

# ARM<sup>®</sup> Debug Interface v5

## Architecture Specification



# ARM Debug Interface v5

## Architecture Specification

Copyright © 2006 ARM Limited. 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.

### Proprietary Notice

ARM and the ARM Powered logo are registered trademarks of ARM Limited.

The ARM logo, AMBA, ARM7, ARM7TDMI, ARM9, ARM10, CoreSight, and TDMI, are trademarks of ARM Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM Limited in good faith.

1. Subject to the provisions set out below, ARM Limited hereby grants to you a perpetual, non-exclusive, nontransferable, royalty free, worldwide licence to use this ARM Architecture Reference Manual for the purposes of developing; (i) software applications or operating systems which are targeted to run on microprocessor cores distributed under licence from ARM Limited; (ii) tools which are designed to develop software programs which are targeted to run on microprocessor cores distributed under licence from ARM Limited; (iii) integrated circuits which incorporate a microprocessor core manufactured under licence from ARM Limited.

2. Except as expressly licensed in Clause 1 you acquire no right, title or interest in the ARM Architecture Reference Manual, or any Intellectual Property therein. In no event shall the licences granted in Clause 1, be construed as granting you expressly or by implication, estoppel or otherwise, licences to any ARM technology other than the ARM Architecture Reference Manual. The licence grant in Clause 1 expressly excludes any rights for you to use or take into use any ARM patents. No right is granted to you under the provisions of Clause 1 to; (i) use the ARM Architecture Reference Manual for the purposes of developing or having developed microprocessor cores or models thereof which are compatible in whole or part with either or both the instructions or programmer's models described in this ARM Architecture Reference Manual; or (ii) develop or have developed models of any microprocessor cores designed by or for ARM Limited; or (iii) distribute in whole or in part this ARM Architecture Reference Manual to third parties without the express written permission of ARM Limited; or (iv) translate or have translated this ARM Architecture Reference Manual into any other languages.

3. THE ARM ARCHITECTURE REFERENCE MANUAL IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.

4. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM tradename, in connection with the use of the ARM Architecture Reference Manual or any products based thereon. Nothing in Clause 1 shall be construed as authority for you to make any representations on behalf of ARM Limited in respect of the ARM Architecture Reference Manual or any products based thereon.

Copyright © 2006 ARM Limited

110 Fulbourn Road Cambridge, England CB1 9NJ

Restricted Rights Legend: Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19

Figure 4-3 on page 4-5 reproduced with permission IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture Copyright 2006, by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

### **Confidentiality Status**

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

### **Web Address**

<http://www.arm.com>



# Contents

## ARM Debug Interface v5 Architecture Specification

### Preface

About this specification .....	xviii
Conventions .....	xx
Further reading .....	xxii
Feedback .....	xxiii

### Chapter 1

#### Introduction

1.1	About the ARM Debug Interface version 5 (ADIV5) .....	1-2
1.2	The function of the ARM Debug Interface .....	1-3
1.3	The subdivisions of an ARM Debug Interface .....	1-5
1.4	Design choices for an ARM Debug Interface v5 implementation .....	1-6

### Chapter 2

#### Overview of the ARM Debug Interface and its components

2.1	ARM Debug Interface overview .....	2-2
2.2	The external interface, the Debug Port (DP) .....	2-3
2.3	The resource interface, the Access Ports (APs) .....	2-5
2.4	ARM Debug Interface implementation examples .....	2-7
2.5	Accessing Access Ports .....	2-8

<b>Chapter 3</b>	<b>Common Debug Port (DP) features</b>	
3.1	Sticky flags and DP error responses .....	3-2
3.2	Pushed compare and pushed verify operations .....	3-5
3.3	The Transaction Counter .....	3-8
3.4	System and Debug power and Debug reset control .....	3-9
<b>Chapter 4</b>	<b>The JTAG Debug Port (JTAG-DP)</b>	
4.1	The Debug TAP State Machine introduction .....	4-2
4.2	The scan chain interface .....	4-3
4.3	IR scan chain and IR instructions .....	4-8
4.4	DR scan chain and DR registers .....	4-12
<b>Chapter 5</b>	<b>The Serial Wire Debug Port (SW-DP)</b>	
5.1	Introduction to the DAP Serial Wire Debug Port .....	5-2
5.2	Introduction to the ARM Serial Wire Debug (SWD) protocol .....	5-3
5.3	Serial Wire Debug protocol operation .....	5-5
5.4	Protocol description .....	5-10
<b>Chapter 6</b>	<b>Debug Port Registers</b>	
6.1	Debug Port registers overview .....	6-2
6.2	Debug Port (DP) register descriptions .....	6-6
<b>Chapter 7</b>	<b>Common Access Port (AP) features</b>	
7.1	Overview of Access Ports (APs) .....	7-2
7.2	The identification model for Access Ports .....	7-3
7.3	Selecting and accessing an AP .....	7-4
<b>Chapter 8</b>	<b>The Memory Access Port (MEM-AP)</b>	
8.1	Overview of the function of a Memory Access Port (MEM-AP) .....	8-2
8.2	MEM-AP functions .....	8-8
8.3	MEM-AP examples of pushed verify and pushed compare .....	8-22
8.4	MEM-AP implementation requirements .....	8-24
<b>Chapter 9</b>	<b>The JTAG Access Port (JTAG-AP)</b>	
9.1	Overview of the function of a JTAG Access Port (JTAG-AP) .....	9-2
9.2	The JTAG Engine Byte Command Protocol .....	9-10
<b>Chapter 10</b>	<b>Access Port (AP) Registers Overview and the Common AP Register</b>	
10.1	Access Port (AP) registers overview .....	10-2
10.2	The common Access Port register, the IDR .....	10-3
<b>Chapter 11</b>	<b>Memory Access Port (MEM-AP) Registers</b>	
11.1	Memory Access Port (MEM-AP) register summary .....	11-2
11.2	MEM-AP detailed register descriptions .....	11-5

<b>Chapter 12</b>	<b>JTAG Access Port (JTAG-AP) Registers</b>	
12.1	JTAG Access Port (JTAG-AP) register summary .....	12-2
12.2	JTAG-AP detailed register descriptions .....	12-5
<b>Chapter 13</b>	<b>Component and Peripheral ID Registers</b>	
13.1	Component and Peripheral ID registers .....	13-2
13.2	The Component ID Registers .....	13-4
13.3	The Peripheral ID Registers .....	13-9
<b>Chapter 14</b>	<b>ROM Tables</b>	
14.1	ROM Table overview .....	14-2
14.2	ROM Table entries .....	14-5
14.3	The MEMTYPE Register .....	14-9
14.4	Component and Peripheral ID Registers .....	14-10
14.5	ROM Table hierarchies .....	14-12
	<b>Glossary</b>	





# List of Tables

## ARM Debug Interface v5 Architecture Specification

	Change History .....	ii
Table 2-1	Comparison of IEEE 1149.1 and JTAG-DP naming .....	2-3
Table 4-1	JTAG-DP signal connections .....	4-4
Table 4-2	Standard IR instructions .....	4-9
Table 4-3	Recommended implementation defined IR instructions for IEEE 1149.1-compliance .....	4-10
Table 4-4	DPACC and APACC ACK responses .....	4-14
Table 4-5	JTAG-DP target response summary, when previous scan was a DPACC access . 4-19	
Table 4-6	JTAG-DP target response summary, when previous scan was an APACC access 4-20	
Table 4-7	Summary of JTAG-DP host responses .....	4-21
Table 5-1	Target response summary for DP read transaction requests .....	5-16
Table 5-2	Target response summary for AP read transaction requests .....	5-17
Table 5-3	Target response summary for DP write transaction requests .....	5-17
Table 5-4	Target response summary for AP write transaction requests .....	5-18
Table 5-5	Summary of host (debugger) responses to the SW-DP acknowledge .....	5-19
Table 6-1	Summary of Debug Port registers .....	6-2
Table 6-2	JTAG-DP register map .....	6-3
Table 6-3	SW-DP register map .....	6-5
Table 6-4	AP Abort Register bit assignments .....	6-6

