value	
		Least significant word (offset `0x04`)	Most significant word (offset `0x08`)
No	32 bits	UNKNOWN	-
Yes	64 bits	UNKNOWN	`0x00000000`

The register is accessible as follows:

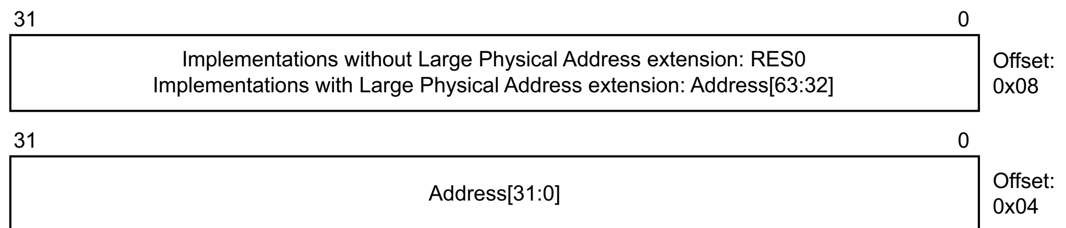
Default
RW

Attributes

A 32-bit or 64-bit MEM-AP register.

Field descriptions

The TAR bit assignments are:



Address[63:32], bits[31:0] of the register word at offset `0x08`

Most significant word of the memory address for the current transfer.

If the MEM-AP implementation does not include the Large Physical Address Extension, this field is RES0.

Address[31:0], bits[31:0] of the register word at offset `0x04`

Least significant word of the memory address for the current transfer.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Accessing TAR

TAR can be accessed from the MEM-AP register space:

Offset	
Least significant byte	Most significant byte[a]
`0x04`	`0x08`

a. Applicable only to MEM-AP implementations with a Large Physical Address Extension.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C2.6.9 T0TR, Tag 0 Transfer register

The T0TR register characteristics are:

Purpose

Stores tag values for transfers.

Usage constraints

The register is accessible as follows:

Default
RW

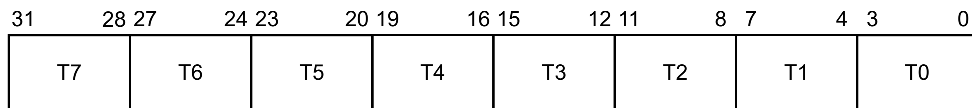
Configurations

Implemented when the Memory Tagging Extension is implemented.

Attributes A 32-bit MEM-AP register.

Field Descriptions

The T0TR bit assignments are:



T7, bits[31:28]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b111`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b111`.

This field resets to an architecturally UNKNOWN value on a Reset.

T6, bits[27:24]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b110`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b110`.

This field resets to an architecturally UNKNOWN value on a Reset.

T5, bits[23:20]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b101`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b101`.

This field resets to an architecturally UNKNOWN value on a Reset.

T4, bits[19:16]

Allocation tag value.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b100`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b100`.

This field resets to an architecturally UNKNOWN value on a Reset.

T3, bits[15:12]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b011`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b011`.

This field resets to an architecturally UNKNOWN value on a Reset.

T2, bits[11:8]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b010`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b010`.

This field resets to an architecturally UNKNOWN value on a Reset.

T1, bits[7:4]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b001`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b001`.

This field resets to an architecturally UNKNOWN value on a Reset.

T0, bits[3:0]

Allocation tag value.

On reads, holds the Allocation tag value for the most recent system memory read transaction performed with address[6:4] == `0b000`.

On writes, holds the Allocation tag value for the next system memory write transaction performed with address[6:4] == `0b000`.

This field resets to an architecturally UNKNOWN value on a Reset.

Accessing T0TR

T0TR can be accessed from the MEM-AP register space:

Offset
`0x30`

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Chapter C3
The JTAG Access Port

This chapter describes the implementation of the *JTAG Access Port* (JTAG-AP), and how a JTAG-AP provides a Debug Port connection to one or more JTAG components. The JTAG-AP is an optional component of a Debug Access Port.

This chapter contains the following sections:

- *About the JTAG-AP* on page C3-190.
- *Operation of the JTAG-AP* on page C3-195.
- *The JTAG Engine Byte Command Protocol* on page C3-198.
- *JTAG-AP register summary* on page C3-205.
- *JTAG-AP register descriptions* on page C3-206.

Note

Chapter C1 *About the AP* gives additional information about APs.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1 About the JTAG-AP

The JTAG Access Port is an optional component of a Debug Access Port (DAP). It enables up to eight legacy IEEE 1149.1 JTAG scan chains to be connected to the DAP. Each scan chain can contain any number of TAPs. However, Arm recommends that only one TAP is connected to each scan chain.

An external debugger accesses a JTAG component, which is connected to a JTAG-AP, through a JTAG scan chain. The debugger accesses this scan chain using APACC accesses to registers in the JTAG-AP. A debugger also has to access JTAG-AP registers to control the JTAG-AP, or to obtain status or identification information from the JTAG-AP.

C3.1.1 Selecting and accessing the JTAG-AP

Figure C3-1 on page C3-191 shows the implementation of a JTAG-AP and how the JTAG-AP connects the DP to up to eight JTAG devices. APACC accesses to the DP are passed to the JTAG-AP.

The method of selecting an AP, and selecting a specific register within the selected AP, is the same for MEM-APs and JTAG-APs, and is summarized in *Selecting and accessing an AP* on page C1-143.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

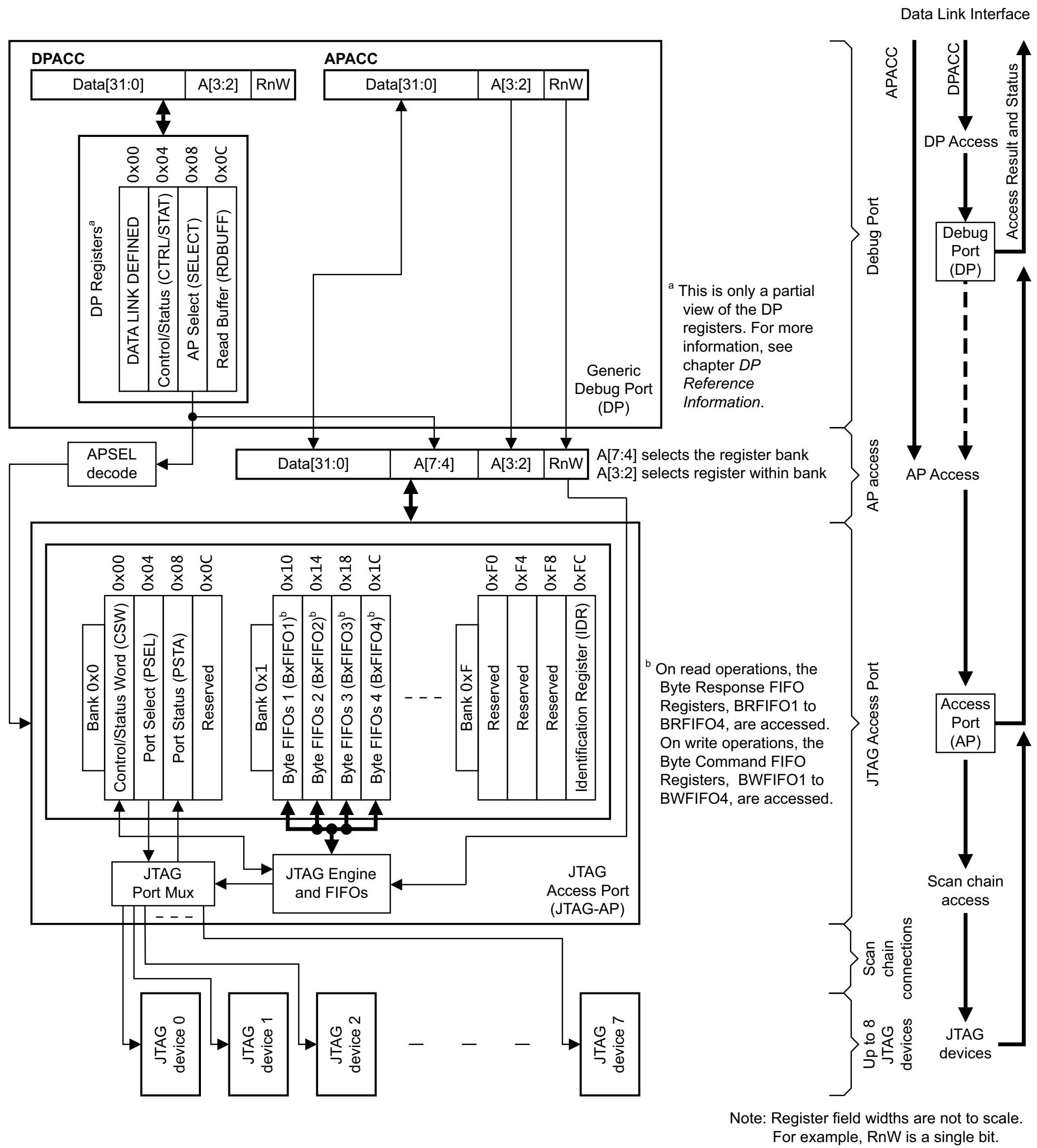


Figure C3-1 JTAG-AP connecting the DP to JTAG devices

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C3.1.2 Logical structure of the JTAG-AP

A JTAG-AP:

- — Interprets a sequence of command bytes from the Command FIFO.
- — Drives standard JTAG signals to the JTAG Port Multiplexer.
- — Receives the **TDO** signal from the Port Multiplexer.
- — Generates a response and passes it to the Response FIFO.

A JTAG-AP comprises:

- The JTAG Port Multiplexer, which multiplexes up to eight JTAG ports to the JTAG Engine. In addition to forwarding the standard JTAG signals to and from each port, it provides control and status signals for each port.
- Byte Command and Response FIFOs, which enable efficient use of the JTAG Engine.

  The following rules apply to the FIFO sizes:
  - — The Response FIFO must be 7 bytes deep.
  - — The Command FIFO must be at least 4 bytes deep. Although the Command FIFO can be up to 7 bytes deep, there is unlikely to be any advantage in having a Command FIFO that is larger than 4 bytes.
- The JTAG-AP registers, which can be divided into three groups:
  - — An Identification Register.
  - — Control and status registers.
  - — FIFO access registers.

Figure C3-2 on page C3-193 shows the JTAG-AP structure.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

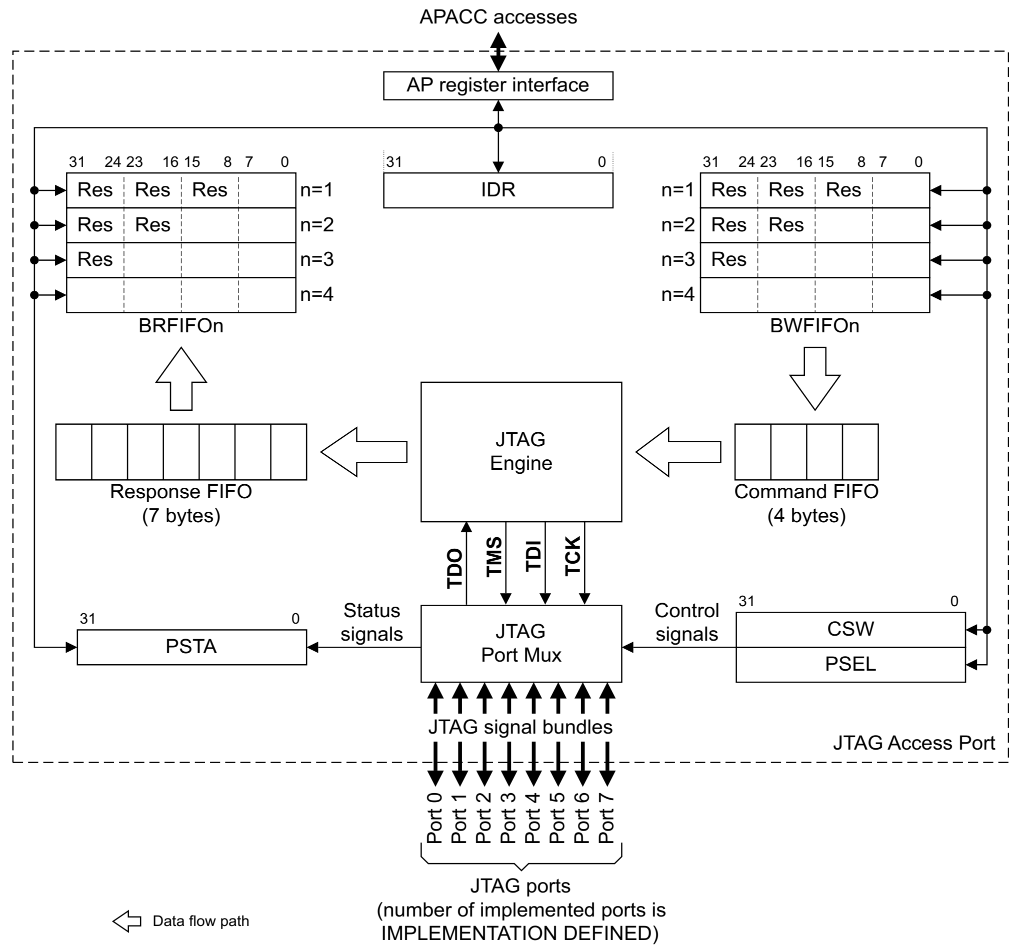


Figure C3-2 Structure of the JTAG Access Port (JTAG-AP)

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

C3.1.3 JTAG port signals

The signal bundle between the JTAG Port Multiplexer and each implemented JTAG port includes:

- The standard IEEE 1149.1 JTAG signals.
- Port control and status signals.

Table C3-1 gives the full signal list that applies to each implemented port.

Table C3-1 JTAG Access Port JTAG port signals

Signal	Direction[a]	Description	Notes
TCK	Out	Test Clock	JTAG IEEE 1149.1 standard signals.
TMS	Out	Test Mode Select	
TDI	Out	Test Data In	
TDO	In	Test Data Out	
TRST*	Out	Test Reset	Active LOW JTAG IEEE 1149.1 standard signal.
nSRSTOUT	Out	Subsystem Reset	Active LOW.
SRSTCONNECTED	In	Subsystem Reset Connected	Tie-off configuration signals to the JTAG Port Multiplexer.
PORTCONNECTED	In	Port Connected	
PORTENABLED	In	Port Enabled	Can be deasserted by the JTAG subsystem, for example when the connected TAP powers down.

a. Signal directions are given relative to the JTAG Port Multiplexer in the JTAG-AP.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.2 Operation of the JTAG-AP

The JTAG-AP communicates with the device using standard JTAG signals and scan chains. This operation is controlled by the JTAG Engine. The Engine includes a serializer that takes TDI data out of the Command FIFO and returns TDO data to the Response FIFO, see Figure C3-2 on page C3-193.