Table 6-5	Identification Code Register bit assignments .....	6-9
Table 6-6	JEDEC JEP-106 manufacturer ID code, with ARM Limited values .....	6-9
Table 6-7	Control/Status Register bit assignments .....	6-10
Table 6-8	Control of pushed operation comparisons by MASKLANE .....	6-14
Table 6-9	Transfer Mode, TRNMODE, bit definitions .....	6-14
Table 6-10	Bit assignments for the AP Select Register, SELECT .....	6-16
Table 6-11	CTRLSEL field bit definitions .....	6-17
Table 6-12	Bit assignments for the Wire Control Register (SW-DP only) .....	6-19
Table 6-13	Turnaround tri-state period field, TURNROUND, bit definitions .....	6-19
Table 6-14	Wire operating mode, WIREMODE, bit definitions .....	6-20
Table 8-1	Summary of AddrInc field values .....	8-9
Table 8-2	Size field values when the MEM-AP supports different access sizes .....	8-13
Table 8-3	Byte-laning of memory accesses from the DRW .....	8-14
Table 8-4	Use of DbgSwEnable bit, bit [31], to control a slave memory port .....	8-20
Table 8-5	Use of DbgSwEnable bit, bit [31], to control the DBGSWENABLE signal .....	8-20
Table 9-1	JTAG Access Port JTAG port signals .....	9-9
Table 9-2	Summary of JTAG Engine Byte Command Protocol .....	9-10
Table 9-3	TMS packet encodings .....	9-11
Table 9-4	TDI_TDO first byte (opcode) format .....	9-13
Table 9-5	TDI_TDO second byte (length byte), packed format .....	9-16
Table 10-1	Summary of the common Access Port (AP) register .....	10-3
Table 10-2	AP Identification Register, IDR, bit assignments .....	10-4
Table 10-3	ARM AP Identification types .....	10-5
Table 11-1	Summary of Memory Access Port (MEM-AP) registers .....	11-2
Table 11-2	Access information for the MEM-AP registers .....	11-3
Table 11-3	Bit assignments for the MEM-AP Control/Status Word Register, CSW .....	11-6
Table 11-4	Bit assignments for the Transfer Address Register, TAR .....	11-8
Table 11-5	Bit assignments for the Data Read/Write Register, DRW .....	11-8
Table 11-6	Mapping of Banked Data Registers onto memory addresses .....	11-9
Table 11-7	Bit assignments for the Banked Data Registers, BD0 to BD3 .....	11-9
Table 11-8	Bit assignments for the Configuration Register, CFG, .....	11-10
Table 11-9	Bit assignments for the Debug Base Address Register, BASE, .....	11-12
Table 11-10	Legacy Debug Base Address Register format when there are no debug entries ... 11-14	
Table 11-11	Legacy Debug Base Address Register format when holding a base address value 11-14	
Table 12-1	Summary of JTAG Access Port (JTAG-AP) registers .....	12-2
Table 12-2	Bank and Offset values for accessing the JTAG-AP registers .....	12-3
Table 12-3	Bit assignments for the JTAG-AP Control/Status Word Register, CSW .....	12-5
Table 12-4	Bit assignments for the JTAG-AP Port Select Register, PSEL .....	12-8
Table 12-5	Bit assignments for the JTAG-AP Port Status Register, PSTA .....	12-10
Table 13-1	Summary of Component and Peripheral ID Registers .....	13-2
Table 13-2	Summary of the Component Identification Registers .....	13-4
Table 13-3	Component Class values in the Component ID .....	13-5
Table 13-4	Component ID0 Register bit assignments .....	13-6
Table 13-5	Component ID1 Register bit assignments .....	13-7
Table 13-6	Component ID2 Register bit assignments .....	13-7

Table 13-7	Component ID3 Register bit assignments .....	13-8
Table 13-8	Summary of the peripheral identification registers .....	13-9
Table 13-9	Register fields for the peripheral identification registers .....	13-10
Table 13-10	Peripheral ID0 Register bit assignments .....	13-12
Table 13-11	Peripheral ID1 Register bit assignments .....	13-13
Table 13-12	Peripheral ID2 Register bit assignments .....	13-14
Table 13-13	Peripheral ID3 Register bit assignments .....	13-14
Table 13-14	Peripheral ID4 Register bit assignments .....	13-15
Table 13-15	Peripheral ID5 to Peripheral ID7 Registers, bit assignments .....	13-16
Table 13-16	Identification Registers for a legacy component .....	13-17
Table 14-1	ROM Table regions .....	14-2
Table 14-2	Format of a ROM Table entry .....	14-5
Table 14-3	MEMTYPE Register bit assignments .....	14-9



# List of Figures

## ARM Debug Interface v5 Architecture Specification

	Key to timing diagram conventions .....	xxi
Figure 1-1	Subdivisions of an ARM Debug Interface .....	1-5
Figure 1-2	Debug Port options and their connections .....	1-7
Figure 1-3	Simple ARM Debug Interface MEM-AP Implementation .....	1-8
Figure 1-4	Simple ARM Debug Interface JTAG-AP Implementation .....	1-8
Figure 1-5	Complex ARM Debug Interface, showing the Access Port options .....	1-9
Figure 2-1	Simple example of an ARM Debug Interface implementation, with ROM Table .....	2-7
Figure 2-2	Structure of the Debug Access Port, showing JTAG-DP accesses to a generic AP .....	2-9
Figure 3-1	Pushed operations overview .....	3-6
Figure 3-2	Power-up request and acknowledgement timing .....	3-12
Figure 3-3	Emulation of power-up control .....	3-13
Figure 3-4	Generation of ACK signals from REQ and ACITVE signals .....	3-13
Figure 3-5	Signal generation for a single power domain .....	3-14
Figure 3-6	Reset request and acknowledge timing .....	3-14
Figure 4-1	Mapping of the JTAG-DP scan chains onto the logical levels of the ARM Debug Interface .....	4-3
Figure 4-2	JTAG-DP physical connection .....	4-4
Figure 4-3	The Debug TAP State Machine (DBGTAPSM) .....	4-5
Figure 4-4	JTAG-DP Instruction Register operation .....	4-8

Figure 4-5	JTAG-DP Bypass Register operation .....	4-12
Figure 4-6	JTAG-DP Device ID Code Register operation .....	4-13
Figure 4-7	Operation of JTAG-DP DP Access and AP Access Registers .....	4-15
Figure 4-8	JTAG-DP ABORT scan chain operation .....	4-22
Figure 5-1	Serial Wire Debug successful write operation .....	5-7
Figure 5-2	Serial Wire Debug successful read operation .....	5-8
Figure 5-3	Serial Wire Debug WAIT response to a packet request .....	5-8
Figure 5-4	Serial Wire Debug FAULT response to a packet request .....	5-9
Figure 5-5	Serial Wire Debug protocol error after a packet request .....	5-9
Figure 5-6	Serial Wire WAIT or FAULT response to a read operation when overrun detection is enabled .....	5-13
Figure 5-7	Serial Wire WAIT or FAULT response to a write operation when overrun detection is enabled .....	5-13
Figure 6-1	AP Abort Register bit assignments .....	6-6
Figure 6-2	Identification Code Register bit assignments .....	6-8
Figure 6-3	Control/Status Register bit assignments .....	6-10
Figure 6-4	Select Register, SELECT, bit assignments .....	6-16
Figure 6-5	Wire Control Register bit assignments (SW-DP only) .....	6-18
Figure 8-1	MEM-AP connecting the DP to debug components .....	8-4
Figure 9-1	JTAG-AP connecting the DP to JTAG devices .....	9-3
Figure 9-2	Structure of the JTAG Access Port (JTAG-AP) .....	9-5
Figure 9-3	TMS packet example with TDI held at 1 .....	9-11
Figure 9-4	TMS packet example with TDI held at 0 .....	9-12
Figure 9-5	TDI_TDO first byte (opcode) format .....	9-12
Figure 9-6	TDI_TDO second byte (length byte), normal format .....	9-14
Figure 9-7	TDI_TDO second byte (length byte), packed format .....	9-15
Figure 9-8	TDI_TDO formatting example. Complete packet for a scan of 21 TCK cycles .....	9-17
Figure 9-9	TDI_TDO response data formatting example. Scan of 21 TCK cycles .....	9-18
Figure 10-1	The AP Identification Register, IDR, for an ARM Limited AP implementation .....	10-3
Figure 11-1	MEM-AP Control/Status Word Register, CSW, bit assignments .....	11-5
Figure 11-2	Configuration Register, CFG, bit assignments .....	11-10
Figure 11-3	Debug Base Address Register, BASE, bit assignments .....	11-11
Figure 12-1	JTAG-AP Control/Status Word Register, CSW, bit assignments .....	12-5
Figure 12-2	JTAG-AP Port Select Register, PSEL, bit assignments .....	12-7
Figure 12-3	JTAG-AP Port Status Register, PSTA, bit assignments .....	12-9
Figure 12-4	Bit assignments for the Byte Read FIFO Registers, BRFIFO1 to BRFIFO4 .....	12-11
Figure 12-5	Bit assignments for the Byte Write FIFO Registers, BWFIFO1 to BWFIFO4 .....	12-11
Figure 13-1	Mapping between the Component ID Registers and the Component ID value .....	13-4
Figure 13-2	Component ID0 Register bit assignments .....	13-6
Figure 13-3	Component ID1 Register bit assignments .....	13-6
Figure 13-4	Component ID2 Register bit assignments .....	13-7
Figure 13-5	Component ID3 Register bit assignments .....	13-8
Figure 13-6	Mapping between the Peripheral ID Registers and the Peripheral ID value ..	13-9
Figure 13-7	Peripheral ID fields .....	13-10
Figure 13-8	Peripheral ID0 Register bit assignments .....	13-12
Figure 13-9	Peripheral ID1 Register bit assignments .....	13-12

Figure 13-10	Peripheral ID2 Register bit assignments .....	13-13
Figure 13-11	Peripheral ID3 Register bit assignments .....	13-14
Figure 13-12	Peripheral ID4 Register bit assignments .....	13-15
Figure 13-13	Peripheral ID5 to Peripheral ID7 Registers, bit assignments .....	13-16
Figure 14-1	ROM Table formats, for 8-bit and 32-bit ROM .....	14-3
Figure 14-2	MEMTYPE Register bit assignments .....	14-9
Figure 14-3	ROM Table hierarchy example .....	14-12
Figure 14-4	Prohibited duplicate ROM Table reference .....	14-14
Figure 14-5	Prohibited circular ROM Table references .....	14-14





# Preface

This preface introduces the *ARM Debug Interface v5 Architecture Specification*. It contains the following sections:

- *About this specification* on page xviii
- *Conventions* on page xx
- *Further reading* on page xxii
- *Feedback* on page xxiii.

## About this specification

This is the *Architecture Specification* for the *ARM Debug Interface v5* (ADIV5).

### Intended audience

This specification is written for system designers and engineers who are specifying, designing or implementing a debug interface to the ADIV5 architecture specification. This includes system designers and engineers who are specifying, designing or implementing a *System-on-Chip* (SoC) device that incorporates a debug interface that complies with the ADIV5 specification.

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

This specification is also intended for engineers who are working with a debug interface that conforms to the ADIV5 specification. This includes:

- designers and engineers who are specifying, designing or implementing hardware debuggers
- those specifying, designing or writing debug software.

These engineers have no control over the design decisions made in the ARM Debug Interface implementation to which they are connecting, but must be able to identify the ADI components that are present, and understand how they operate.

### Using this manual

This specification is organized into the following chapters:

#### **Chapter 1** *Introduction*

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 made when implementing an ADI.

#### **Chapter 2** *Overview of the ARM Debug Interface and its components*

Read this chapter for a simple description of each of the components that might be included in an ADI, and for an introduction to the register model for accessing an ADI.

#### **Chapter 3** *Common Debug Port (DP) features*

Read this chapter for a description of the features that must be implemented on the *Debug Port* (DP) of an ADI.

Every ADI includes a single DP, either a *JTAG Debug Port* (JTAG-DP) or a *Serial Wire Debug Port* (SW-DP)

#### **Chapter 4** *The JTAG Debug Port (JTAG-DP)*

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 are used to access the JTAG-DP.

**Chapter 5 *The Serial Wire Debug Port (SW-DP)***

Read this chapter for a description of the *Serial Wire Debug Port* (SW-DP), and the Serial Wire Debug protocol used for accesses to a SW-DP.

**Chapter 6 *Debug Port Registers***

Read this chapter for a description of the Debug Port Registers.

Most of these registers are common to the JTAG-DP and the SW-DP, and the chapter describes the differences between the JTAG-DP and the SW-DP register implementations.

**Chapter 7 *Common Access Port (AP) features***

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

**Chapter 8 *The Memory Access Port (MEM-AP)***

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

**Chapter 9 *The JTAG Access Port (JTAG-AP)***

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

**Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register***

Read this chapter for an overview of the AP registers, and for a description of the registers that are common to all ADI APs.

**Chapter 11 *Memory Access Port (MEM-AP) Registers***

Read this chapter for a description of the MEM-AP register implementation.

**Chapter 12 *JTAG Access Port (JTAG-AP) Registers***

Read this chapter for a description of the JTAG-AP register implementation.

**Chapter 13 *Component and Peripheral ID Registers***

Read this chapter for a description of the Component and Peripheral ID Registers. These registers are part of the register space of every debug component that complies with the ADIv5 architecture specification.

**Chapter 14 *ROM Tables***

Read this chapter for a description of ARM debug component ROM Tables. Any ADI can include a ROM Table, and an ADI with more than one debug component must include at least one ROM Table.

**Glossary**

Read the Glossary for definitions of some of the terms used in this specification.

## Conventions

Conventions that this specification can use are described in:

- *Typographic*
- *Timing diagrams*
- *Signals* on page xxi.

## Typographic

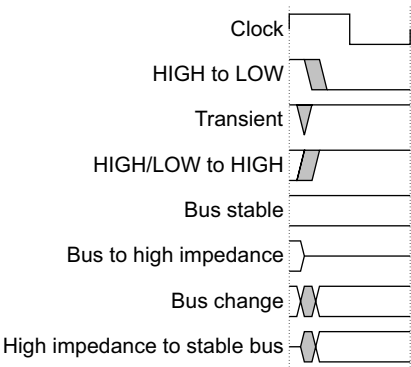
The typographical conventions are:

<i>italic</i>	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
<b>bold</b>	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
monospace	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
<b>SMALL CAPITALS</b>	Denote a few terms that have specific technical meanings. Their meanings can be found in the Glossary.
<b>&lt; and &gt;</b>	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. These terms appear in normal font in running text. For example: <ul style="list-style-type: none"> <li>• MRC p15, 0 &lt;Rd&gt;, &lt;CRn&gt;, &lt;CRm&gt;, &lt;Opcode_2&gt;</li> <li>• The Opcode_2 value selects which register is accessed.</li> </ul>

## Timing diagrams

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

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.



Key to timing diagram conventions

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 and LOW for active-LOW signals.
- Lower-case n**           Denotes an active-LOW signal.
- Prefix DBG**            Denotes debug signals.

## Further reading

This section lists publications by ARM Limited, and by third parties.

ARM Limited periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets, addenda, and the Frequently Asked Questions list.

## ARM publications

This specification contains information that is specific to the ARM Debug Interface architecture specification. See the following documents for other relevant information:

- *ETM Architecture Specification* (ARM IHI 0014)
- *CoreSight Design Kit Technical Reference Manual* (ARM DDI 0314)
- *CoreSight Architecture Specification* (ARM IHI 0029)
- *ARM Architecture Reference Manual Debug Supplement* (ARM DDI 0379).

## Other publications

This section lists relevant documents published by third parties:

- *IEEE 1149.1-2001 IEEE Standard Test Access Port and Boundary Scan Architecture* (JTAG)
- *JEP106M, Standard Manufacture's Identification Code*, JEDEC Solid State Technology Association.

## Feedback

ARM Limited welcomes feedback on the ARM Debug Interface architecture specification.

### Feedback on this specification

If you have any comments on this specification, send email to [errata@arm.com](mailto:errata@arm.com) giving:

- the document title
- the document number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.





# Chapter 1

## Introduction

This chapter introduces the ARM Debug Interface architecture and summarizes the design decisions required for an ARM Debug Interface implementation. It contains the following sections:

- *About the ARM Debug Interface version 5 (ADIV5) on page 1-2*
- *The function of the ARM Debug Interface on page 1-3*
- *The subdivisions of an ARM Debug Interface on page 1-5*
- *Design choices for an ARM Debug Interface v5 implementation on page 1-6.*

## 1.1 About the ARM Debug Interface version 5 (ADIV5)

ADIV5 is the fifth major version of the ARM Debug Interface.

All previous versions of the ADI 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 Limited recommends that ADIV4 is used with implementations of the ARMv6 architecture.

ADIV5 removes the link between the ADI and ARM processor cores. An implementation of ADIV5 can access any debug component that complies with the ADIV5 specification. The details of the resources accessed by ADIV5 are defined by the resources, rather than the ADI.