The external debugger:

1. Encodes JTAG commands and data into the JTAG Engine Byte Command Protocol, which is described in *The JTAG Engine Byte Command Protocol* on page C3-198.
2. Writes to the BWFIFOn registers to transfer the encoded JTAG Engine commands and data to the JTAG Command FIFO.
3. Reads from the BRFIFOn registers to collect JTAG TDO in response to the encoded JTAG Engine commands.
4. Decodes the actual TDO data from the response data.

The JTAG Engine provides the connection between stages 2 and 3 of this process.

Note

The JTAG-AP can connect to up to eight JTAG devices. The debugger must write to the PSEL register to select which JTAG port or ports the JTAG Port Multiplexer connects to the JTAG Engine.

The debugger can start reading data from TDO before completing writing data to TDI, as long as it has completed writing the command header. A debugger can take advantage of this principle to exchange data that exceeds the size of the command and response FIFOs.

For example:

1. The debugger writes two bytes to BWFIFO2, to specify:
   a. A TDI_TDO scan, with 64 bits of TDI data.
   b. The TDO data is to be returned to the debugger.
2. The debugger writes a word to BWFIFO4, containing the first 32 bits of TDI data.
3. The debugger reads a word from BRFIFO4, to obtain the first 32 bits of TDO data.
4. The debugger writes another word to BWFIFO4, with the next 32 bits of TDI data.
5. The debugger reads another word from BRFIFO4, to obtain the next 32 bits of TDO data.

This method provides an efficient encapsulation of the JTAG scan chain.

If the requested data is not available, a read of BRFIFOn stalls, as described in *Stalling accesses* on page C3-196. To reduce the number of stalls that are caused by AP accesses to devices with a slow clock, the debugger can write several bytes of TDI data before attempting to read the first byte of TDO data.

Operation of the JTAG-AP is described in more detail in *The JTAG Engine Byte Command Protocol* on page C3-198.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.1 Stalling accesses

AP accesses to JTAG engine FIFOs can be stalled.

As shown in Figure C3-2 on page C3-193, the JTAG Engine FIFOs comprise the following registers:

- The Byte Response FIFO Registers, BRFIFO1 to BRFIFO4.
- The Byte Command FIFO Registers, BWFIFO1 to BWFIFO4.

The JTAG Engine FIFOs are described in BRFIFO1-BRFIFO4, and can be used to access the JTAG state machine and JTAG scan chains as described in *The JTAG Engine Byte Command Protocol* on page C3-198.

AP accesses to JTAG-AP registers that do not access the JTAG Engine FIFOs cannot be stalled. As shown in Figure C3-2 on page C3-193, this rule applies to the following registers:

- CSW.
- PSEL.
- PSTA.
- IDR.

A JTAG-AP access can stall in the following situations:

Stalling read accesses

The JTAG-AP can stall read accesses to the Byte Response FIFO Registers, BRFIFO1 to BRFIFO4.

Depending which of these registers is targeted, a single register read transfers between 1 and 4 bytes of data from the byte Response FIFO. The register access stalls if the FIFO does not contain enough data. For example, if the Response FIFO only contains 2 bytes of data and a read access is performed to BRFIFO4 to transfer 4 bytes of data, the access stalls and remains stalled until there are 4 bytes of data available in the Response FIFO.

CSW.RFIFOCNT can be read to find the number of bytes of data that are available in the Response FIFO. A read of the CSW always completes immediately.

Stalling write accesses

The JTAG-AP can stall write accesses to the Byte Command FIFO Registers, BWFIFO1 to BWFIFO4.

Depending which of these registers is targeted, a single register write transfers between 1 and 4 bytes of data into the byte Command FIFO. The register access stalls if the FIFO does not contain enough free space to accept all the write data. For example, if the Command FIFO only has 1 byte free and a write access to BWFIFO3 is performed to transfer 3 bytes into the Command FIFO, the access stalls and remains stalled until the Command FIFO is able to accept the three bytes of data.

CSW.WFIFOCNT can be read to find the number of command bytes in the Command FIFO that are waiting to be processed by the JTAG Engine, which can be used to calculate the number of free bytes in the FIFO. A read of the CSW always completes immediately.

C3.2.2 Resetting connected JTAG devices or subsystems

Resets of JTAG devices or subsystems that are connected to the JTAG-AP can be triggered with the following signals:

- The **TRST*** signal for JTAG Test Resets.
- The **nSRSTOUT** signal for subsystem resets.

These signals are controlled by the CSW.TRST_OUT and CSW.SRST_OUT fields. A JTAG test reset might have to be clocked out for several **TCK** cycles with **TMS** HIGH to generate the reset. For more information see *CSW, Control/Status Word Register* on page C3-211.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.2.3 Pushed transaction and transaction counter support

A JTAG-AP supports pushed transactions and sequences of transactions to the following registers only:

- PSTA.
- BRFIFO1-BRFIFO4.

For more information, see:

- *Pushed-compare and pushed-verify operations* on page B1-44.
- *The transaction counter* on page B1-43.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.3 The JTAG Engine Byte Command Protocol

All JTAG commands, including TMS and TDI data, are written to the JTAG-AP Command FIFO through the interface that is provided by the four Byte Write FIFO Registers, BWFIFO1 to BWFIFO4. To provide high command packing, the JTAG commands are encoded as a byte protocol, and depending on which of the Byte Write FIFO registers is written to, up to 4 bytes can be written to the FIFO in a single operation. See *BWFIFO1-BWFIFO4, Byte FIFO registers for write access* on page C3-208.

Data from the **TDO** signal from the JTAG Port Multiplexor is transferred to the JTAG-AP Response FIFO. The four Byte Read FIFO Registers provide an interface to the Response FIFO. See *BRFIFO1-BRFIFO4, Byte FIFO registers for read access* on page C3-206.

In the JTAG Engine Byte Command Protocol, all commands are 1 byte (8-bits). Table C3-2 summarizes the commands and the following sections describe them in more detail. Where appropriate, the command descriptions also describe the TDO data that is produced by the command, and how it is encoded in the Byte Read FIFOs.

Table C3-2 Summary of JTAG Engine Byte Command Protocol

Bits of the Command byte								Opcode
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	
0	Opcode payload							TMS
1	0	0	Opcode payload					TDI_TDO
1	0	1	X	X	X	X	X	Reserved
1	1	0	X	X	X	X	X	Reserved
1	1	1	X	X	X	X	X	Reserved

C3.3.1 The encoding of the TMS packet

The TMS packet is a single byte. The payload of the packet holds:

- Between 1 and 5 data bits to be sent on **TMS**.
- An indication of whether **TDI** is held at 0 or at 1 while these bits are sent.

While a TMS packet is being executed, no response is captured from **TDO**. The normal use of TMS packets is to move around the JTAG state machine. See *The Debug TAP State Machine (DBGTAPSM)* on page B3-88.

Table C3-3 shows the possible encodings of a TMS packet.

Table C3-3 TMS packet encodings

Command byte								Bits of TMS data
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	
0	TDI	1	TMS[4]	TMS[3]	TMS[2]	TMS[1]	TMS[0]	5
0	TDI	0	1	TMS[3]	TMS[2]	TMS[1]	TMS[0]	4
0	TDI	0	0	1	TMS[2]	TMS[1]	TMS[0]	3
0	TDI	0	0	0	1	TMS[1]	TMS[0]	2
0	TDI	0	0	0	0	1	TMS[0]	1

When the JTAG Engine decodes a TMS packet, **TDI** is held at the value indicated by bit [6] while all the TMS data bits are sent. If you have to send **TMS** bits with different **TDI** values, you must use multiple TMS packets.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

The TMS data bits are sent LSB first, so in each row of Table C3-3 on page C3-198, TMS[0] is the first bit to be sent.

For example, the TMS packet that is used to send the **TMS** bit sequence `1-1-0-1`, while keeping **TDI** at 1, is shown in Figure C3-3. As the diagram shows, this sequence of **TMS** signals takes four **TCK** cycles.

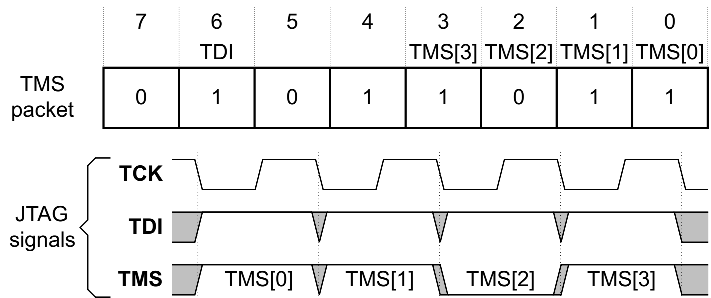


Figure C3-3 TMS packet example with TDI held at 1

To send **TMS** bit sequence `1-0`, while keeping **TDI** at 0, the TMS packet is as shown in Figure C3-4. As shown in the diagram, this sequence of **TMS** signals takes two **TCK** cycles.

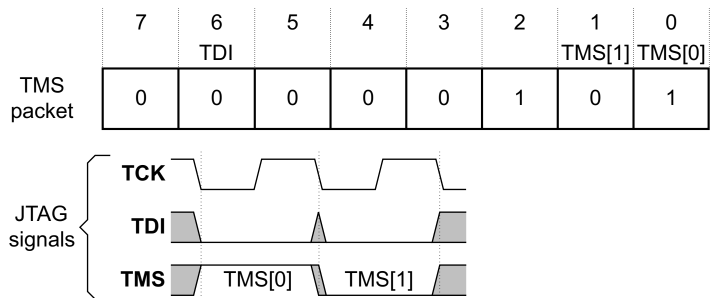


Figure C3-4 TMS packet example with TDI held at 0

C3.3.2 The encoding of the TDI_TDO packet

A TDI_TDO packet is a multi-byte packet that is at least two bytes long. It comprises:

- The TDI_TDO opcode byte.
- A second byte, that contains:
  - For short packets, of fewer than 7 TDI bits, the packed TDI bits.
  - Otherwise, the length of the packet.
- If required, between 1 and 16 extra bytes containing the TDI bits.

The following subsections describe these bytes.

The TDI_TDO opcode byte, the first byte of the packet

This byte is the packet header. It indicates the start of a TDI_TDO packet, and contains information about the command subtype. Figure C3-5 shows the format of this byte.

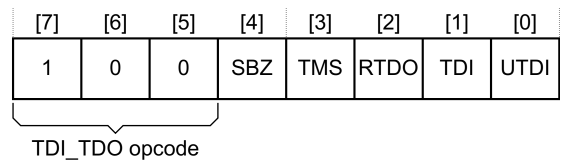


Figure C3-5 TDI_TDO first byte (opcode) format

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits[3:0] are control bits that define the TDI_TDO subtype. Table C3-4 describes all the bits of the first byte of the TDI_TDO packet.

Table C3-4 Format of the first byte of TDI_TDO (opcode)

Bit	Name	Value[a]	Description
[7]	TDI_TDO	`0b1`	The value of these bits indicates whether this byte is the first byte of a TDI_TDO packet
[6]		`0b0`	
[5]		`0b0`	
[4]	-	SBZ	Reserved, Should Be Zero.
[3]	TMS	-	**TMS** value to use on the last cycle of the scan: `0b0` = **TMS** LOW on last cycle `0b1` = **TMS** HIGH on last cycle. For all earlier cycles of the scan: • If the previous TMS or TDI_TDO packet finished with **TMS** high for the last cycle, it is UNPREDICTABLE whether **TMS** is HIGH or LOW for this scan. • In all other cases, **TMS** is LOW.
[2]	RTDO	-	Read **TDO**. This bit determines whether **TDO** values returned during the scan are captured and placed in the Response FIFO: `0b0` = Do not capture **TDO** `0b1` = Capture **TDO**. **Caution** Do not set this bit to `0b1` if more than one JTAG port is selected and enabled. If you do, the **TDO** values captured are UNKNOWN.
[1]	TDI	-	**TDI** value to use throughout the scan if the UTDI bit is `0b1`: `0b0` = hold **TDI** signal LOW throughout the scan `0b1` = hold **TDI** signal HIGH throughout the scan The value of the TDI bit is ignored if the UTDI is `0b0`.
[0]	UTDI	-	Use TDI bit. This bit determines whether the TDI bits to be used during the scan are supplied in the other bytes of the TDI_TDO packet, or whether the TDI bit, bit[1], specifies the **TDI** signal to use throughout the scan: `0b0` = TDI bits for the scan are supplied in the other bytes of the TDI_TDO packet. `0b1` = The TDI bit, bit[1], determines the **TDI** signal to use throughout the scan. If this bit is `0b1`, no TDI data is included in the TDI_TDO packet[b].

a. Given for bits that have a fixed value for the TDI_TDO first byte.

b. When the Packed format is used for the second byte of the packet, certain bits of that byte are designated as TDI data bits. If UTDI = `0b1`, however, the value of these bits is ignored, as described in *The TDI_TDO length byte, the second byte of the packet* on page C3-201. There is no advantage in using the packed format when UTDI = `0b1`, but it is possible to do so.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

The TDI_TDO length byte, the second byte of the packet

There are two alternative formats for the second byte of the TDI_TDO packet:

Normal If bit[7] of the TDI_TDO length byte is zero, the byte is in the normal length byte format, and specifies the length of the scan, which can be any value between 1 and 128 bits. Bits [6:0] of the byte give the length in bits of the required scan, minus one, as shown in Figure C3-6.

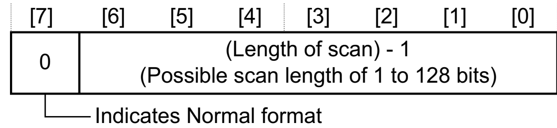


Figure C3-6 TDI_TDO second byte (length byte), normal format

When the TDI_TDO length byte is in the normal format:

- If the UTDI bit of the first byte of the TDI_TDO packet is `0b0`, the TDI data for the scan is packed into extra bytes of the packet, that follow the length byte. See *The data bytes, the remaining byte or bytes of the packet* on page C3-202 for more information.
- If the UTDI bit of the first byte of the TDI_TDO packet is `0b1`, no TDI data is required for the scan, and the length byte is the last byte of the packet. If the UTDI bit is `0b1`, the TDI_TDO packet is always two bytes long.

See *The TDI_TDO opcode byte, the first byte of the packet* on page C3-199 for more information about the UTDI bit.

Packed If bit[7] of the second byte of the TDI_TDO packet is one, the byte is in the packed length byte format, and contains between 1 and 6 bits of TDI data:

- The length of the required scan is implied by the data in bits[6:0].
- If the UTDI bit of the first byte of the TDI_TDO packet is `0b0`, the TDI data for the scan is packed into the least significant bits of the length byte.
- The second byte is the last byte of the TDI_TDO packet, meaning the packet is 2 bytes long.

Note

The packed format of the TDI_TDO length byte can only be used if the required scan contains 6 bits or less.

Figure C3-7 shows the permitted contents of the length byte when the packed format is used.