ADIV5 has three major advantages:

- ADIV5 interfaces can access a greater range of devices.
- Implementation of the ADI can be separated from implementation of the resource, enabling greater reuse of implementations. However, this separation is not required.
- Use of the ADIV5 abstractions permits reuse of software tools, such as debuggers.

Versions 1 to 4 of the debug interface 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 low pin count Serial Wire interface.

## 1.2 The function of the ARM Debug Interface

This chapter describes the context in which an *ARM Debug Interface* (ADI) might be implemented.

This section gives a summary of the types of debug functionality provided by debug components in an embedded *System on Chip* (SoC). This information is given in the subsections:

- *Embedded core debug functionality.*
- *System debug functionality* on page 1-4.

The ADI is designed to provide easy access to Soc debug components.

See *Compatibility between CoreSight and ARM Debug interfaces* on page 1-4 for information about compatibility with the CoreSight™ architecture.

### 1.2.1 Embedded core debug functionality

To enable applications to be debugged, an embedded microprocessor might provide facilities that enable you to:

- Modify the state of the processor. An external host must be able to modify the contents of the internal registers and the memory system.
- Determine the state of the processor. An external host must be able to read the values of the internal registers, and read from the memory system.
- 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*.
- Force the processor to enter or exit Debug state from an external system.
- Determine when the core enters or leaves Debug state.
- Trace program flow around programmable events.

Two examples of technologies that provide these facilities are:

- the ARMv7 Debug Architecture, see the *ARM Architecture Reference Manual Debug Supplement*
- the Embedded Trace Macrocell, see the *ETM Architecture Specification*.

In addition, the ARM Debug Interface v5 enables you to access legacy components that implement an IEEE 1149.1 JTAG interface. This means that the ADIv5 can access processors that implement earlier versions of the ADI.

### 1.2.2 System debug functionality

Debug extends beyond the boundaries of an embedded microprocessor core. Engineers must be able to debug:

- components within an embedded SoC
- the interconnection fabric of the system.

Examples of debug facilities that might be provided at the system level, outside the embedded microprocessor core, include:

- Facilities to:
  - Examine the state of the system, including state that might not be visible to the embedded microprocessor core.
  - Trace accesses on the interconnection fabric. These might be accesses by the microprocessor core, or accesses by other devices such as *Direct Memory Access* (DMA) engines.
- Mechanisms for low-intrusion diagnostic messaging between software and a debugger.
- Cross-triggering mechanisms that enable debug components to signal each other.
- A fabric for the efficient streaming and collection of diagnostic information, such as program trace.

Ideally, functions operate with minimal change to the behavior of the system being debugged.

Examples of technologies that provide these facilities are:

- the ADI
- the CoreSight debug architecture, and the CoreSight components:
  - CoreSight *AHB Trace Macrocell* (HTM)
  - CoreSight *Instrumentation Trace Macrocell* (ITM)
  - CoreSight *Embedded Cross-Trigger* (ECT)
  - the *AMBA™ Trace Bus* (ATB) components.

For more information see the *CoreSight Architecture Specification* and the *CoreSight Design Kit Technical Reference Manual*.

### 1.2.3 Compatibility between CoreSight and ARM Debug interfaces

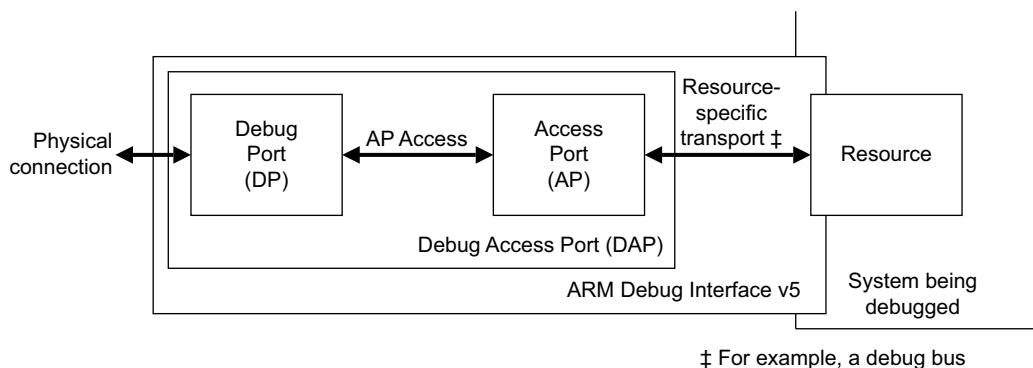
ADIV5 is designed to be compatible with the ARM CoreSight architecture:

- a CoreSight interface implementation is a valid implementation of ADIV5
- the ADIV5 specification does not require an ADI to be CoreSight compliant.

The *CoreSight Design Kit Technical Reference Manual* includes a description of the CoreSight interface, the CoreSight Debug Access Port.

## 1.3 The subdivisions of an ARM Debug Interface

Figure 1-1 shows the logical subdivisions of an ADIV5.



**Figure 1-1 Subdivisions of an ARM Debug Interface**

### Note

Although this specification logically divides the ARM Debug Interface into the elements shown in Figure 1-1, this specification does not require implementations to be structured in this way. This specification describes the programmer's model for the ADI, and these subdivisions give a convenient representation of the programmer's model.

### 1.3.1 The conventional model of the ADI

Figure 1-1 shows a conventional model for using an ADI. In this, the Access Port of the ADI connects through a resource-specific transport, such as a debug bus, to a resource that is part of the system being debugged. This model is used extensively in this specification.

The ADI components defined by this specification include two alternatives for the resource-specific transport that connects the ADI to the system being debugged:

- An ADI can be used to connect to a memory-mapped resource, such as a debug peripheral. For such connections the ADI defines a *Memory Access Port* (MEM-AP) programmer's model.
- An ADI can be used to connect to a legacy IEEE 1149.1 JTAG device. For such connections the ADI defines a *JTAG Access Port* (JTAG-AP) and associated programmer's model.

This specification does not mandate the transport between the AP and the resource. In particular, it does not require an MEM-AP to use a bus to connect to the system being debugged. For example, an ADI might be directly integrated into the resource. However, referring to Figure 1-1, logically a MEM-AP always accesses a memory-mapped resource in the *system being debugged*. For this reason, this specification describes MEM-AP accesses to the system being debugged as *memory accesses*.

## 1.4 Design choices for an ARM Debug Interface v5 implementation

*The subdivisions of an ARM Debug Interface* on page 1-5 introduces the logical subdivisions of an ARM Debug Interface, shown in Figure 1-1 on page 1-5. This section outlines the design choices that must be made before implementing an ARM Debug Interface. The following subsections consider each of the two functional blocks of the interface:

- *Choices for the Debug Port (DP)*
- *Choices for the Access Ports (APs)* on page 1-7.

---

### Note

This Architecture Specification is written for engineers wanting to implement an ARM Debug Interface, and for those using an ARM Debug Interface. If you are reading the Specification to learn more about a particular ARM Debug Interface implementation you must understand the design choices that have been made in that implementation. Those choices might be implicit in the connections to the Debug Interface. If you are uncertain about the choices you must obtain details from the implementor of the Debug Interface.

---

The Debug Port and Access Port implementations are described in more detail in Chapter 2 *Overview of the ARM Debug Interface and its components*.

### 1.4.1 Choices for the Debug Port (DP)

An ARM Debug Interface has a single Debug Port. There are two implementation options for the DP, and these are summarized in the following subsections:

- *The Serial Wire Debug Port (SW-DP)*
- *The JTAG Debug Port (JTAG-DP)* on page 1-7.

If you are designing or specifying an ARM Debug Interface you must decide which of these Debug Ports to implement.

---

### Note

ARM Limited might define other DPs in the future.

---

### The Serial Wire Debug Port (SW-DP)

The SW-DP is a low pin count interface that uses a packet-based protocol to read or write register information. It is based on the CoreSight Serial Wire interface.

---

### Note

Contact ARM Limited for details of tools support for the Serial Wire Debug Port.

---

## The JTAG Debug Port (JTAG-DP)

The JTAG-DP is based on the *IEEE 1149.1 Test Access Port (TAP) and Boundary Scan Architecture*, widely referred to as JTAG.

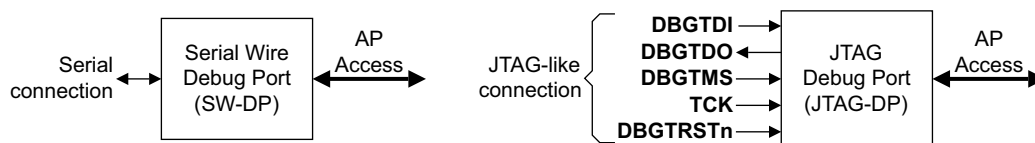
With the JTAG-DP, IEEE 1149.1 scan chains are used to read or write register information.

### Note

Only the logical Debug TAP State Machine (DBGTAPSM) architecture and associated support instructions and scan chain are included in this specification. The precise physical interface and mechanism for performing DBGTAPSM transactions are not mandated, and are not described here. However, ARM Limited recommends that an interface compatible with the IEEE 1149.1 standard is used.

## Connections to the Debug Port

Figure 1-2 shows the connections to each of the Debug Port implementations:

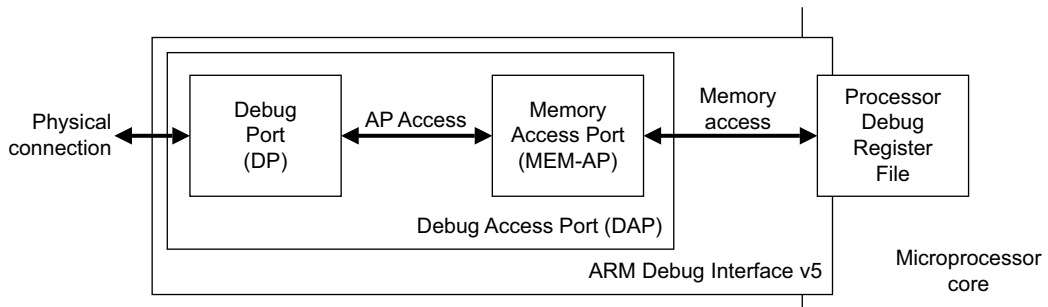


**Figure 1-2 Debug Port options and their connections**

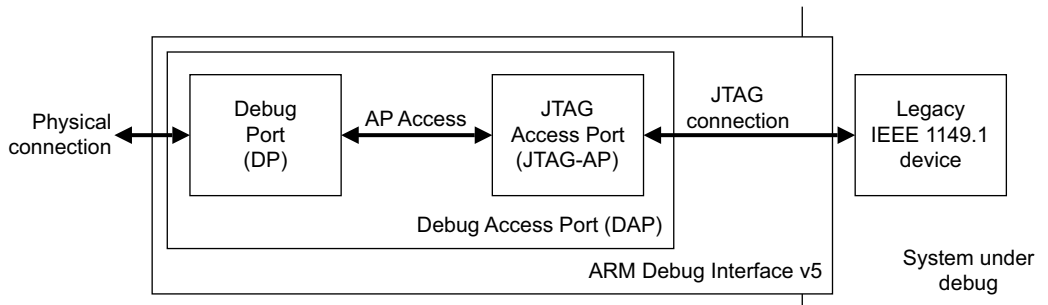
### 1.4.2 Choices for the Access Ports (APs)

An ARM Debug Interface always includes at least one Access Port, and might contain multiple APs. The simplest ARM Debug Interface uses a single AP to connect to a single debug component. The basic examples of this are:

- using an MEM-AP to connect to a single debug component, such as a microprocessor core, as shown in Figure 1-3 on page 1-8
- using a JTAG-AP to connect to a single legacy IEEE 1149.1 device, as shown in Figure 1-4 on page 1-8.



**Figure 1-3 Simple ARM Debug Interface MEM-AP Implementation**



**Figure 1-4 Simple ARM Debug Interface JTAG-AP Implementation**

However, because a single ADI can include multiple APs, the design choices relating to APs are at two levels:

- Choices about the number of APs in the ADI, and whether each AP is a MEM-AP or a JTAG-AP. These decisions are considered in *Top-level AP planning choices*.
- The choices that have to be made for each implemented AP. These choices are considered in:
  - *Choices for each JTAG-AP* on page 1-11
  - *Choices for each MEM-AP* on page 1-10.

### Top-level AP planning choices

In a more complex debug system, three ways in which an AP can be implemented are:

- As a *Memory Access Port* (MEM-AP) with a memory-mapped Debug Bus connection. The Debug Bus connects directly to one or more Debug Register Files.
- As a Memory Access Port 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 Access Port* (JTAG-AP). This connects directly to one or more JTAG devices, and enables connection to legacy hardware components.

---

**Note**

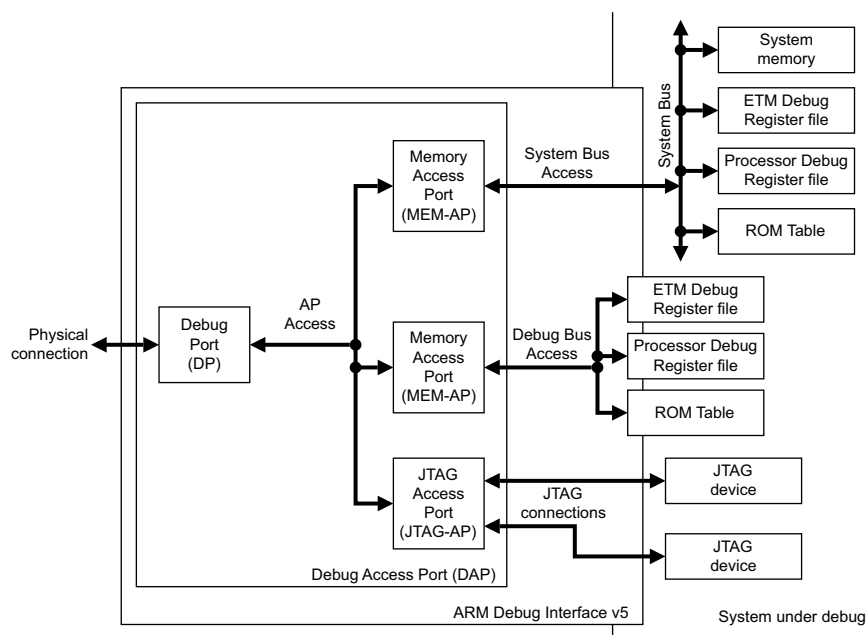
---

The connection between legacy hardware components and a JTAG-AP is defined by the JTAG standard. For more information see Chapter 9 *The JTAG Access Port (JTAG-AP)*.

---

If you are designing or specifying an ARM Debug Interface you must decide how many APs are required, and how each of them is implemented. This depends largely on the debug components of the system to which your ADI connects.

Figure 1-5 shows a more complex ARM Debug Interface, and illustrates the different Access Port options.



**Figure 1-5 Complex ARM Debug Interface, showing the Access Port options**

---

**Note**


---

The ARM Debug Interface v5 architecture specification:

- Supports additional Access Ports. Every Access Port must follow the base standard for identification given in this specification. Debuggers must be able to ignore Access Ports that they do not recognize.
  - Permits multiple register files to be accessed by a single Memory Access Port. This specification includes a base standard for register file identification, and debuggers must be able to ignore register files that they do not recognize or do not support.
  - Permits a Memory Access Port to access a mixture of system memory and Debug Register files.
- 

In any ARM Debug Interface v5 implementation, such as those illustrated in Figure 1-3 on page 1-8 and Figure 1-5 on page 1-9, the Debug Port can be either an SW-DP or a JTAG-DP. The physical connections to each of these options are shown in Figure 1-2 on page 1-7.

### Choices for each MEM-AP

The main decisions to be made for a MEM-AP concern the connection between the MEM-AP and the memory-mapped debug components connected to it. These decisions depend, largely, on the requirements of those debug components. They include:

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

If an ADI MEM-AP connects to more than one debug component then the system must include one or more *ROM Tables*, to provide information about the debug system. However, a system designer might chose to include a ROM Table in a system that has only one other component. See *ROM Tables* on page 8-2 for more information.

It is IMPLEMENTATION DEFINED whether a MEM-AP includes certain features, for more information see:

- *MEM-AP functions* on page 8-8
- *MEM-AP implementation requirements* on page 8-24.

The implementation of the connection between the MEM-AP and the debug components can determine whether some of these features are included. In particular, certain features must be included if the connection is less than 32-bits wide. The debug components themselves might place limitations on the connection, for example a component might require 32-bit accesses.

For more information about implementing a MEM-AP see Chapter 8 *The Memory Access Port (MEM-AP)*.

## Choices for each 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.

In addition, a single JTAG scan chain can contain multiple TAPs. However ARM Limited recommends that each scan chain connected to a JTAG-AP contains only one TAP.