	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
Scan length = 6 bits	1	1	TDI[5]	TDI[4]	TDI[3]	TDI[2]	TDI[1]	TDI[0]
Scan length = 5 bits	1	0	1	TDI[4]	TDI[3]	TDI[2]	TDI[1]	TDI[0]
Scan length = 4 bits	1	0	0	1	TDI[3]	TDI[2]	TDI[1]	TDI[0]
Scan length = 3 bits	1	0	0	0	1	TDI[2]	TDI[1]	TDI[0]
Scan length = 2 bits	1	0	0	0	0	1	TDI[1]	TDI[0]
Scan length = 1 bit	1	0	0	0	0	0	1	TDI[0]

Indicates Packed format

Figure C3-7 TDI_TDO second byte (length byte), packed format

The packed format for the TDI_TDO length byte is summarized in Table C3-5 on page C3-202.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

Table C3-5 TDI_TDO length byte, packed format

Scan length (bits)	Must be zero bits	Data start flag	TDI data for scan[a]
6	None	Bit[6] = `0b1`	Bits[5:0]
5	Bit[6] = `0b0`	Bit[5] = `0b1`	Bits[4:0]
4	Bits[6:5] = `0b00`	Bit[4] = `0b1`	Bits[3:0]
3	Bits[6:4] = `0b000`	Bit[3] = `0b1`	Bits[2:0]
2	Bits[6:3] = `0b0000`	Bit[2] = `0b1`	Bits[1:0]
1	Bits[6:2] = `0b00000`	Bit[1] = `0b1`	Bit[0]

a. When the UTDI bit of the first byte of the TDI_TDO packet is `0b1`, the values of these bits are ignored.

When the TDI_TDO length byte is in the packed format:

- If the UTDI bit of the first byte of the TDI_TDO packet is `0b0`, the data that is packed into bits[5:0] of the length byte determines the value of the **TDI** signal during the scan. Bit[0] of the length byte always holds TDI[0], meaning that this bit determines the **TDI** signal value for the first **TCK** cycle of the scan.
- If the UTDI bit of the first byte of the TDI_TDO packet is `0b1`, the data that is packed into bits[5:0] of the length byte only indicates the length of the required scan, and does not affect the value of the **TDI** signal during the scan. For example, if the complete length byte is `0b10001XXX`, referring to Figure C3-7 on page C3-201 shows that a scan of 3 bits is required. The **TDI** signal value, for all three bits, is the TDI value from the first byte of the packet, see Table C3-3 on page C3-198.

See also *The TDI_TDO opcode byte, the first byte of the packet* on page C3-199.

Note

The packed format can be used when the UTDI bit in the first byte of the packet is `0b1`. However, there is no advantage in using the packed format when UTDI = `0b1`, because the normal format is easier to use, and the TDI_TDO packet is 2 bytes long, whichever format is used.

The data bytes, the remaining byte or bytes of the packet

If the TDI_TDO opcode byte is `0x00`, and the length byte is in the normal format, the TDI_TDO packet is more than two bytes long. In this case:

- Bits[6:0] of the length byte contain the required scan length minus one, in bits.
- The TDI data for the scan is packed into extra bytes of the packet.

The packing of TDI data uses as few bytes as possible, and the least significant bit of TDI data, TDI[0], is always bit[0] of the first data byte. TDI[0] is the **TDI** signal value for the first **TCK** cycle of the scan.

The number of data bytes required is the length of the scan divided by eight, rounded up to an integer value. In the last data byte, any bits that are not required for TDI data must be `0b0`. For example, a scan of 21 cycles requires 3 data bytes, giving a total TDI_TDO packet size of 5 bytes. Figure C3-8 on page C3-203 shows the formatting of the complete TDI_TDO packet for this example.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
TDI_TDO opcode, with TMS = 1 and RTDO = 1	1	0	0	0	1	1	0	0
Length byte, normal format (bit [7] = 0)	0	0	0	1	0	1	0	0
First data byte	TDI[7]	TDI[6]	TDI[5]	TDI[4]	TDI[3]	TDI[2]	TDI[1]	TDI[0]
Second data byte	TDI[15]	TDI[14]	TDI[13]	TDI[12]	TDI[11]	TDI[10]	TDI[9]	TDI[8]
Third data byte	0	0	0	TDI[20]	TDI[19]	TDI[18]	TDI[17]	TDI[16]

Figure C3-8 TDI_TDO formatting example. Complete packet for a scan of 21 TCK cycles

The bit assignments for the bytes shown in Figure C3-8 are:

Byte 1, the opcode byte

Bits[7:5] The TDI_TDO opcode, `0b100`.

Bit[3] The TMS bit. A value of `0b1` indicates that **TMS** must be HIGH for the last cycle of the scan.

Bit[2] The RTDO bit. A value of `0b1` indicates that TDO data must be captured during the scan.

Byte 2, the length byte

Bit[7] A value of `0b0` indicates that this length byte is in normal format.

Bits[6:0] The value of ((length of scan) - 1). This field has the value `0b0010100`, which is 20, meaning the scan length is 21 bits.

Bytes 3 and 4, the first and second data bytes

These bytes contain TDI[15:0], the TDI data for the first 16 cycles of the scan.

Byte 5, the third data byte

This byte contains TDI[20:16], the TDI data for the final five cycles of the scan. Any bits that are not required for TDI data must be `0b0`, so bits [7:5] = `0b000`.

C3.3.3 Response bytes from a TDI_TDO packet

If the Read TDO (RTDO) bit, which is bit [2] of a TDI_TDO packet header, is `0b1`, the value of the **TDO** signal is captured for each **TCK** cycle of the scan. This captured TDO data is packed into bytes and each byte is inserted into the Response FIFO when it is completed.

Figure C3-8 shows a TDI_TDO packet with RTDO = `0b1`.

Note

If more than one JTAG port is selected and enabled, the returned TDO values are UNKNOWN.

The number of bytes of TDO data that is inserted in the Response FIFO is the scan length divided by 8, rounded up to an integer value. When the scan length is not an exact multiple of 8, the last byte of returned data is padded with bits having a value of `0b0`.

The scan stalls if the Response FIFO is full when a byte of TDO data is ready for insertion.

Figure C3-9 on page C3-204 shows the formatting of the TDO data bytes transferred to the Response FIFO for a scan of 21 **TCK** cycles where TDO capture is enabled.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
First data byte transferred to Response FIFO	TDO[7]	TDO[6]	TDO[5]	TDO[4]	TDO[3]	TDO[2]	TDO[1]	TDO[0]
Second data byte transferred to Response FIFO	TDO[15]	TDO[14]	TDO[13]	TDO[12]	TDO[11]	TDO[10]	TDO[9]	TDO[8]
Third data byte transferred to Response FIFO	0	0	0	TDO[20]	TDO[19]	TDO[18]	TDO[17]	TDO[16]

Figure C3-9 TDI_TDO response data formatting example. Scan of 21 TCK cycles

If the RTDO bit is `0b0`, no response bytes are placed in the Response FIFO.

For details about the *Read TDO* (RTDO) bit, see *The TDI_TDO opcode byte, the first byte of the packet* on page C3-199.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.4 JTAG-AP register summary

Table C3-6 shows the memory map of the JTAG-AP registers, and indicates where they are described in detail.

For more information on accessing AP registers, see *Using the Debug Port to access Access Ports* on page A1-28.

All the registers that are listed in Table C3-6 are required in every JTAG-AP implementation.

Table C3-6 Summary of JTAG Access Port (JTAG-AP) registers

Register	Address	Access	Reset value	Notes
CSW	0x00	RW	Depends on the state of the connected signals when the register is read[a]	CSW
PSEL	0x04	RW	UNKNOWN	PSEL
PSTA	0x08	RW	0x00000000	PSTA
-	0x0C	-	-	Reserved, RES0
BRFIFO1-BRFIFO4	0x10	RO	[b]	Read, single entry
		WO	-	Write, single entry
	0x14	RO	[b]	Read, two entries
		WO	-	Write, two entries
	0x18	RO	[b]	Read, three entries
		WO	-	Write, three entries
	0x1C	RO	[b]	Read, four entries
		WO	-	Write, four entries
-	0x20 - 0xF8	-	-	Reserved, RES0
IDR	0xFC	RO	IMPLEMENTATION DEFINED	See *IDR, Identification Register* on page C1-144

a. For details about the reset values of individual fields, see the CSW field descriptions.

b. Accesses to Byte FIFO Read Registers stall until data is available in the FIFO. Therefore they do not have reset values.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.5 JTAG-AP register descriptions

This section describes each of the JTAG-AP registers. Table C3-6 on page C3-205 shows these registers, and indexes the full register descriptions in this section. The following subsections describe these registers:

- *BRFIFO1-BRFIFO4, Byte FIFO registers for read access*.
- *BWFIFO1-BWFIFO4, Byte FIFO registers for write access* on page C3-208.
- *CSW, Control/Status Word Register* on page C3-211.
- *PSEL, Port Select register* on page C3-214.
- *PSTA, Port Status Register* on page C3-216.

C3.5.1 BRFIFO1-BRFIFO4, Byte FIFO registers for read access

The BRFIFO1-BRFIFO4 characteristics are:

Purpose

Enable 1 byte, 2 bytes, 3 bytes, or 4 bytes to be read in parallel from the Response FIFO.

Register	BRFIFO1	BRFIFO2	BRFIFO3	BRFIFO4
Address	`0x10`	`0x14`	`0x18`	`0x1C`
Number of bytes read from Response FIFO	1	2	3	4

The JTAG Engine Byte Command protocol that is used for the commands and responses is described in *The JTAG Engine Byte Command Protocol* on page C3-198.

Usage constraints

BRFIFO1-BRFIFO4 are mapped to the same JTAG-AP register addresses as BRFIFO1-BRFIFO4. The AP accesses the BRFIFOn on read operations, and the BWFIFOn on write operations.

An AP transaction that reads more responses than are available in the Response FIFO stalls until enough data is available to match the request. To check the number of response bytes that are available, read the CSW.RFIFOCNT field before initiating an AP transaction to read from the Response FIFO.

BRFIFO1-BRFIFO4 are accessible as follows:

Default
RO

Configurations

Included in all implementations.

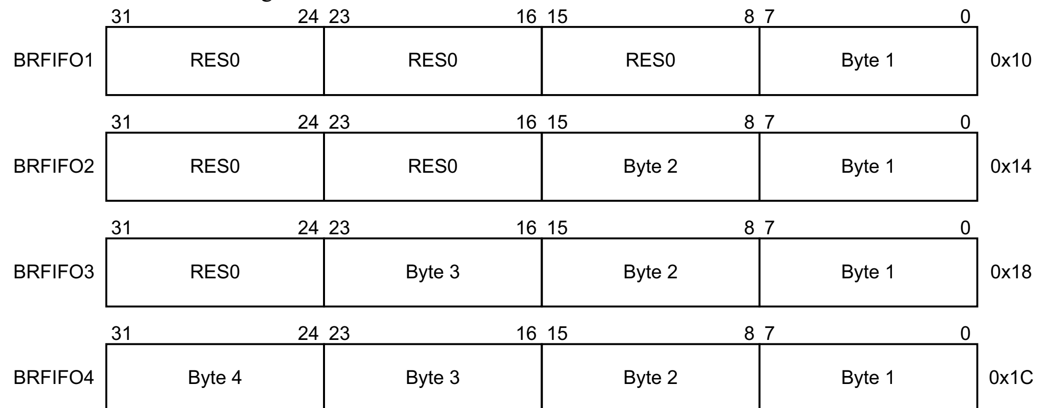
Attributes

A set of four 32-bit RO registers.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Field descriptions

The BRFIFO1-BRFIFO4 bit assignments are:



BRFIFO1 bits[31:8]

RES0

Byte 1, BRFIFO1 bits[7:0]

The first byte to be read from the Response FIFO.

BRFIFO2 bits[31:16]

RES0

Byte 2, BRFIFO2 bits[15:8]

The second byte to be read from the Response FIFO.

Byte 1, BRFIFO2 bits[7:0]

The first byte to be read from the Response FIFO.

BRFIFO3 bits[31:24]

RES0

Byte 3, BRFIFO3 bits[23:16]

The third byte to be read from the Response FIFO.

Byte 2, BRFIFO3 bits[15:8]

The second byte to be read from the Response FIFO.

Byte 1, BRFIFO3 bits[7:0]

The first byte to be read from the Response FIFO.

Byte 4, BRFIFO4 bits[31:24]

The fourth byte to be read from the Response FIFO.

Byte 3, BRFIFO4 bits[23:16]

The third byte to be read from the Response FIFO.

Byte 2, BRFIFO4 bits[15:8]

The second byte to be read from the Response FIFO.

Byte 1, BRFIFO4 bits[7:0]

The first byte to be read from the Response FIFO.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Accessing BRFIFO1-BRFIFO4

BRFIFO1-BRFIFO4 can be accessed from the JTAG-AP register space:

Access	Offset			
	BRFIFO1	BRFIFO2	BRFIFO3	BRFIFO4
Read	0x10	0x14	0x18	0x1C
Number of bytes read	1	2	3	4

C3.5.2 BWFIFO1-BWFIFO4, Byte FIFO registers for write access

The BWFIFO1-BWFIFO4 characteristics are:

Purpose

Enable 1 byte, 2 bytes, 3 bytes, or 4 bytes to be written in parallel to the Command FIFO.

Register	BWFIFO1	BWFIFO2	BWFIFO3	BWFIFO4
Address	0x10	0x14	0x18	0x1C
Number of bytes written to Command FIFO	1	2	3	4

The JTAG Engine Byte Command protocol that is used for the commands and responses is described in *The JTAG Engine Byte Command Protocol* on page C3-198.

Usage constraints

BWFIFO1-BWFIFO4 are mapped to the same JTAG-AP register addresses as BRFIFO1-BRFIFO4. The AP accesses the BRFIFOn on read operations, and the BWFIFOn on write operations.

An AP transaction that writes more commands than there is space for in the Command FIFO stalls until there is enough space in the Command FIFO. Space in the Command FIFO is freed as commands are executed by the JTAG Engine. To check the number of commands already present in the Command FIFO, read the CSW.WFIFOCNT field before initiating an AP transaction to write to the Command FIFO. The number of additional commands you can write to the FIFO can be calculated by subtracting the return value from the size of the Command FIFO.

BWFIFO1-BWFIFO4 registers are accessible as follows:

Default
WO

Configurations

Included in all implementations.