Therefore, the design decisions that must be made for each JTAG-AP are:

- How many JTAG scan chains are connected to the JTAG-AP?
- Do any of the scan chains contain more than one TAP?

For more information see Chapter 9 *The JTAG Access Port (JTAG-AP)*.



# Chapter 2

## Overview of the ARM Debug Interface and its components

This section introduces the components of the ARM Debug Interface, in more detail than the descriptions in Chapter 1 *Introduction*. It also describes how the *Debug Access Port* (DAP) is used to access registers within the DAP and in the connected system. This description is based on JTAG-DP accesses to a Debug Interface, and shows how registers are located at different levels within the Debug Interface model.

This chapter contains the following sections:

- *ARM Debug Interface overview* on page 2-2
- *The external interface, the Debug Port (DP)* on page 2-3
- *The resource interface, the Access Ports (APs)* on page 2-5
- *ARM Debug Interface implementation examples* on page 2-7
- *Accessing Access Ports* on page 2-8.

## 2.1 ARM Debug Interface overview

Logically, the ARM Debug Interface (ADI) consists of:

- A number of registers that are private to the ADI. These are referred to as the *Debug Access Port* (DAP) Registers.
- A means to access the DAP registers.
- A means to access the debug registers of the debug components to which the ADI is connected.

This logical organization of the ADI is shown in Figure 1-1 on page 1-5.

Because the DAP logically consists of two parts, the Debug Port and Access Ports, it must support two types of access:

- Access to the Debug Port (DP) registers. This is provided by Debug Port accesses (DPACC).
- Access to Access Port (AP) registers. This is provided by Access Port accesses (APACC).

An ADI can include multiple Access Ports.

An AP is responsible for accessing debug component registers, such as processor debug logic, ETM and trace port registers. These accesses are made in response to APACC accesses in a manner defined by the AP.

The method of making these accesses depends on the Debug Port that is implemented, as summarized in *The external interface, the Debug Port (DP)* on page 2-3.

More generally, examples of possible targets of AP accesses include:

- the debug registers of the core processor
- ETM or Trace Port debug registers
- a ROM table, see Chapter 14 *ROM Tables*
- a memory system
- a legacy JTAG device.

## 2.2 The external interface, the Debug Port (DP)

An ARM Debug Interface implementation includes a single *Debug Port* (DP), that provides the external physical connection to the interface. The ARM Debug Interface v5 specification supports two DP implementations:

- the JTAG Debug Port (JTAG-DP), see *The JTAG Debug Port (JTAG-DP)*
- the Serial Wire Debug Port (SW-DP), see *The Serial Wire Debug Port (SW-DP)* on page 2-4.

These alternative DP implementations provide different mechanisms for making Access Port and Debug Port accesses. However, they have a number of common features. In particular, each implementation provides:

- a means of identifying the DAP, using an identification code scheme
- a means of making DP and AP accesses
- a means of aborting a register access that appears to have faulted.

### 2.2.1 The JTAG Debug Port (JTAG-DP)

The JTAG-DP is based on the IEEE 1149.1 *Test Access Port* (TAP) and Boundary Scan Architecture, widely referred to as JTAG. Because the JTAG-DP is intended for accessing debug components, the naming conventions of IEEE 1149.1 are changed, as shown in Table 2-1:

**Table 2-1 Comparison of IEEE 1149.1 and JTAG-DP naming**

IEEE 1149.1	JTAG-DP	JTAG-DP name
<b>TAP</b>	<b>DBGTAP</b>	Debug Test Access Port.
<b>TAPSM</b>	<b>DBGTAPSM</b>	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 4-4 for the complete list of the JTAG-DP signal names.

For more information see *IEEE 1149.1 Test Access Port and Boundary Scan Architecture*.

With the JTAG-DP, IEEE 1149.1 scan chains are used to read or write register information. A pair of scan chain registers are used:

**DPACC**      Used for Debug Port (DP) accesses.

**APACC**      Used for Access Port (AP) accesses. An APACC access might access a register of a debug component of the system to which the interface is connected.

The scan chain model implemented by a JTAG-DP has the concepts of *capturing* the current value of APACC or DPACC, and of *updating* APACC or DPACC with a new value. An update might cause a read or write access to a DAP register that might then cause a read or write access to a debug register of a connected debug component.

The DBGTAP scan chains are described in more detail in Chapter 4 *The JTAG Debug Port (JTAG-DP)*.

### 2.2.2 The Serial Wire Debug Port (SW-DP)

The Serial Wire Debug Port implementation provides a bi-directional serial connection to the ARM Debug Interface. It is based on the CoreSight Serial Wire Interface, and can be implemented as either a synchronous or an asynchronous serial port.

Communications with the SW-DP use a three-phase protocol:

- A host-to-target packet request.
- A target-to-host acknowledge response.
- A data transfer phase, if required. This can be target-to-host or host-to-target, depending on the request made in the first phase.

A packet request from a debugger indicates whether the required access is to a DP register (DPACC) or to an AP register (APACC), and includes a two-bit register address.

This protocol is described in more detail in Chapter 5 *The Serial Wire Debug Port (SW-DP)*.



## 2.3 The resource interface, the Access Ports (APs)

An Access Port provides the interface between an ADI and one or more debug components. This specification defines two Access Ports, and the complete description of each of these is summarized in:

- *Guide to the detailed description of a MEM-AP*
- *Guide to the detailed description of a JTAG-AP on page 2-6.*

ARM Limited might define additional access ports in the future. In addition, an ARM Debug Interface (ADI) might include additional access ports, not specified by ARM Limited.

All Access Ports *must* follow a base standard for identification, and debuggers must be able to recognize and ignore Access Ports that they do not support. For more information see Chapter 7 *Common Access Port (AP) features*.

As described in Chapter 1 *Introduction*:

- The simplest ADI has only one AP. This can be either a MEM-AP or a JTAG-AP.
- More complex ADIs can have multiple APs. These might be:
  - a mixture of MEM-APs and JTAG-APs
  - all MEM-APs
  - all JTAG-APs.

The Debug Port uses exactly the same process for accessing MEM-APs and JTAG-APs. However the connection to the system being debugged is very different for MEM-APs and JTAG-APs. *Accessing Access Ports* on page 2-8 describes how to access any AP (MEM-AP or JTAG-AP), and summarizes how this gives access to resources in the system being debugged.

### 2.3.1 Guide to the detailed description of a MEM-AP

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

- 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 7 *Common Access Port (AP) features*
- Chapter 8 *The Memory Access Port (MEM-AP)*.

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

- Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*
- Chapter 11 *Memory Access Port (MEM-AP) Registers*.

The MEM-AP provides access to 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 are described in Chapter 13 *Component and Peripheral ID Registers*.

If the MEM-AP connects to more than one debug component then it must also include at least one ROM Table. ROM tables are accessed through a MEM-AP, and are described in Chapter 14 *ROM Tables*.

---

**Note**

As shown in *Implementation examples with a single debug component* on page 2-7, an ADI with only one functional debug component might also implement a ROM Table.

---

### 2.3.2 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 7 *Common Access Port (AP) features*
- Chapter 9 *The JTAG Access Port (JTAG-AP)*.

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

- Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*
- Chapter 12 *JTAG Access Port (JTAG-AP) Registers*.

---

**Note**

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 of the use of this connection are outside the scope of this specification.

---

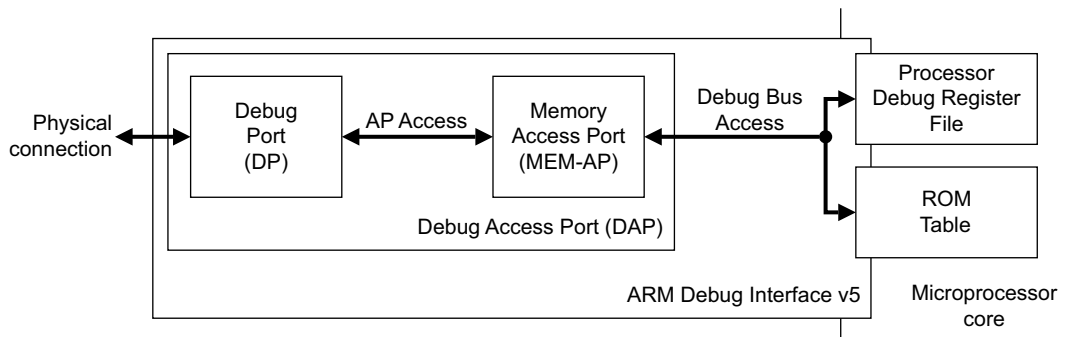
## 2.4 ARM Debug Interface implementation examples

To summarize the description of the ARM Debug Interface components it is useful to consider simple and more complex implementation examples.

### 2.4.1 Implementation examples with a single debug component

In the simplest ARM Debug Interface implementations, the interface connects to a single debug component. This is likely to be a microprocessor core that conforms to the ARMv7 Debug Architecture. The simplest possible implementation is shown in Figure 1-3 on page 1-8.

However, *ROM Tables* on page 8-2 explains that a system with only a single debug component often implements a ROM Table. This gives an implementation similar to the example shown in Figure 2-1.



**Figure 2-1 Simple example of an ARM Debug Interface implementation, with ROM Table**

### 2.4.2 Complex implementation example

In a more complex system, there can be multiple Access Ports, and each access port can be connected to multiple components, or multiple address spaces. An example of a more complex implementation is shown in Figure 1-5 on page 1-9.

In Figure 2-1, and in all other illustrations of the ARM Debug Interface, the Debug Port can be either a Serial Wire Debug Port or a JTAG Debug Port. See Figure 1-2 on page 1-7 for an illustration of the physical connection to each of these Debug Port options.

## 2.5 Accessing Access Ports

The Debug Port on an ADI provides the connection from an external debugger to the interface, and so to the system being debugged. Figure 2-2 on page 2-9 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 is split into two main control units, the Debug Port (DP) and the Access Port (AP), and the physical connection to the debugger is part of the DP. The DAP supports two types of access, Debug Port (DP) accesses and Access Port (AP) accesses. Because Debug Ports usually have serial interfaces, the methods of making these accesses are kept as short as possible. However, all accesses are 32-bits.

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

- Chapter 5 *The Serial Wire Debug Port (SW-DP)*,
- Chapter 6 *Debug Port Registers*.

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

- for a DP register access, these bits determine which register is accessed
- these bits are also carried through into AP accesses, as summarized in Figure 2-2 on page 2-9.

One of the four registers within the DP is the AP Select Register, SELECT. This register specifies a particular Access Port, and a bank of four 32-bit words within the register map of that AP. It enables up to 256 Access Ports to be implemented, and gives access to any one of 16 four-word banks of registers on the selected AP.

In any AP access transaction from the debugger, the two address bits A[3:2] are decoded to select one of the four 32-bit words from the register bank indicated by the SELECT Register. In other words, they select a specific register within the selected four-register bank.

This access model is shown in Figure 2-2 on page 2-9. 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 are explained in greater detail in later sections.

The implementation of this model is shown, also, in:

- Figure 8-1 on page 8-4, for a MEM-AP implementation
- Figure 9-1 on page 9-3, for a JTAG-AP implementation.

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

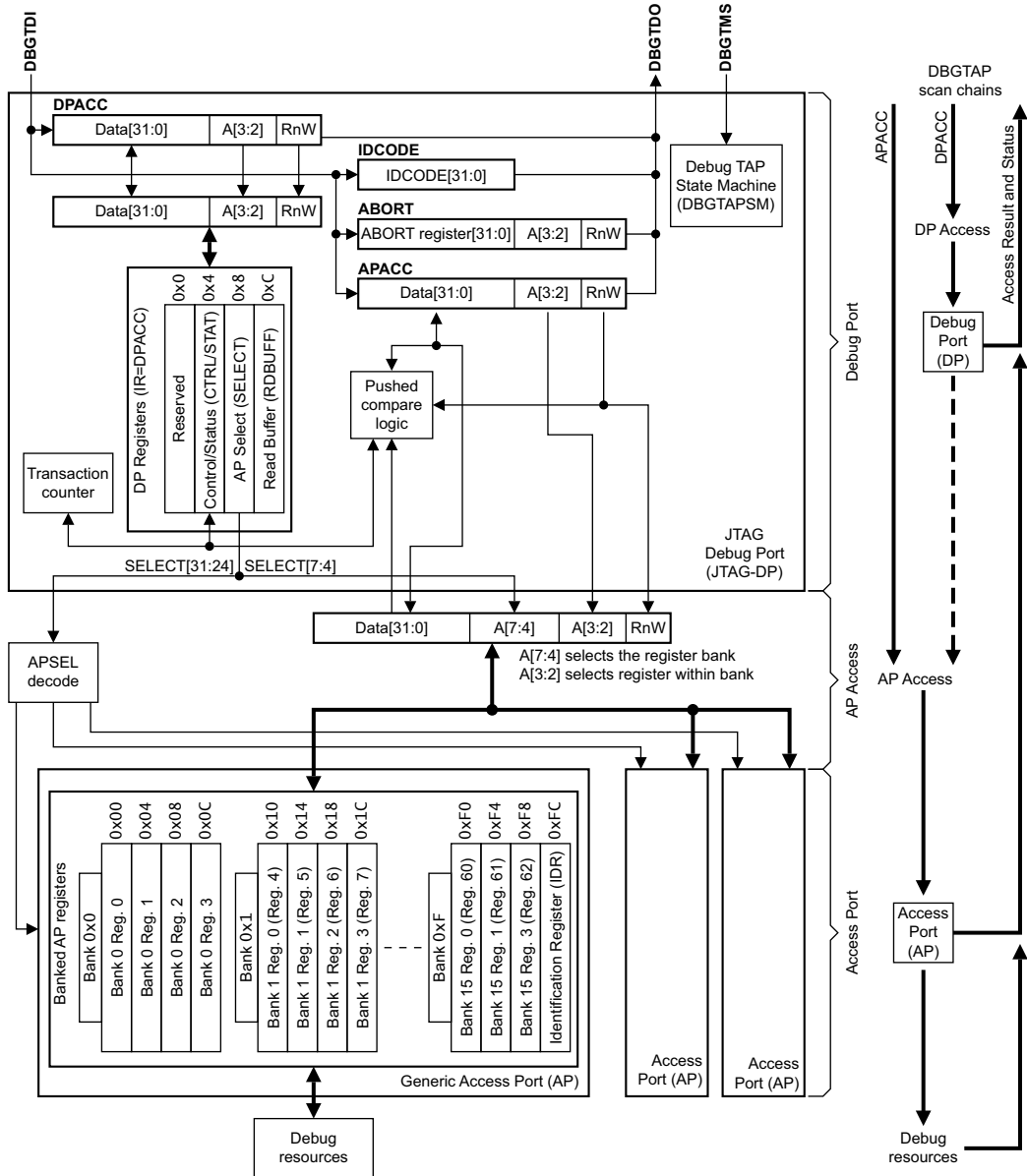


Figure 2-2 Structure of the Debug Access Port, showing JTAG-DP accesses to a generic AP

### 2.5.1 Accessing debug resources

Accessing the AP gives access to the system being debugged, shown as the access to *Debug resources* in Figure 2-2 on page 2-9. The descriptions of the MEM-AP and JTAG-AP given earlier in this chapter show that the details of this access are very different for a MEM-AP and a JTAG-AP. In summary:

- With a MEM-AP, the debug resources are logically memory-mapped, and the connection between the MEM-AP and a debug resource is outside the scope of this specification. Chapter 8 *The Memory Access Port (MEM-AP)* describes the method of accessing these resources, in the section *MEM-AP register accesses and memory accesses* on page 8-6.
- With 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*. More information about accessing the resources is given in Chapter 9 *The JTAG Access Port (JTAG-AP)*.

# Chapter 3

## Common Debug Port (DP) features

This chapter describes the features that must be implemented by every ARM Debug Interface Debug Port. This means these features must be implemented on both Serial Wire Debug Ports (SW-DPs) and on JTAG Debug Ports (JTAG-DPs). Additional information about DPs is given in the following chapters:

- Chapter 4 *The JTAG Debug Port (JTAG-DP)*
- Chapter 5 *The Serial Wire Debug Port (SW-DP)*
- Chapter 6 *Debug Port Registers*.

This chapter contains the following sections:

- *Sticky flags and DP error responses* on page 3-2
- *Pushed compare and pushed verify operations* on page 3-5
- *The Transaction Counter* on page 3-8
- *System and Debug power and Debug reset control* on page 3-9.

## 3.1 Sticky flags and DP error responses

Within both a SW-DP and a JTAG-DP, error conditions are recorded using sticky flags. These sticky flags are described in the following sections:

- *Read and write errors*
- *Overrun detection* on page 3-3
- *Protocol errors, SW-DP only* on page 3-4.