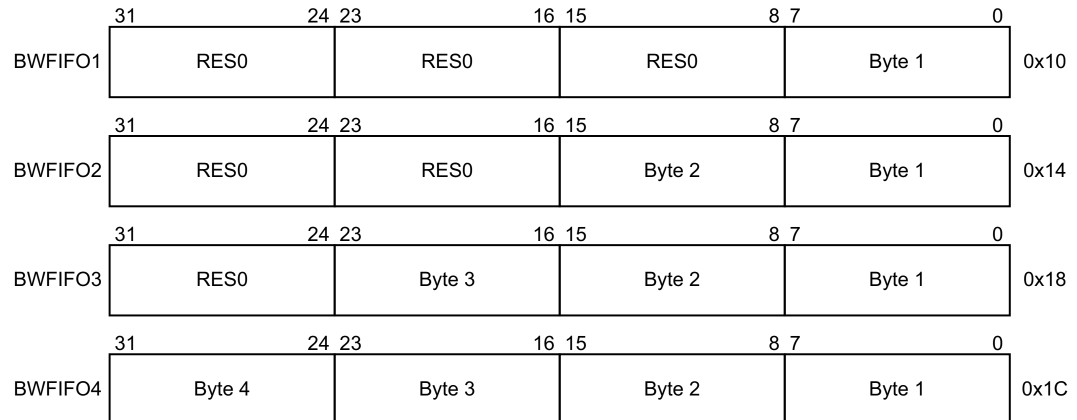
Attributes

A set of four 32-bit WO registers.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Field descriptions

The BWFIFO1-BWFIFO4 bit assignments are:



BWFIFO1 bits[31:8]

RES0

Byte 1, BWFIFO1 bits[7:0]

The first byte to be written to the Command FIFO.

BWFIFO2 bits[31:16]

RES0

Byte 2, BWFIFO2 bits[15:8]

The second byte to be written to the Command FIFO.

Byte 1, BWFIFO2 bits[7:0]

The first byte to be written to the Command FIFO.

BWFIFO3 bits[31:24]

RES0

Byte 3, BWFIFO3 bits[23:16]

The third byte to be written to the Command FIFO.

Byte 2, BWFIFO3 bits[15:8]

The second byte to be written to the Command FIFO.

Byte 1, BWFIFO3 bits[7:0]

The first byte to be written to the Command FIFO.

Byte 4, BWFIFO4 bits[31:24]

The fourth byte to be written to the Command FIFO.

Byte 3, BWFIFO4 bits[23:16]

The third byte to be written to the Command FIFO.

Byte 2, BWFIFO4 bits[15:8]

The second byte to be written to the Command FIFO.

Byte 1, BWFIFO4 bits[7:0]

The first byte to be written to the Command FIFO.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Accessing BWFIFO1-BWFIFO4

BWFIFO1-BWFIFO4 can be accessed from the JTAG-AP register space:

Access	**Offset**			
	BWFIFO1	**BWFIFO2**	**BWFIFO3**	**BWFIFO4**
Write	`0x10`	`0x14`	`0x18`	`0x1C`
Number of bytes written	1	2	3	4

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.3 CSW, Control/Status Word Register

The CSW register attributes are:

Purpose

CSW register configures and controls transfers through the JTAG interface.

Usage constraints

Several fields in the register are read-only, see *Field descriptions*.

CSW is accessible as follows:

Default
RW

Configurations

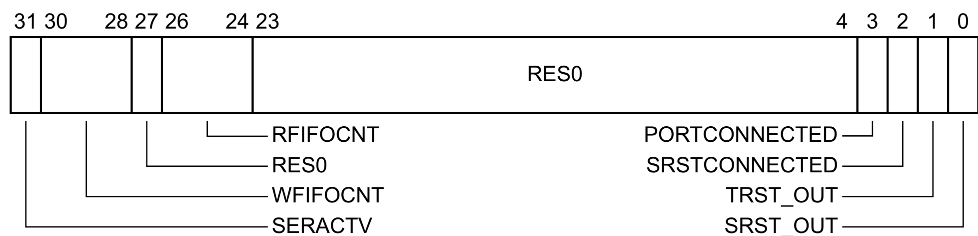
Included in all implementations.

Attributes

A32-bit read/write register.

Field descriptions

The CSW bit assignments are:



SERACTV, bit[31]

JTAG Engine active.

This read-only field can have one of the following values:

`0b0` JTAG Engine is inactive, provided WFIFOCNT is also `0b0`.

`0b1` JTAG Engine is processing commands from the Command FIFO.

Note

The JTAG Engine is only guaranteed to be inactive if both SERACTV and WFIFOCNT are zero.

The reset value of this field is `0b0`.

WFIFOCNT, bits[30:28]

Command FIFO outstanding byte count.

This read-only field returns the number of command bytes held in the Command FIFO that have yet to be processed by the JTAG Engine. The reset value is `0b000`.

Bit[27]

Reserved, RES0.

RFIFOCNT, bits[26:24]

Response FIFO outstanding byte count.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This read-only field returns the number of bytes of response data available in the Response FIFO.

The reset value of this field is `0b000`.

Bits[23:4]

Reserved, RES0.

PORTCONNECTED, bit[3]

Selected ports connected.

This read-only field returns the logical AND of the **PORTCONNECTED** signals from all ports that are currently selected.

This field is read-only. The reset value depends on the state of the connected signals when the register is read.

SRSTCONNECTED, bit[2]

Selected ports reset connected.

This read-only field returns the logical AND of the **SRSTCONNECTED** signals from all ports that are currently selected.

The reset value depends on the state of the connected signals when the register is read.

TRST_OUT, bit[1]

This field drives the **TRST*** signal for the currently selected port or ports.

`0b0` Deassert **TRST*** HIGH.

`0b1` Assert **TRST*** LOW.

Note

The **TRST*** signal is active LOW: when TRST_OUT has the value `0b1`, the **TRST*** output is LOW.

TRST_OUT does not self-reset: it must be cleared to `0b0` by a software write. The reset value is `0b0`.

Although TRST_OUT drives the **TRST*** signal, writing to this field only causes the field value to change. It might be necessary to clock the devices connected to the selected JTAG ports using **TCK**, to enable the devices to recognize the change on **TRST***:

1. Write `0b1` to the CSW.TRST_OUT bit, to specify that **TRST*** must be asserted LOW.
2. Drive a sequence of at least five **TMS** = 1 clocks from the JTAG Engine by issuing the command `0b00111111` to the JTAG Engine. This sequence guarantees that the TAP enters the Test-Logic/Reset state, even if it has no **TRST*** connection.
3. Write `0b0` to CSW.TRST_OUT, to make sure that the **TRST*** signal is HIGH on subsequent **TCK** cycles.

If the JTAG connection is not clocked in this way while **TRST*** is asserted LOW, some or all TAPs might not reset.

SRST_OUT, bit[0]

This field drives the **nSRSTOUT** signal for the port or ports that are currently selected, and can have one of the following values:

`0b0` Deassert **nSRSTOUT** HIGH.

`0b1` Assert **nSRSTOUT** LOW.

Note

The **nSRSTOUT** signal is active LOW: when SRST_OUT has the value `0b1`, the **nSRSTOUT** output is LOW.

SRST_OUT does not self-reset: it must be cleared to `0b0` by a software write. The reset value is `0b0`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Accessing CSW

CSW can be accessed from the JTAG-AP register space:

Offset
0x00

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.4 PSEL, Port Select register

The PSEL characteristics are:

Purpose

PSEL selects one or more JTAG ports to be driven by the JTAG Engine.

Usage constraints

PSEL must only be written to when the JTAG Engine is inactive and the WFIFO is empty. Writing to PSEL at any other time has UNPREDICTABLE results, so before writing to PSEL, you must read the JTAG-AP CSW and make sure that the SERACTV and WFIFOCNT fields are both zero.

The reset value of PSEL is UNKNOWN.

PSEL is accessible as follows:

Default
RW

Configurations

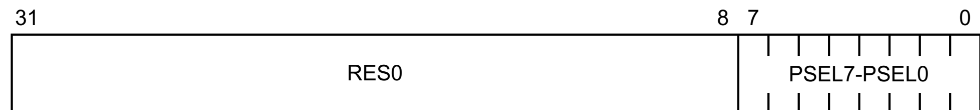
Included in all implementations.

Attributes

A 32-bit read/write register.

Field descriptions

The PSEL bit assignments are:



Bits[31:8]

Reserved, RES0.

PSEL7-PSEL0, bits[7:0]

Select control for the JTAG ports.

The possible values of each of the PSEL*n* fields are:

`0b0` JTAG port *n* is not selected.

`0b1` JTAG port *n* is selected.

If JTAG port *n* is not connected to the JTAG-AP, it is IMPLEMENTATION DEFINED whether PSEL*n* is read/write or RES0.

Note

JTAG port *n* is enabled only if all the following are true:

- The port is connected to the JTAG-AP.
- PSEL*n* is `0b1`.
- The **PORTENABLED** signal from the port to the JTAG-AP is asserted HIGH.

When more than one JTAG port is selected in PSEL:

- The same values for **TDI**, **TMS**, **TRST***, and **nSRSTOUT** are driven to all selected ports.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- The return values from **TDO** are UNKNOWN.

Using the normal, serially connected model for JTAG, IR updates are always made in parallel, which enables updating multiple TAPs in parallel. This update mechanism can be useful, for example to provide synchronized behavior.

Because each JTAG port can contain multiple TAPs connected in series, the process for updating TAPs in parallel is as follows:

1. Scan each JTAG port in turn, by selecting each port in turn in the PSEL register. When scanning a port, leave the required TAP in the TAP Exit1 or Exit2 state.
2. When all ports have been scanned in this way, write to PSEL again to select all the required ports.
3. Scan through the TAP Update state. All the TAPs are updated synchronously.

Accessing PSEL

PSEL can be accessed from the JTAG-AP register space:

Offset
`0x04`

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

C3.5.5 PSTA, Port Status Register

The PSTA register characteristics are:

Purpose

PSTA indicates whether a connected and selected JTAG port has been disabled, even if it has been re-enabled.

Usage constraints

Writing a value with any non-zero bits to PSTA when the JTAG-AP engine is not idle is UNPREDICTABLE. The JTAG-AP Engine is idle when both CSW.SERACTV and CSW.WFIFOCNT are zero.

The reset value of PSTA0-PSTA7 is `0b0`.

The register is accessible as follows:

Default
RW

Configurations

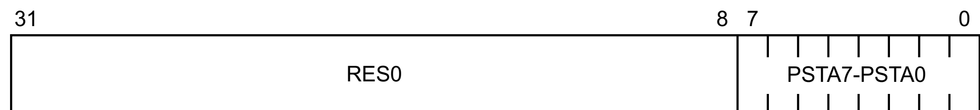
Included in all implementations.

Attributes

A 32-bit read/write register.

Field descriptions

The PSTA bit assignments are:



Bits[31:8]

RES0.

PSTA7-PSTA0, bits[7:0]

Each field PSTAn represents a sticky status flag for JTAG port *n*, and behaves as R/W1C.

PSTAn is set to `0b1` if all the following are true:

- JTAG port *n* is connected to the JTAG-AP.
- PSEL.PSELn is `0b1`.
- JTAG port *n* is disabled.

Once set to `0b1`, PSTAn remains set until it is written with the value `0b1`.

As long as PSTAn is `0b1`, JTAG port *n* remains disabled.

If JTAG port *n* is not connected to the JTAG-AP, PSTAn is RES0.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-7 shows the behavior of PSTAn on reads and writes:

Table C3-7 Read and write behavior of PSTA*n*

Value	Meaning on reads	Action on writes
`0b0`	Port has not been disabled, or port is not connected.	No action, write is ignored.
`0b1`	Port has been disabled.	Clear PSTAn to `0b0`.

Accessing PSTA

PSTA can be accessed from the JTAG-AP register space:

Offset
`0x08`

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Chapter C4
COM-AP programmers' model

This chapter describes the COM-AP. It contains the following sections:

- *About the COM-AP* on page C4-220.
- *COM-AP register map* on page C4-221.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1 About the COM-AP

I_{ZBNM} The COM Port functionality can be included in a Debug Interface Access Port.

I_{MNZC} This specification defines a new type of Access Port, which is known as a COM-AP, for use in systems using ADIv5. For more information, see *AP requirements* on page C1-142.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

4.2 COM-AP register map

R_VBNM A COM-AP implements the register map that is shown in Table C4-1.

Table C4-1 COM-AP register map

Offset	Register	Description
`0x00-0x7C`	-	COM Port programmers' model. See the *Advanced Communication Channel Architecture Specification* for more information.
`0x80-0xF8`	-	Reserved, RES0.
`0xFC`	IDR	Identification Register. See the *Advanced Communication Channel Architecture Specification* for more information.

4.2.1 DP abort

The following rules describe the DP abort:

R_MNZX The COM-AP optionally implements the DP abort mechanism.

R_POIY When an abort request occurs, the abort is ignored if there is no ongoing input transaction to the COM-AP.

R_SDFO When an abort request occurs, if there is an ongoing input transaction to the COM-AP:

- The input transaction must complete in finite time.
- If the input transaction did not complete normally, SR.TRINPROG is set to `0b1` until the input transaction completes normally.

I_ZOPQ Arm recommends that if the input transaction did not complete normally, the COM-AP returns an error to the requester of the input transaction.

I_GGSS After an abort request, the COM-AP is an UNKNOWN state, and is IMPLEMENTATION DEFINED which COM-AP registers are accessible. Arm recommends that:

- Reads of all registers operate as normal.
- Writes to DR and DBR while SR.TRINPROG is `0b1` return an error, otherwise they operate as normal.

I_BLKA Normally, only writes to the DBR have the possibility of stalling input transactions for a variable amount of time until there is space in the TxEngine FIFO. This means that DP aborts normally only affect DBR writes.

4.2.2 IDR, Identification Register

I_ZLKZ For a full description of the IDR, see the *IDR, Identification Register* on page C1-144.

R_ZPBD IDR.Class is `b0001` for a COM-AP.

R_WBRT For a COM-AP designed by Arm, the IDR fields are assigned as follows:

- IDR.TYPE takes the value `0x0`.
- IDR.VARIANT takes an IMPLEMENTATION DEFINED value.
- IDR.REVISION takes an IMPLEMENTATION DEFINED value.
- The JEP106 value in the IDR.DESIGNER takes the value `0x23B`, which is Arm's JEP106 value.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Part D

ROM Tables

Chapter D1
About ROM Tables

The chapter describes ROM Tables. It includes the following sections:

- *ROM Tables Overview* on page D1-226.
- *ROM Table Types* on page D1-227.
- *Component and Peripheral ID Registers for ROM Tables* on page D1-228.
- *Location of the ROM Table* on page D1-229.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.1 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 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 © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

D1.2 ROM Table Types

The following types of ROM Tables in the *Arm® Debug Interface Architecture Specification (ADIv6.0)* are permitted to be used with ADIv5:

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 *Arm® Debug Interface Architecture Specification (ADIv6.0)*.

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.
- The PIDR4.SIZE field must be 0.
- A ROM Table must occupy a single 4KB block of memory.
- For ADIv5 implementations, the DEVID.FORMAT field must be 0, which indicates that the ROM Table entries are 32 bits wide.

For a detailed description of the Class 0x9 ROM Table entries and registers, see *Arm® Debug Interface Architecture Specification (ADIv6.0)*.

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 with an ADIv5-compliant interface.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

D1.3 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 registers in *Arm® CoreSight™ Architecture Specification* describes these registers. This section only describes particular features of the registers when they relate to a ROM Table.

D1.3.1 Identifying the debug SoC, system, or subsystem

The Unique Component Identifier in a ROM table uniquely identifies the SoC, platform, or subsystem described by the ROM table. For example:

- A cluster of components grouped together with a ROM table hierarchy pointing to all the components is uniquely identified by the outermost ROM Table in the cluster.
- A subsystem of all components connected to a single Memory Access Port is uniquely identified by the outermost ROM Table in the subsystem. This ROM Table is usually the first component pointed to by the Memory Access Port.
- An SoC, consisting of multiple Memory Access Ports implementing the Arm Debug Interface version 5, is uniquely identified by the collective Unique Component Identifiers from all of the outermost ROM Tables pointed to by each of the Memory Access Ports.

An SoC, system, or subsystem might be configurable when being built. For example, a cluster of processors might permit the number of processors to be configurable. The ROM Table, which describes such a collection of components, might have the same Unique Component Identifier for all configurations of the system. Although, this is only permitted when components are either included or excluded, and is not permitted to be the same when the location of any component in the address map changes or components significantly change in function. In effect, a ROM Table Unique Component Identifier uniquely identifies a superset configuration of the collection of components. ROM Tables with the same Unique Component Identifier might only describe a subset of this superset.

If the ADI implements DPv2, the DP TARGETID register also uniquely identifies the SoC or platform, and Arm deprecates use of the top-level ROM Table Peripheral ID registers as a unique identifier by tools.

Note

- If SWJ-DP is implemented, it is not required that both the JTAG-DP and SW-DP implement the same DP architecture version, and therefore TARGETID. Tools might be using a DP that does not implement DPv2.
- Deprecation of the use of the top-level ROM Table peripheral ID registers by tools does not remove the requirement on implementations to provide a unique identifier in the top-level ROM Table peripheral ID registers. Future releases of this manual might remove this requirement.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

D1.4 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

Each 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`.

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 © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part E
Appendixes

Appendix E1
Standard Memory Access Port Definitions

This appendix provides information on implementing the Memory Access Port (MEM-AP). It contains the following sections:

- *Introduction* on page E1-234.
- *AMBA AXI3 and AXI4* on page E1-235.
- *AMBA AXI4 with ACE-Lite* on page E1-237.
- *AMBA AXI5* on page E1-240.
- *AMBA AHB3* on page E1-243.
- *AMBA AHB5* on page E1-245.
- *AMBA AHB5 with enhanced HPROT control* on page E1-247.
- *AMBA APB2 and APB3* on page E1-249.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.1 Introduction

The MEM-AP programmers' model includes IMPLEMENTATION DEFINED features. This appendix provides reference implementation options for implementers and users of MEM-APs when connecting to standard memory interfaces. In particular, it provides the recommended interpretations of the following fields:

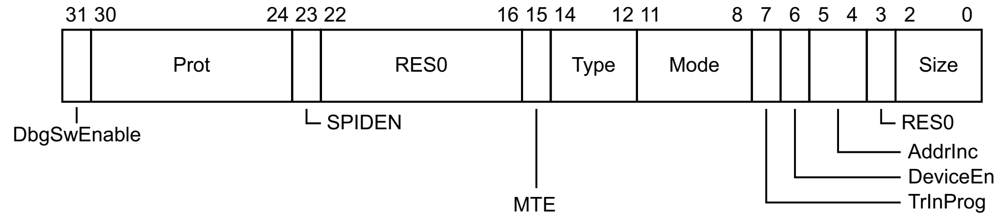
- CSW.Prot.
- CSW.SPIDEN.
- CSW.Type.
- CSW.AddrInc.
- CSW.Size.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.2 AMBA AXI3 and AXI4

This section describes the implementation of the CSW register for AMBA AXI3 and AXI4 implementations. For more information, see *AMBA® AXI™ and ACE™ Protocol Specification AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™*.

E1.2.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178.

Prot, bits[30:24]

For reads, the CSW.Prot field drives the AXI **ARCACHE** and **ARPROT** signals.

For writes, the CSW.Prot field drives the AXI **AWCACHE** and **AWPROT** signals.

The settings for the CSW.Prot field are:

PROT[2:0], bits[30:28]

Drives **AxPROT[2:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-1.

Table E1-1 CSW.Prot mapping to ARPROT or AWPROT

Bit	ARPROT signal	AWPROT signal	Description
30	**ARPROT[2]**	**AWPROT[2]**	Instruction
29	**ARPROT[1]**	**AWPROT[1]**	Non-secure
28	**ARPROT[0]**	**AWPROT[0]**	Privileged

CSW.Prot[29], Non-secure, specifies a non-secure transfer. Its behavior depends on the vale of CSW.SPIDEN. For values in CSW.Prot[29]:

`0b1` Non-secure transfer requested. **ARPROT[1]** or **AWPROT[1]** is HIGH.

`0b0` Secure transfer requested. If CSW.SPIDEN is `0b1`, **ARPROT[1]** or **AWPROT[1]** is LOW. If CSW.SPIDEN is `0b0`, no transfer is initiated, and Arm recommends that an error response is returned to the DP if an access is made to the DRW or Banked Data registers.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CACHE[3:0], bits[27:24]

Drives **AxCACHE[3:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-2.

Table E1-2 CSW.Prot mapping to ARCACHE or AWCACHE

Bit	ARCACHE signal	AWCACHE signal
27	**ARCACHE[3]**	**AWCACHE[3]**
26	**ARCACHE[2]**	**AWCACHE[2]**
25	**ARCACHE[1]**	**AWCACHE[1]**
24	**ARCACHE[0]**	**AWCACHE[0]**

Note

AMBA AXI4 requires asymmetrical usage of **ARCACHE** and **AWCACHE**.

The reset value of CSW.Prot is `0b0110000`.

SPIDEN, bit[23]

The CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**.

Bits[22:16]

RES0.

Type, bits[15:12]

RES0.

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

AddrInc, bits[5:4]

CSW.AddrInc supports the *Increment Packed* mode of transfer. See *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

CSW.Size must support word, half-word, and byte size accesses. It is IMPLEMENTATION DEFINED whether larger access sizes are supported.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

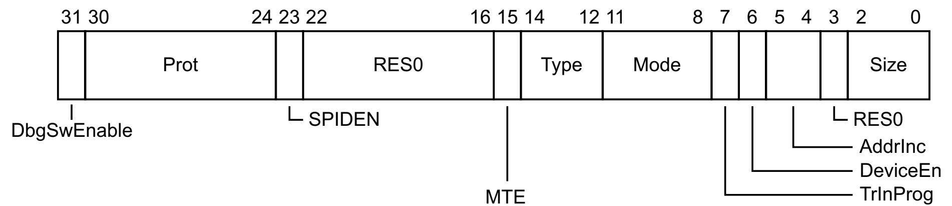
E1.3 AMBA AXI4 with ACE-Lite

This section describes the register implementation for AMBA AXI4 implementations with ACE-Lite For more information, see the *AMBA® AXI™ and ACE™ Protocol Specification AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™*.

The following registers are covered:

- *CSW register implementation*.
- *MBT register implementation* on page E1-239.

E1.3.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178.

Prot, bits[30:24]

For reads, the CSW.Prot field drives the AXI **ARCACHE** and **ARPROT** signals.

For writes, the CSW.Prot field drives the AXI **AWCACHE** and **AWPROT** signals.

The settings for the CSW.Prot field are:

PROT[2:0], bits[30:28]

Drives **AxPROT[2:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-3.

Table E1-3 CSW.Prot mapping to ARPROT or AWPROT

Bit	ARPROT signal	AWPROT signal	Description
30	**ARPROT[2]**	**AWPROT[2]**	Instruction
29	**ARPROT[1]**	**AWPROT[1]**	Non-secure
28	**ARPROT[0]**	**AWPROT[0]**	Privileged

CSW.Prot[29], Non-secure, specifies a non-secure transfer. Its behavior depends on the value of CSW.SPIDEN. For values in CSW.Prot[29]:

`0b1` Non-secure transfer requested. **ARPROT[1]** or **AWPROT[1]** is HIGH.

`0b0` Secure transfer requested. If CSW.SPIDEN is `0b1`, **ARPROT[1]** or **AWPROT[1]** is LOW. If CSW.SPIDEN is `0b0`, no transfer is initiated, and Arm recommends that an error response is returned to the DP if an access is made to the DRW or Banked Data registers.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CACHE[3:0], bits[27:24]

Drives **AxCACHE[3:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-4.

Table E1-4 CSW.Prot mapping to ARCACHE or AWCACHE

Bit	ARCACHE signal	AWCACHE signal
27	**ARCACHE[3]**	**AWCACHE[3]**
26	**ARCACHE[2]**	**AWCACHE[2]**
25	**ARCACHE[1]**	**AWCACHE[1]**
24	**ARCACHE[0]**	**AWCACHE[0]**

Note

AMBA AXI4 requires asymmetrical usage of **ARCACHE** and **AWCACHE**.

The reset value of CSW.Prot is `0b0110000`.

SPIDEN, bit[23]

The CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**.

Bits[22:16]

RES0.

Type, bits[15:12]

The CSW.Type field drives the AXI **AxDOMAIN** signals, where **x** is **R** for reads and **W** for writes.

The settings for the CSW.Type bit field are:

bit[15] Reserved, RES0.

DOMAIN[1:0], bits[14:13]

Possible values are:

`0b00` Non-shareable.

`0b01` Inner shareable.

`0b10` Outer shareable.

`0b11` System.

The reset value of this field is `0b11`.

EnMBT, bit[12]

Enable MBT accesses.

It is IMPLEMENTATION DEFINED whether this field is RW or RAO. If it is RW, the reset value is `0b0`, and must be set to `0b1` before writing to the MBT register.

Mode, bits[11:8]

It is IMPLEMENTATION DEFINED whether this field is RW or RO. If it is RW, the reset value is `0b0000`, and must be set to `0b0001` before writing to the MBT register. If it is RO, then it has the fixed value `0b0001`.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

AddrInc, bits[5:4]

CSW.AddrInc supports the *Increment Packed* mode of transfer. See *Packed transfers* on page C2-158.

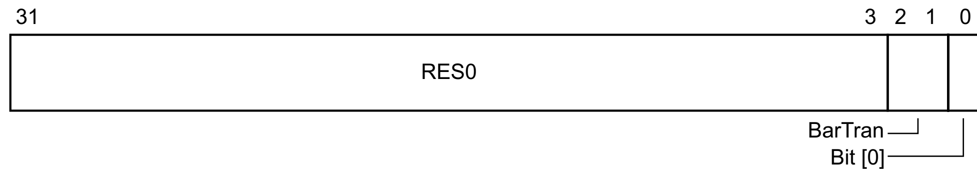
Bit[3]

RES0.

Size, bits[2:0]

CSW.Size must support word, half-word, and byte size accesses. It is IMPLEMENTATION DEFINED whether larger access sizes are supported.

E1.3.2 MBT register implementation



Attributes

MBT register is a read/write register.

Bits[31:3]

Reserved, RES0.

BarTran, bits[2:1]

Possible values are:

`0b00`	Reserved
`0b01`	Memory barrier
`0b10`	Reserved
`0b11`	Synchronization barrier.

Bit[0]

On reads:

`0b0`	Barrier transaction in progress.
`0b1`	No barrier transaction in progress.

SBO on writes.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.4 AMBA AXI5

This section describes the register implementation for AMBA AXI5 implementations. For more information, see the *AMBA® AXI™ and ACE™ Protocol Specification AXI5*.

The following registers are covered:

- *CSW register implementation* on page E1-237.

Support for the Memory Tagging Extension in an AXI5 MEM-AP is optional. When the Memory Tagging Extension is implemented:

- The MEM-AP implements the Large Data Extension, supporting access sizes of up to at least 64-bits.
- The MEM-AP implements the Memory Tagging Extension, supporting 4-bit tags with a 16-byte memory tagging granule.

When memory tagging is enabled, the data size selected by CSW.SIZE must be one of:

- 64-bits.
- An integer multiple of 128-bits.

Note

This specification permits a MEM-AP with memory tagging to only support data value sizes up to 64 bits. However AXI5 does require a bus width of 128-bit and any AXI-AP design needs to ensure it implements the requirements of both specifications.

System memory read accesses with memory tagging enabled perform AXI *Transfer* tag operations on **ARTAGOP**.

System memory write accesses with memory tagging enabled perform AXI *Update* tag operations on **AWTAGOP**

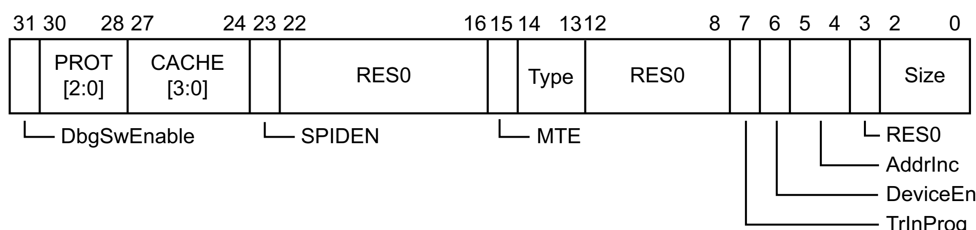
Note

AXI-AP has no need for the *Transfer* or *Match* tag operations on **AWTAGOP**.

When a 64-bit system memory read access is initiated, the AXI-AP performs a read transaction of at least 128-bits, and discards the unused data bytes.

When a 64-bit system memory write access is initiated, the AXI-AP performs a write transaction of at least 128-bits, and drives the write strobes appropriately to ensure only the correct 64-bits of data are transferred.

E1.4.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178.

Prot, bits[30:24]

For reads, the CSW.Prot field drives the AXI **ARCACHE** and **ARPROT** signals.

For writes, the CSW.Prot field drives the AXI **AWCACHE** and **AWPROT** signals.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

The settings for the CSW.Prot field are:

PROT[2:0], bits[30:28]

Drives **AxPROT[2:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-3 on page E1-237.

Table E1-5 CSW.Prot mapping to ARPROT or AWPROT

Bit	ARPROT signal	AWPROT signal	Description
30	**ARPROT[2]**	**AWPROT[2]**	Instruction
29	**ARPROT[1]**	**AWPROT[1]**	Non-secure
28	**ARPROT[0]**	**AWPROT[0]**	Privileged

CSW.Prot[29], Non-secure, specifies a non-secure transfer. Its behavior depends on the value of CSW.SPIDEN. For values in CSW.Prot[29]:

`0b1` Non-secure transfer requested. **ARPROT[1]** or **AWPROT[1]** is HIGH.

`0b0` Secure transfer requested. If CSW.SPIDEN is `0b1`, **ARPROT[1]** or **AWPROT[1]** is LOW. If CSW.SPIDEN is `0b0`, no transfer is initiated, and Arm recommends that an error response is returned to the DP if an access is made to the DRW or Banked Data registers.

CACHE[3:0], bits[27:24]

Drives **AxCACHE[3:0]**, where **x** is **R** for reads and **W** for writes, see Table E1-4 on page E1-238.

Table E1-6 CSW.Prot mapping to ARCACHE or AWCACHE

Bit	ARCACHE signal	AWCACHE signal
27	**ARCACHE[3]**	**AWCACHE[3]**
26	**ARCACHE[2]**	**AWCACHE[2]**
25	**ARCACHE[1]**	**AWCACHE[1]**
24	**ARCACHE[0]**	**AWCACHE[0]**

Note

AMBA AXI4 requires asymmetrical usage of **ARCACHE** and **AWCACHE**.

The reset value of CSW.Prot is `0b0110000`.

SPIDEN, bit[23]

The CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**.

Bits[22:16]

RES0.

MTE, bit[15]

Memory Tagging control. The possible values of this bit are:

`0b0` Memory tagging accesses disabled.

`0b1` Memory tagging accesses enabled.

When memory tagging accesses are enabled, system read and write accesses via DRW, BDx, and DARx, use T0TR for transferring tag information.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

When the Memory Tagging Extension is not implemented, this field is RES0.

Type, bits[14:13]

The CSW.Type field drives the AXI **AxDOMAIN** signals, where **x** is **R** for reads and **W** for writes.

The settings for the CSW.Type bit field are:

DOMAIN[1:0], bits[14:13]

Possible values are:

`0b00` Non-shareable.

`0b01` Inner shareable.

`0b10` Outer shareable.

`0b11` System.

The reset value of this field is `0b11`.

Bits[12:8]

RES0.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

AddrInc, bits[5:4]

CSW.AddrInc supports the *Increment Packed* mode of transfer. See *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

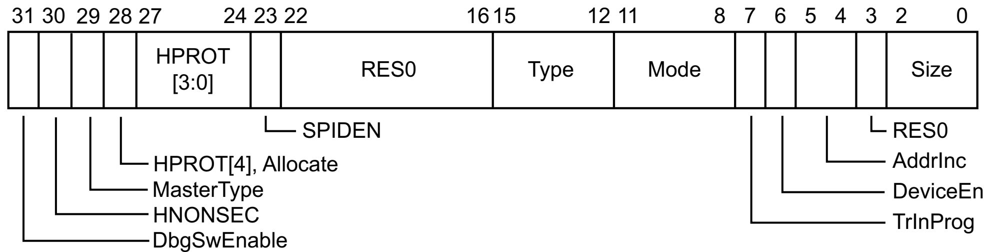
CSW.Size must support word, half-word, and byte size accesses. It is IMPLEMENTATION DEFINED whether larger access sizes are supported.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.5 AMBA AHB3

This section describes the implementation of the CSW register for AMBA AHB implementations. For more information, see the *AMBA® Specification (Rev 2.0)* and the *AMBA® 3 AHB-Lite™ Protocol Specification*.

E1.5.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178.

Prot, bits[30:24]

The CSW.Prot field drives the AHB **HPROT** signals. The settings for the CSW.Prot field are:

HNONSEC, bit[30]

Drives the value of an IMPLEMENTATION DEFINED **HNONSEC** signal.

HNONSEC is not a formally defined AHB3 signal.

If implemented, the reset value of this field is `0b1`.

If not implemented, this field is SBO, and if set to `0b0` the behavior of an AHB-AP transaction is UNPREDICTABLE.

MasterType, bit[29]

Master Type bit. MasterType permits the AHB-AP to mimic a second AHB Requester by driving a different value on **HMASTER[3:0]**. Support for this function is IMPLEMENTATION DEFINED. Valid values for this bit are:

`0b1` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the AHB-AP.

`0b0` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the second bus transaction Requester.

HPROT[4], Allocate, bit[28]

Drives **HPROT[4]**, Allocate. **HPROT[4]** is an Armv5 extension to AHB. For more information, see the *Arm1136JF-S™ and Arm1136J-S™ Technical Reference Manual*.

If the AHB Requester interface does not support the Armv5 extension to AHB, this bit is RAZ/WI.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPROT[3:0], bits[27:24]

Drives **HPROT[3:0]**. See Table E1-7. Support for each **HPROT** signal in the AHB Requester interface is IMPLEMENTATION DEFINED.

Table E1-7 CSW.Prot mapping

Bit	HPROT signal	Description	Description when not implemented at the AHB Requester interface
27	**HPROT[3]**	Cacheable	RAZ/WI
26	**HPROT[2]**	Bufferable	RAZ/WI
25	**HPROT[1]**	Privileged	RAO/WI
24	**HPROT[0]**	Data	RAO/WI

The reset value of CSW.Prot is `0b1000011`.

SPIDEN, bit[23]

It is IMPLEMENTATION DEFINED whether the CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**. Otherwise, the CSW.SPIDEN bit is RAZ.

This bit is always read-only.

Note

AMBA AHB does not support Security Extensions.

Bits[22:16]

RES0.

Type, bits[15:12]

RES0.

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

AddrInc, bits[5:4]

Support for the *Increment Packed* mode of transfer is IMPLEMENTATION DEFINED. See *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

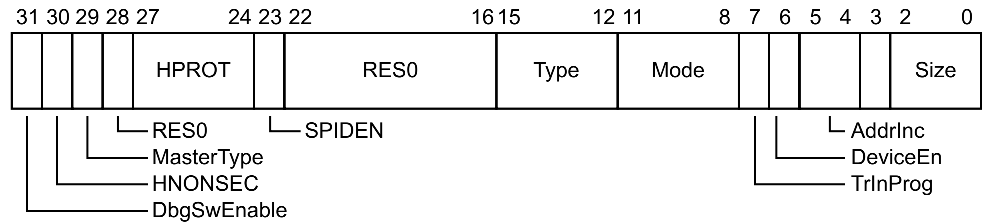
CSW.Size must support word, half-word, and byte size accesses.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.6 AMBA AHB5

This section describes the implementation of the CSW register for AMBA AHB implementations. For more information, see the *AMBA® Specification (Rev 2.0)* and the *Arm® AMBA® 5 AHB Protocol Specification*.

E1.6.1 CSW register implementation



DbgSwEnable, bit[31]

See *CFG, Configuration register* on page C2-175.

HNONSEC, Bit[30]

Drives the value of HNONSEC. It is IMPLEMENTATION DEFINED whether the HNONSEC field is supported.

If implemented, the reset value of this field is `0b1`.

MasterType, bit[29]

Master Type field. MasterType permits the AHB-AP to mimic a second AHB Requester by driving a different value on **HMASTER[3:0]**. Support for this function is IMPLEMENTATION DEFINED. Valid values for this field are:

`0b1` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the AHB-AP.

`0b0` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the second bus transaction Requester.

If this function is not implemented, the field is RES0.

Bit[28]

RES0.

HPROT, bits[27:24]

Drives the value of **HPROT[6:0]**:

- Support for each **HPROT** signal is IMPLEMENTATION DEFINED.
- **HPROT[5]** is always driven with the value 0.
- Bit[27] drives **HPROT[6]**, **HPROT[4]**, and **HPROT[3]**.
- Bit[26] drives **HPROT[2]**.
- Bit[25] drives **HPROT[1]**.
- Bit[24] drives **HPROT[0]**.

SPIDEN, bit[23]

It is IMPLEMENTATION DEFINED whether CSW.SPIDEN reflects the state of the CoreSight authentication interface. If Secure debug is not supported, CSW.SPIDEN is RES0.

This field is always read-only.

Bits[22:16]

RES0.

Type, bits[15:12]

RES0.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

Transfer in progress. This field has one of the following values:

`0b0` The connection to the memory system is idle.

`0b1` A transfer is in progress on the connection to the memory system.

After an ABORT operation, debug software can read this bit to check whether the aborted transaction completed.

DeviceEn, bit[6]

Device enabled.

This field has one of the following values:

`0b0` The MEM-AP is not enabled.

`0b1` Transactions can be issued through the MEM-AP.

This bit corresponds to the value of the **DEVICEEN** signal, which is a control input to the DAP. If **DEVICEEN** is not implemented this bit is RAO. See also *Enabling access to the connected debug device or memory system* on page C2-152.

This field is read-only.

AddrInc, bits[5:4]

Support for the Increment Packed mode of transfer is IMPLEMENTATION DEFINED. See *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

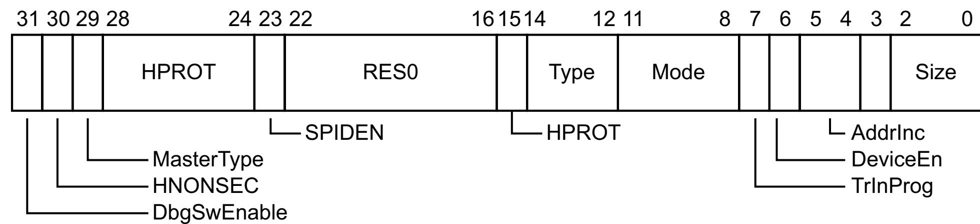
CSW.Size must support word, half-word, and byte size accesses.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.7 AMBA AHB5 with enhanced HPROT control

This section describes the implementation of the CSW register for AMBA AHB with enhanced HPROT control implementations. For more information, see the *AMBA® Specification (Rev 2.0)* and the *Arm® AMBA® 5 AHB Protocol Specification*.

E1.7.1 CSW register implementation



DbgSwEnable, bit[31]

See *CFG, Configuration register* on page C2-175.

HNONSEC, Bit[30]

Drives the value of **HNONSEC**. It is IMPLEMENTATION DEFINED whether the HNONSEC field is supported.

If implemented, the reset value of this field is `0b1`.

MasterType, bit[29]

Requester Type field. MasterType permits the AHB-AP to mimic a second AHB Requester by driving a different value on **HMASTER[3:0]**. Support for this function is IMPLEMENTATION DEFINED. Valid values for this field are:

`0b1` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the AHB-AP.

`0b0` Drive **HMASTER[3:0]** with the bus transaction Requester ID for the second bus transaction Requester.

If this function is not implemented, the field is RES0.

HPROT, bits[28:24, 15]

Drives the value of **HPROT[6:0]**:

- Support for each **HPROT** signal is IMPLEMENTATION DEFINED.
- Bit[15] drives **HPROT[6]**.
- **HPROT[5]** is always driven with the value 0.
- Bits[28:24] drive **HPROT[4:0]**.

SPIDEN, bit[23]

It is IMPLEMENTATION DEFINED whether CSW.SPIDEN reflects the state of the CoreSight authentication interface. If Secure debug is not supported, CSW.SPIDEN is RES0.

This field is always read-only.

Bits[22:16]

RES0.

HPROT, bit[15]

Used to control **HPROT**, see the HPROT field.

Type, bits[14:12]

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RES0.

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

Transfer in progress. This field has one of the following values:

`0b0` The connection to the memory system is idle.

`0b1` A transfer is in progress on the connection to the memory system.

After an ABORT operation, debug software can read this bit to check whether the aborted transaction completed.

DeviceEn, bit[6]

Device enabled.

This field has one of the following values:

`0b0` The MEM-AP is not enabled.

`0b1` Transactions can be issued through the MEM-AP.

This bit corresponds to the value of the **DEVICEEN** signal, which is a control input to the DAP. If **DEVICEEN** is not implemented this bit is RAO. See also *Enabling access to the connected debug device or memory system* on page C2-152.

This field is read-only.

AddrInc, bits[5:4]

Support for the Increment Packed mode of transfer is IMPLEMENTATION DEFINED. See *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

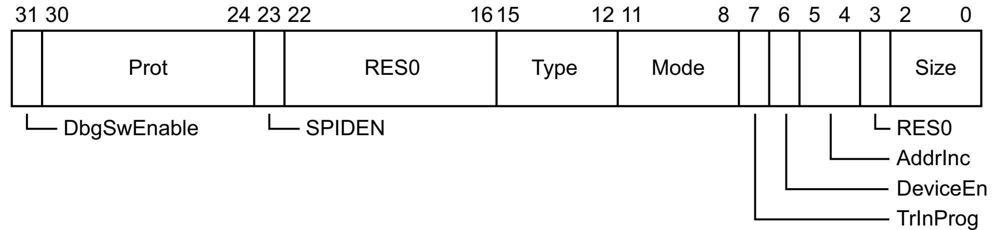
CSW.Size must support word, half-word, and byte size accesses.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E1.8 AMBA APB2 and APB3

This section describes the implementation of the CSW register for AMBA APB2 and APB3 implementations. For more information see the *AMBA® Specification (Rev 2.0)*, and the *AMBA® APB Protocol Specification Version: 2.0*.

E1.8.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178

Prot, bits[30:24]

RES0.

SPIDEN, bit[23]

RES0.

Bits[22:16]

RES0.

Type, bits[15:12]

RES0.

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

AddrInc, bits[5:4]

CSW.AddrInc does not support the *Increment Packed* mode of transfer, and reads as `0b00`. See also *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

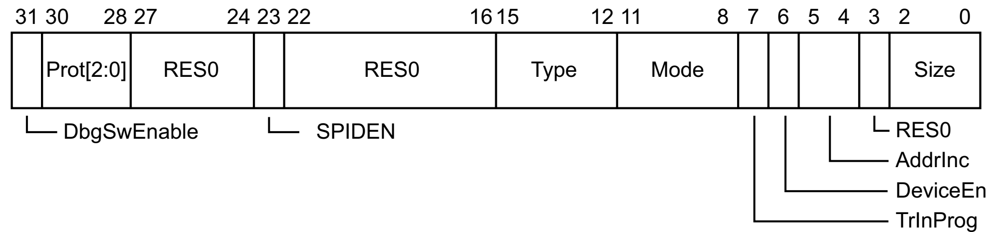
CSW.Size only supports word accesses, and reads as `0b010`. Writes to CSW.Size are ignored.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.9 AMBA APB4 and APB5

This section describes the implementation of the CSW register for AMBA APB4 and APB5 implementations. For more information see the *AMBA® APB Protocol Specification*.

E1.9.1 CSW register implementation



DbgSwEnable, bit[31]

See *CSW, Control/Status Word register* on page C2-178.

Prot[2:0], bits[30:28]

Drives **PPROT[2:0]**.

Bit[29], Non-secure, corresponds to Prot[1], and specifies a non-secure transfer. This bit can have one of the following values:

`0b1` Non-secure transfer requested. **PPROT[1]** is HIGH.

`0b0` Secure transfer requested. The resulting behavior depends on the value of the CSW.SPIDEN field:

- If CSW.SPIDEN is `0b1`, **PPROT[1]** is LOW.
- If CSW.SPIDEN is `0b0`, no transfer is initiated, and Arm recommends that, if an access is made to the DRW or BD0-BD3 registers, an error response is returned.

Bits[27:24]

RES0.

SPIDEN, bit[23]

It is IMPLEMENTATION DEFINED whether CSW.SPIDEN reflects the state of the CoreSight authentication interface. If Secure debug is not supported, CSW.SPIDEN is RES0.

This field is always read-only.

Bits[22:16]

RES0.

Type, bits[15:12]

RES0.

Mode, bits[11:8]

RES0.

TrInProg, bit[7]

See *CSW, Control/Status Word register* on page C2-178.

DeviceEn, bit[6]

See *CSW, Control/Status Word register* on page C2-178.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

AddrInc, bits[5:4]

CSW.AddrInc does not support the *Increment Packed* mode of transfer, and reads as `0b00`. See also *Packed transfers* on page C2-158.

Bit[3]

RES0.

Size, bits[2:0]

CSW.Size only supports word accesses, and reads as `0b010`. Writes to CSW.Size are ignored.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Appendix E2
Cross-over with the Arm Architecture

This appendix describes the required or recommended options for the Arm Debug Interface for the Armv6-M and all Armv7 and Armv8 architecture profiles. It contains the following sections:

- *Introduction* on page E2-254.
- *Armv6-M, Armv7-M, and Armv8-M architecture profiles* on page E2-255.
- *PEs with a physical address of up to 32 bits* on page E2-256.
- *PEs with a physical address greater than 32 bits* on page E2-257.
- *Summary of the requirements for ADIv5 implementations* on page E2-258.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.1 Introduction

ADIv5 is the recommended external debug interface for Armv6-M, all Armv7, all Armv8,and Arm v9 architecture profiles.

When designing with Arm Cortex™ processor cores and Arm CoreSight technology, the choice of Debug Access Port (DAP) features might be at the discretion of the system designer. Arm recommends that system designers choose a DAP that implements all the recommended features for each Arm architecture processing element (PE) that is contained in the design.

ADIv5 might also be used with other architecture variants. For example, an ADIv5 JTAG Access Port (JTAG-AP) might access a Debug Test Access Port (DBGTAP), as defined by Arm Debug Interface v4 (ADIv4) for Armv6 architecture processors.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E2.2 Armv6-M, Armv7-M, and Armv8-M architecture profiles

Arm recommends that a DP that implements ADIv5 is used to access the debug features of the Armv6-M, Armv7-M, or Armv8-M architecture.

Arm recommends that the DP implements the SWD interface, either through an SW-DP or SWJ-DP. A JTAG-DP is permitted.

When accessing debug features of the Armv6-M architecture, or the Armv8-M architecture without the Main Extension, Arm recommends that the DP implements the MINDP model. See *MINDP, Minimal DP extension* on page B1-40.

There must be one MEM-AP for each PE, which complies with the following rules:

- The MEM-AP must be able to address the complete memory space visible to the PE, including all debug peripherals and the NVIC.
- The MEM-AP must support byte, half-word, and word size accesses to memory.
- A MEM-AP that is used to access debug features of the Armv6-M architecture, or the Armv8-M architecture without the Main Extension, is not required to support the packed increment transfer mode.
- A MEM-AP that is used to access debug features of the Armv7-M architecture, or the Armv8-M architecture with the Main Extension, is permitted to support the packed increment transfer mode.

Other APs can be connected to the DAP.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.3 PEs with a physical address of up to 32 bits

The Armv7-A without the Large Physical Address Extension, Armv7-R, and Armv8-R AArch32 architecture profiles do not require a DAP that is compliant with ADIv5. The Arm development tools, however, do require a DAP that is compliant with ADIv5.

Where a DAP that is compliant with ADIv5 is implemented, Arm recommends that the DP implements the JTAG and SWD interfaces through an SWJ-DP.

Many PEs can be connected to a single MEM-AP. The MEM-AP must only be able to address the debug peripherals of the connected PEs. If the MEM-AP can only address the debug peripherals, it is only required to support word size accesses to memory, and therefore is not required to support the packed increment transfer mode.

Arm recommends that debug implementations include a MEM-AP that can address the complete memory space visible to the PE or PEs. This MEM-AP might be a second MEM-AP in the DAP. Arm recommends that a MEM-AP that can access the complete memory space supports byte, half-word, and word size accesses to memory. This MEM-AP is permitted to support the packed increment transfer mode.

Other APs can also be connected to the DAP.

Note

Do not confuse the Armv7-A or Armv8-A Large Physical Address Extension with the ADI MEM-AP Large Physical Address Extension.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E2.4 PEs with a physical address greater than 32 bits

The requirements for Armv7-A with Large Physical Address Extension, and Armv8-A, Armv8-R AArch64, and Armv9-A architecture profiles are the same as for those described in *PEs with a physical address of up to 32 bits* on page E2-256, with the following additions for any MEM-AP with system access:

- MEM-AP Large Physical Address Extension, up to at least the size that is supported by the PE.
- For Armv7-A systems with the Large Physical Address Extension, Arm recommends that the MEM-AP implements the Large Data Extension providing at least doubleword accesses, to allow for atomic update of page table entries.
- For Armv8-A, Armv8-R AArch64, and Armv9-A systems, the MEM-AP must implement the Large Data Extension providing at least doubleword accesses.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E2.5 Summary of the requirements for ADIv5 implementations

Table E2-1 summarizes the required and recommended components of an ADI implementation for each of the Arm architecture variants for which ADIv5 is the required or recommended DAP.

Table E2-1 Recommended ADI implementations for Arm Architecture variants

Component		Armv6-M and Armv8-M without Main Extension	Armv7-M and Armv8-M with Main Extension	PEs with a physical address up to 32 bits	PEs with a physical address greater than 32 bits
DAP	ADIv5	Required	Required	Recommended	Recommended
DP	JTAG-DP	Permitted	Permitted	Permitted	Permitted
	SW-DP	-	-	Permitted	Permitted
	SWJ-DP	-	-	Recommended	Recommended
	SWJ-DP or SW-DP	Recommended	Recommended	-	-
MEM-AP	One per PE	Required	Required	Permitted	Permitted
	Access to system memory	Required	Required	Permitted	Permitted
	Support for 8-bit and 16-bit accesses	Required	Required	Required only if system access is supported	Required only if system access is supported
	Support for 32-bit accesses	Required	Required	Required	Required
	Support for 64-bit accesses	Permitted	Permitted	Permitted	Required
	Support for large physical addresses	Not permitted	Not permitted	Permitted	Required if system access is supported
	Support for packed increment transfers	Permitted	Permitted	Permitted	Permitted
	Support for the Memory Tagging Extension	Not permitted	Not permitted	Permitted	Required if system access is supported and system supports memory tagging.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Appendix E3
Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some *helper* procedures and functions that are used by pseudocode. It contains the following sections:

- *About the Arm pseudocode* on page E3-260.
- *Pseudocode for instruction descriptions* on page E3-261.
- *Data types* on page E3-263.
- *Operators* on page E3-268.
- *Statements and control structures* on page E3-274.
- *Built-in functions* on page E3-279.
- *Miscellaneous helper procedures and functions* on page E3-282.
- *Arm pseudocode definition index* on page E3-284.

Note

This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode. This appendix is not complete. Changes are planned for future releases.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. *Pseudocode for instruction descriptions* on page E3-261 gives general information about this instruction pseudocode, including its limitations.

The following sections describe the Arm pseudocode in detail:

- *Data types* on page E3-263.
- *Operators* on page E3-268.
- *Statements and control structures* on page E3-274.

Built-in functions on page E3-279 and *Miscellaneous helper procedures and functions* on page E3-282 describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. *Arm pseudocode definition index* on page E3-284 contains the indexes to the pseudocode.

E3.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `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.

For more information, see *Special statements* on page E3-278.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in *General limitations of Arm pseudocode* on page E3-260 and *Limitations of the instruction pseudocode* on page E3-262.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. *Instruction encoding diagrams and instruction pseudocode* gives more information about the pseudocode provided for each instruction.

E3.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is CONSTRAINED UNPREDICTABLE. For more information, see *SBZ or SBO fields T32 and A32 in instructions on page K1-11158*.
- A named single bit or a bit in a named multi-bit field. The `cond` field in bits[31:28] of many A32/T32 instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- The encoding diagram is not for an A32/T32 instruction.
- The encoding diagram is for an A32/T32 instruction that does not have a `cond` field in bits[31:28].
- The encoding diagram is for an A32/T32 instruction that has a `cond` field in bits[31:28], and bits[31:28] of the instruction are not `0b1111`.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.

   In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

   If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is CONSTRAINED UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as CONSTRAINED UNPREDICTABLE, see *SBZ or SBO fields T32 and A32 in instructions on page K1-11158*.
5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode, if present, performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

E3.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see *Ordering constraints on page E2-6790*.
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
  - `UNDEFINED` instruction.
  - Hyp trap.
  - Monitor trap.
  - Trap to AArch64 exception.

  In such cases, the `UNDEFINED` pseudocode statement or call to the applicable trap function lies inside the `if ConditionPassed() then …` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:
  - *Conditional execution of undefined instructions on page G1-8582*.
  - *EL2 configurable controls on page G1-8626*.
  - *EL3 configurable controls on page G1-8646*.
  - *Configurable instruction controls on page D1-4555*.
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. *Combinations of floating-point exceptions on page E1-6767* describes the exact rules.

  Note

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in *Handling exceptions that are taken to an Exception level using AArch32 on page G1-8545*.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.3 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page E3-264.
- *Reals* on page E3-264.
- *Booleans* on page E3-264.
- *Enumerations* on page E3-265.
- *Structures* on page E3-265.
- *Tuples* on page E3-266.
- *Arrays* on page E3-267.

E3.3.1 General data type rules

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

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

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.

E3.3.2 Bitstrings

This section describes the bitstring data type.

Syntax

`bits(N)` The type name of a bitstring of length `N`.

`bit` A synonym of `bits(1)`.

Description

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 0.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Bitstring constants literals 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 literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right *N*-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit *N*-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

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 that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

E3.3.3 Integers

This section describes the data type for integer numbers.

Syntax

`integer` The type name for the integer data type.

Description

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, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal integer literals 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`.

E3.3.4 Reals

This section describes the data type for real numbers.

Syntax

`real` The type name for the real data type.

Description

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, and the pseudocode provides functions to interpret those bitstrings as reals.

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

E3.3.5 Booleans

This section describes the Boolean data type.

Syntax

`boolean` The type name for the Boolean data type.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

`TRUE` The two values a Boolean variable can take.

Description

A Boolean is a logical `TRUE` or `FALSE` value.

Note

This is not the same type as `bit`, which is a bitstring of length 1. A Boolean can only take on one of two values: `TRUE` or `FALSE`.

E3.3.6 Enumerations

This section describes the enumeration data type.

Syntax and examples

`enumeration` Keyword to defined a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

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

E3.3.7 Structures

This section describes the structure data type.

Syntax and examples

`type` The keyword used to declare the structure data type.

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

An example definition for a new structure called `ShiftSpec` that contains an bitstring member called `shift` and a integer member named `amount`. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named `abc` of type `ShiftSpec`.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

In the syntax section, the example defines a structure called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer named `amount`. After declaring a variable of that type named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

E3.3.8 Tuples

This section describes the tuple data type.

Examples

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

An example of the tuple syntax.

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

An example of assigning values to a tuple.

Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)` and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.3.9 Arrays

This section describes the array data type.

Syntax

`array` The type name for the array data type.

`array data_type array_name[A..B];`

Declaration of an array of type `data_type`, which might be compound data type. It is named `array_name` and is indexed with an integer range from `A` to `B`.

Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

```
array bits(64) _R[0..30];
```

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

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

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

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. See *Function and procedure calls* on page E3-274.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.4 Operators

This section describes:

- *Relational operators*.
- *Boolean operators*.
- *Bitstring operators* on page E3-269.
- *Arithmetic operators* on page E3-269.
- *The assignment operator* on page E3-270.
- *Precedence rules* on page E3-272.
- *Conditional expressions* on page E3-272.
- *Operator polymorphism* on page E3-272.

E3.4.1 Relational operators

The following operations yield results of type `boolean`.

Equality and non-equality

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

Both `x` and `y` must be of type `bits(N)`, `real`, `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `'1000'`, `'1100'`, `'1001'`, and `'1101'`. This is known as a bitmask.

Note

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

Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x >= y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions separated by commas.

E3.4.2 Boolean operators

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

If `x` and `y` are Boolean expressions, 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`.

Note

This is known as short circuit evaluation.

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`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

If x and y are booleans or Boolean expressions, then the result of x != y is the same as the result of exclusive-ORing x and y together. The operator EOR only accepts bitstring arguments.

E3.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

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, logically ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring concatenation and slicing

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.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is x<integer_list>, where x is the integer or bitstring being sliced, and <integer_list> is a comma-separated list of integers enclosed in angle brackets. 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, and y is 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.

The notation for a range expression is 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> represents instr<31, 30, 29, 28>.

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 APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31> as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, APSR.<N, C, Q> is synonymous with APSR <31, 29, 27>.

E3.4.4 Arithmetic operators

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Unary plus and minus

If x is an integer or real, then +x is x unchanged, -x is x with its sign reversed. Both 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.

There are two cases where the types of x and y can be different. A bitstring and an integer can be added together to allow the operation PC + 4. An integer can be subtracted from a bitstring to allow the operation PC - 2.

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 x and y 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.

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

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

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

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

Scaling

If x and n are of type integer, then:

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

Raising to a power

If x is an integer or a real and n is an integer, then x^n is the result of raising x to the power of n, and:

- If x is of type integer, then x^n is of type integer.
- If x is of type real, then x^n is of type real.

E3.4.5 The assignment operator

The assignment operator is the = character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

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

This following subsection defines valid expression syntax.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

General expression syntax

An expression is one of the following:

- A literal.
- 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 defined in an Arm architecture specification defines a correspondingly named pseudocode 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:

- 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,
- Be promoted as providing any useful information to software.

Note

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 on 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.

Note

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as `-` to indicate that the corresponding item of the assigned tuple value is discarded. For example:

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

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- 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.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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

E3.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see *Boolean operators* on page E3-268.
2. Operators 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.

E3.4.7 Conditional expressions

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`.

E3.4.8 Operator polymorphism

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

Table E3-1 on page E3-272 summarizes the operand types valid for each unary operator and the result type. Table E3-2 on page E3-272 summarizes the operand types valid for each binary operator and the result type.

Table E3-1 Result and operand types permitted for unary operators

Operator	Operand Type	Result Type
`-`	`integer`	`integer`
	`real`	`real`
`NOT`	`bits(N)`	`bits(N)`
`!`	`boolean`	`boolean`

Table E3-2 Result and operand types permitted for binary operators

Operator	First operand type	Second operand type	Result type
`==`	`bits(N)`	`integer`	`boolean`
		`bits(N)`	
	`integer`	`integer`	
	`real`	`real`	
	`enumeration`	`enumeration`	
	`boolean`	`boolean`	
`!=`	`bits(N)`	`bits(N)`	`boolean`
	`integer`	`integer`	
	`real`	`real`	

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table E3-2 Result and operand types permitted for binary operators (continued)

Operator	First operand type	Second operand type	Result type
<, > <= , >=	integer	integer	boolean
	real	real	
+, -	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
		integer	
<<, >>	integer	integer	integer
*	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
/	real	real	real
DIV	integer	integer	integer
MOD	integer	integer	integer
	bits(N)	integer	
&&, ||	boolean	boolean	boolean
AND, OR, EOR	bits(N)	bits(N)	bits(N)
^	integer	integer	integer
	real	integer	real

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

E3.5 Statements and control structures

This section describes the statements and program structures available in the pseudocode:

- *Statements and Indentation*.
- *Function and procedure calls*.
- *Conditional control structures* on page E3-276.
- *Loop control structures* on page E3-277.
- *Special statements* on page E3-278.
- *Comments* on page E3-278.

E3.5.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

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 structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

E3.5.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Procedure and function definitions

A procedure definition has 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 definition line is not terminated by a semicolon. This distinguishes 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>;
```

Note

A function or procedure name can include a ".". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

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

Its related function definition with no return type is equivalent to writing to an array. For example:

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

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

Procedure calls

A procedure call has the form:

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

Return statements

A procedure return has the form:

```
return;
```

A function return has the form:

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
return <expression>;
```

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

E3.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

if … then … else …

In addition to being a ternary operator, a multi-line `if … then … else …` structure can act as a control structure and has 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 `elsif` 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 the `then` part, and in the `else` part if it is present, contain only simple statements such as:

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

Note

In these forms, `<statement 1>`, `<statement 2>`, and `<statement A>` must be terminated by semicolons. This, and the fact that the `else` part is optional, distinguish its use as a control structure from its use as a ternary operator.

---

case … of …

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

```
case <expression> of
    when <literal values1>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    when <literal values2>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    … more "when" groups if required …

    otherwise
```


Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

```
        <statement A>;
        <statement B>;
        …
        <statement Z>;
```

In this structure, `<literal values1>` and `<literal values2>` consist of literal values of the same type as `<expression>`, separated by commas. There can be additional `when` groups in the structure. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, the literal values can also include bitstring literals containing `'x'` bits, known as bitmasks. For details, see *Equality and non-equality* on page E3-268.

E3.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.

repeat … until …

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

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

It executes the statement block at least once, and the loop repeats until `<boolean expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

while … do

A `while … do` structure has the form:

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

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

for …

A `for …` structure has the form:

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

The `<assignable_expression>` is initialized to `<integer_expr1>` and compared to `<integer_expr2>`. If `<integer_expr1>` is less than `<integer_expr2>`, the loop body is executed and the `<assignable_expression>` incremented by one. This repeats until `<assignable expression>` is more than or equal to `<integer_expr2>`.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where `<integer_expr1>` is decremented after the loop body executes and continues until `<assignable expression>` is less than or equal than `<integer_expr2>`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.5.5 Special statements

This section describes statements with particular architecturally defined behaviors.

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

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

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

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

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

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

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

IMPLEMENTATION_DEFINED

This subsection describes the statement:

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

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

E3.5.6 Comments

The pseudocode supports two styles of comments:

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

`/**/` statements might not be nested, and the first `*/` ends the comment.

Note

Comment lines do not require a terminating semicolon.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.6 Built-in functions

This section describes:

- *Bitstring manipulation functions*.
- *Arithmetic functions* on page E3-280.

E3.6.1 Bitstring manipulation functions

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.

Bitstring concatenation and replication

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 count

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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 the bits of `x` that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of the bits of `x` 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()` is the integer whose two's complement representation is `x`.

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

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

E3.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

Absolute value

If `x` is either of type real or integer, `Abs(x)` returns the absolute value of `x`. The result is the same type as `x`.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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`.
  - `RoundUp(x)` if `x < 0.0`.

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

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. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.7 Miscellaneous helper procedures and functions

This section lists the prototypes of miscellaneous *helper* procedures and functions used by the pseudocode, together with a brief description of the effect of the procedure or function. The pseudocode does not define the operation of these helper procedures and functions.

Note

Chapter J1 *Armv8 Pseudocode* also has an entry for each of these functions, but currently these entries do not say anything about the effect of the function. When this information is added in Chapter J1, this section will be removed from the manual.

E3.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

```
EndOfInstruction();
```

E3.7.2 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option);
```

E3.7.3 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address);
```

E3.7.4 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address);
```

E3.7.5 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address);
```

E3.7.6 Hint_Yield()

This procedure performs a *Yield* hint.

```
Hint_Yield();
```

E3.7.7 IsExternalAbort()

This function returns `TRUE` if the abort currently being processed is an External abort and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
```


Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.7.8 IsAsyncAbort()

This function returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

boolean IsAsyncAbort(FaultRecord fault);
```

E3.7.9 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```
bits(11) LSInstructionSyndrome();
```

E3.7.10 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```
integer ProcessorID();
```

E3.7.11 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```
boolean RemapRegsHaveResetValues();
```

E3.7.12 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally defined reset values to those values. For more information about the affected registers, see:

- *Reset behavior on page D1-4564*.
- *PE state on reset into AArch32 state on page G1-8602*.

```
AArch64.ResetControlRegisters(boolean ResetIsCold)
AArch32.ResetControlRegisters(boolean ResetIsCold)
```

E3.7.13 ThisInstr()

This function returns the bitstring encoding of the currently executing instruction.

```
bits(32) ThisInstr();
```

Note

Currently, this function is used only on 32-bit instruction encodings.

E3.7.14 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:

```
integer ThisInstrLength();
```


Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

E3.8 Arm pseudocode definition index

This section contains the following tables:

- Table E3-3 on page E3-284 which contains the pseudocode data types.
- Table E3-4 on page E3-284 which contains the pseudocode operators.
- Table E3-5 on page E3-285 which contains the pseudocode keywords and control structures.
- Table E3-6 on page E3-286 which contains the statements with special behaviors.

Table E3-3 Index of pseudocode data types

Keyword	Meaning
`array`	Type name for the array type
`bit`	Keyword equivalent to `bits(1)`
`bits(N)`	Type name for the bitstring of length `N` data type
`boolean`	Type name for the Boolean data type
`enumeration`	Keyword to define a new enumeration type
`integer`	Type name for the integer data type
`real`	Type name for the real data type
`type`	Keyword to define a new structure

Table E3-4 Index of pseudocode operators

Operator	Meaning
`-`	Unary minus on integers or reals
	Subtraction of integers, reals, and bitstrings
	Used in the left-hand side of an assignment or a tuple to discard the result
`+`	Unary plus on integers or reals
	Addition of integers, reals, and bitstrings
`.`	Extract named member from a list
	Extract named bit or field from a register
`:`	Bitstring concatenation
	Integer range in bitstring extraction operator
`!`	Boolean NOT
`!=`	Comparison for inequality
`(…)`	Around arguments of procedure or function
`[…]`	Around array index
	Around arguments of array-like function
`*`	Multiplication of integers, reals, and bitstrings
`/`	Division of reals

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Table E3-4 Index of pseudocode operators (continued)

Operator	Meaning
`&&`	Boolean AND
`<`	*Less than* comparison of integers and reals
`<…>`	Slicing of specified bits of bitstring or integer
`<<`	Multiply integer by power of 2
`<=`	*Less than or equal* comparison of integers and reals
`=`	Assignment operator
`==`	Comparison for equality
`>`	*Greater than* comparison of integers and reals
`>=`	*Greater than or equal* comparison of integers and reals
`>>`	Divide integer by power of 2
`||`	Boolean OR
`^`	Exponential operator
`AND`	Bitwise AND of bitstrings
`DIV`	Quotient from integer division
`EOR`	Bitwise EOR of bitstrings
`IN`	Tests membership of a certain expression in a set of values
`MOD`	Remainder from integer division
`NOT`	Bitwise inversion of bitstrings
`OR`	Bitwise OR of bitstrings
`case … of …`	Control structure for the
`if … then … else …`	Condition expression selecting between two values

Table E3-5 Index of pseudocode keywords and control structures

Operator	Meaning
`/*…*/`	Comment delimiters
`//`	Introduces comment terminated by end of line
`FALSE`	One of two values a Boolean can take (other than `TRUE`)
`for … = …to …`	Loop control structure, counting up from the initial value to the upper limit
`for … = … downto …`	Loop control structure, counting down from the initial value to the lower limit
`if … then … else …`	Conditional control structure
`otherwise`	Introduces default case in `case … of …` control structure

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table E3-5 Index of pseudocode keywords and control structures (continued)

Operator	Meaning
`repeat … until …`	Loop control structure that runs at least once until the termination condition is satisfied
`return`	Procedure or function return
`TRUE`	One of two values a Boolean can take (other than `FALSE`)
`when`	Introduces specific case in `case … of …` control structure
`while … do …`	Loop control structure that runs until the termination condition is satisfied

Table E3-6 Index of special statements

Keyword	Meaning
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior
`SEE`	Points to other pseudocode to use instead
`UNDEFINED`	Cause Undefined Instruction exception
`UNKNOWN`	Unspecified value
`UNPREDICTABLE`	Unspecified behavior

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Glossary

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

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

ADI
See Arm Debug Interface (ADI).

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

See also AMBA.

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

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

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

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

AXI

An AMBA bus protocol that supports:

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

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

Big-endian

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

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

See also Little-endian and Endianness.

Boundary scan chain

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

Burst

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

Cold reset

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

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

See also Processing Element (PE), Warm reset.

Completer

An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.

See also Requester.

Core reset

See Warm reset.

DAP

See Debug Access Port (DAP).

Data Link layer

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

DATA LINK DEFINED

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

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

DBGTAP

See Debug Test Access Port (DBGTAP).

Debug Access Port (DAP)

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

Debug Test Access Port (DBGTAP)

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

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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

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

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

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

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

See also Little-endian and Big-endian.

ETM See *Embedded Trace Macrocell (ETM)*.

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

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

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

IMP DEF *See* IMPLEMENTATION DEFINED.

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

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

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

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

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

JTAG Access Port (JTAG-AP)
An optional component of the DAP that provides debugger access to on-chip scan chains.

JTAG Debug Port (JTAG-DP)
An optional external interface for the DAP that provides a standard JTAG interface for debug access.

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

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

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

See also Big-endian and Endianness.

PE *See* Processing Element (PE).

Powerup reset *See* Cold reset.

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

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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

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

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

See also Read-As-One (RAO).

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

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

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

See also Read-As-Zero (RAZ).

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

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

Requester An agent in a computing system that is capable of initiating memory transactions.

See also Completer.

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

Note

RES0 is not used in descriptions of instruction encodings.

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

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

This means the definition of RES0 for register fields is:

If a bit is RES0 in all contexts

It is IMPLEMENTATION DEFINED whether:

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

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

   Note

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

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

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

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

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

If a bit is RES0 only in some contexts

When the bit is described as RES0:

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

  Note

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

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

For any RES0 bit, software:

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

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

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

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

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

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

RES1

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

Note

RES1 is not used in descriptions of instruction encodings.

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

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

This means the definition of RES1 for register fields is:

If a bit is RES1 in all contexts

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

It is IMPLEMENTATION DEFINED whether:

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

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

     Note

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

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

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

If a bit is RES1 only in some contexts

When the bit is described as RES1:

- An indirect write to the register sets the bit to `0b1`.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

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

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an unknown value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the core, other than determining the value read back from that bit.

For any RES1 bit, software:

- Must not rely on the bit reading as `0b1`.
- Must use an *SBOP* policy to write to the bit.

The RES1 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as `0b1`, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as *SBO*.

This RES1 description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 1s.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved

Unless otherwise stated in the architecture or product documentation:

- Reserved instruction and 32-bit system control register encodings are unpredictable.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

- Reserved 64-bit system control register encodings are undefined.
- Reserved register bit fields are UNK/SBZP.

SBO *See* Should-Be-One (SBO).

SBOP *See* Should-Be-One-or-Preserved (SBOP).

SBZ *See* Should-Be-Zero (SBZ).

SBZP *See* Should-Be-Zero-or-Preserved (SBZP).

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

Serial Wire debug (SWD)

A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals required for a JTAG connection.

Serial-Wire Debug Port (SW-DP)

The interface for Serial Wire Debug.

Serial Wire JTAG Debug Port (SWJ-DP)

The SWJ-DP is a combined JTAG-DP and SW-DP that you can use to connect either a Serial Wire Debug (SWD) or JTAG probe to a target.

Should-Be-One (SBO)

Hardware must ignore writes to the field.

Software should write the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as `0b1`, or to a field that should be written as all 1s.

Should-Be-One-or-Preserved (SBOP)

The Armv7 Large Physical Address Extension modified the definition of SBOP to apply to register fields that are SBOP in some but not all contexts. From the introduction of Armv8 such register fields are described as RES1, see *RES1*. The definition of SBOP given here applies only to fields that are SBOP in all contexts.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it should preserve the value of the field by writing the value that it previously read from the field. Otherwise, it should write the field as all 1s.

If software writes a value to the field that is not a value previously read for the field and is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as `0b1`, or to a field that should be written as its preserved value or as all 1s.

See also Should-Be-Zero-or-Preserved (SBZP), Should-Be-One (SBO).

Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Software should write the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE result.

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

Should-Be-Zero-or-Preserved (SBZP)

The Armv7 Large Physical Address Extension modified the definition of SBZP to apply to register fields that are SBZP in some but not all contexts. From the introduction of Armv8 such register fields are described as res0, see *RES0*. The definition of SBZP given here applies only to field that are SBZP in all contexts.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value previously read for the field and is not all 0s, it must expect an UNPREDICTABLE result.

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

See also Should-Be-One-or-Preserved (SBOP), Should-Be-Zero (SBZ).

SWD *See* Serial Wire debug (SWD).

SW-DP *See* Serial-Wire Debug Port (SW-DP).

SWJ-DP *See* Serial Wire JTAG Debug Port (SWJ-DP)

TAP *See* Test Access Port (TAP).

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. In the JTAG standard, the **nTRST** signal is optional, but this signal is mandatory in Arm processors because it is used to reset the debug logic.

See also Joint Test Action Group (JTAG), Debug Test Access Port (DBGTAP).

Trace port A port on a device, such as a processor or ASIC, to output trace information.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of the elements of the access.

See also Aligned.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE and do not return UNKNOWN values.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

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

UNP *See* UNPREDICTABLE.

UNPREDICTABLE

For an Arm processor, UNPREDICTABLE means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect. An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In an implementation that supports Virtualization, the Non-secure execution of unpredictable instructions at a lower level of privilege can be trapped to the hypervisor, provided that at least one instruction that is not unpredictable can be trapped to the hypervisor if executed at that lower level of privilege.

For an Arm trace macrocell, UNPREDICTABLE means that the behavior of the macrocell cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is UNPREDICTABLE. UNPREDICTABLE behavior can affect the behavior of the entire system, because the trace macrocell can cause the core to enter Debug state, and external outputs can be used for other purposes.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

Note

In issue A of this document, UNPREDICTABLE also meant an UNKNOWN value.

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

W1C Hardware must implement the bit as follows:

- Writing a `0b1` to the bit clears the bit to `0b0`.
- Writing a `0b0` to the bit has no effect.

Warm reset Also known as a core reset. Initializes most of the processor functionality, excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

See also Cold reset.

WI Hardware must must ignore writes to the field. Software can rely on writes being ignored. This description can apply to a single bit, or to a field.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned A data item having a memory address that is divisible by four.

Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.

 Copyright © 2006, 2009, 2012-2013, 2017-2018, 2020, 2022 Arm Limited or its affiliates. All rights reserved.