Another sticky flag is used to report the result of pushed operations, see *Pushed compare and pushed verify operations* on page 3-5. This flag, STICKYCMP, behaves in the same way as the sticky flags described in this section.

### ———— Note ————

When set to 1, a sticky flag remains set until it is explicitly cleared to 0. Even if the condition that caused the flag to be set to 1 no longer applies the flag remains set until the debugger clears it to 0.

The method for clearing sticky flags is different for the JTAG-DP and the SW-DP. Table 6-7 on page 6-10, summarizes how these flags are cleared to 0.

Errors can be returned by the DAP itself, or might come from a debug resource, for example, from a memory access made by a MEM-AP to a debug register file of a processor that is powered down.

Within the Debug Port, errors are flagged by sticky flags in the DP Control/Status Register (CTRL/STAT). When an error is flagged the current transaction is completed and any additional APACC (AP Access) transactions are discarded until the sticky flag is cleared to 0.

The DP response to an error condition might be:

- To signal an error response immediately. This happens with the SW-DP.
- To immediately discard all transactions as complete. This happens with the JTAG-DP.

This means that a debugger must check the Control/Status Register after performing a series of APACC transactions, to check if an error occurred. If a sticky flag is set to 1, the debugger clears the flag to 0 and then, if necessary, initiates more APACC transactions to find the cause of the sticky flag condition. Because the flags are sticky the debugger does not have to check the flags after every transaction, it only has to check the Control/Status Register periodically. This reduces the overhead of checking for errors.

### 3.1.1 Read and write errors

A read or write error might occur within the DAP, or come from the resource being accessed. In either case, when the error is detected the Sticky Error flag, STICKYERR, in the Control/Status Register is set to 1.

A read/write error might be generated if the debugger makes an AP transaction request while the debug power domain is powered down. See *System and Debug power and Debug reset control* on page 3-9 for information about power domains.



### 3.1.2 Overrun detection

Debug Ports support an overrun detection mode. This mode enables a user to send blocks of commands to an emulator on a high latency, high throughput connection. These commands should be sent with sufficient in-line delays to make overrun errors unlikely. However the DAP can be programmed so that, if an overrun error occurs, the DAP flags the error by setting the STICKYORUN flag in the DP Control/Status Register to 1. In this overrun detection mode the debugger must check for overrun errors after each sequence of APACC transactions, by checking this flag. For more information see *The Control/Status Register, CTRL/STAT* on page 6-10.

Overrun detection mode is enabled by setting the Overrun Detect bit, ORUNDETECT, in the DP Control/Status Register to 1. While this bit is set to 1, the only permitted response to any transaction is:

- OK/FAULT on the JTAG-DP
- OK on the SW-DP.

In overrun detection mode, any other response at any point is treated as an error and causes the Sticky Overrun flag STICKYORUN in the DP Control/Status Register to be set to 1. The value of the Sticky Error flag, STICKYERR, is not changed.

The debugger must clear STICKYORUN to 0 to enable transactions to resume.

See *The Control/Status Register, CTRL/STAT* on page 6-10 for more information.

#### ————— **Note** —————

The method of clearing the STICKYORUN flag to 0 is different for a JTAG-DP and a SW-DP. See Table 6-7 on page 6-10 for more information.

The behavior of the Debug Port when the Sticky Overrun flag is set to 1 is Debug Port IMPLEMENTATION DEFINED.

If a new transaction is attempted, and results in an overrun error, before an earlier transaction has completed, the first transaction still completes normally. Other sticky flags might be set to 1 on completion of the first transaction.

If the overrun detection mode is disabled, by clearing the ORUNDETECT flag to 0, while STICKYORUN is set to 1, the subsequent value of STICKYORUN is UNPREDICTABLE. To leave overrun detection mode a debugger must:

- check the value of the STICKYORUN bit in the Control/Status register
- clear the STICKYORUN bit to 0, if it is set to 1
- clear the ORUNDETECT bit to 0, to disable overrun detection mode.

### 3.1.3 Protocol errors, SW-DP only

---

#### Note

---

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

---

On the Serial Wire Debug interface, protocol errors can occur, for example because of wire-level errors. These errors might be detected by the parity checks on the data:

- If the SW-DP detects a parity error in a message header, the Debug Port does not respond to the message. The debugger must be aware of this possibility. If it does not receive a response to a message, the debugger must back off. It must then request a read of the IDCODE register, to ensure the Debug Port is responsive, before retrying the original access. For details of the IDCODE register see *The Identification Code Register, IDCODE* on page 6-8.
- If the SW-DP detects a parity error in the data phase of a write transaction, it sets the Sticky Write Data Error flag, WDATAERR, in the Control/Status (CTRL/STAT) Register. Subsequent accesses from the debugger, other than IDCODE, CTRL/STAT or ABORT, result in a FAULT response. For details of the CTRL/STAT register see *The Control/Status Register, CTRL/STAT* on page 6-10.

For more information about the parity check bits in the SW-DP protocol, see *Parity in the SWD protocol* on page 5-4.

If a debugger receives a FAULT response from the SW-DP, it must read the CTRL/STAT register and check the sticky flag values.

The WDATAERR flag is cleared to 0 by writing 1 to the WDERRCLR field of the AP Abort Register, see *The AP Abort Register, ABORT* on page 6-6.

## 3.2 Pushed compare and pushed verify operations

Debug Ports support operations where the value written as an AP transaction is used at the DP level to compare against a target read:

- the debugger writes a value as an AP transaction
- the DP performs a read from the AP
- the DP compares the two values and updates the Sticky Compare flag, *STICKYCMP*, in the DP Control/Status register, based on the result of the comparison:
  - pushed compare sets *STICKYCMP* to 1 if the values match
  - pushed verify sets *STICKYCMP* to 1 if the values do not match.

Whenever the *STICKYCMP* bit is set to 1 in this way, any outstanding transaction repeats are cancelled.

These operations are described as pushed operations. For more information see *The Control/Status Register, CTRL/STAT* on page 6-10.

Pushed operations are enabled using the Transfer Mode bits, *TRNMODE*, in the DP Control/Status Register, see *Transfer mode (TRNMODE), bits [3:2]* on page 6-14.

The DP includes a byte lane mask, so that the compare or verify operation can be restricted to particular bytes in the word. This mask is set using the *MASKLANE* bits in the Control/Status register. For more information about this masking see *MASKLANE and the bit masking of the pushed compare and pushed verify operations* on page 6-14.

Figure 3-1 on page 3-6 gives an overview of the pushed operations.

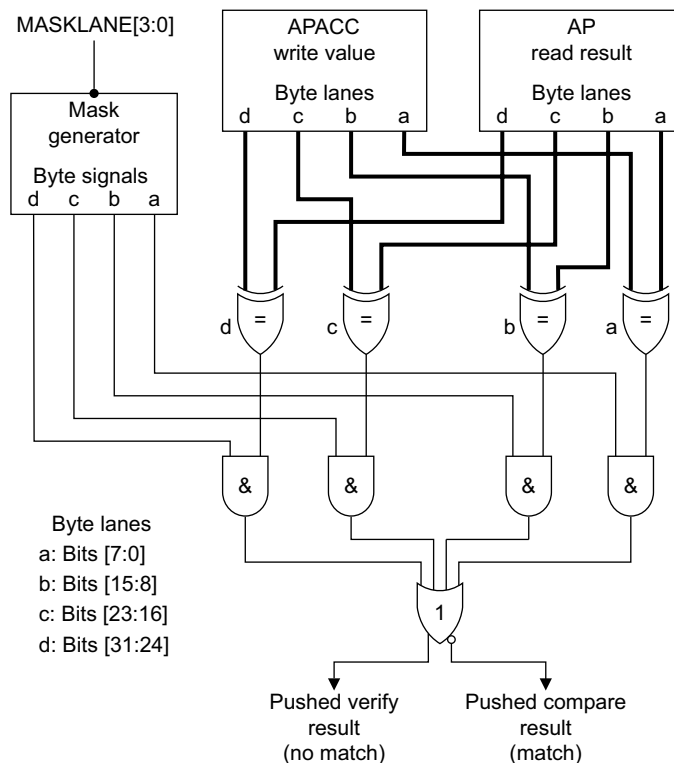


Figure 3-1 Pushed operations overview

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.

Considering pushed operations on a specific AP makes it easier to understand how these operations are implemented. On a *Memory Access Port* (MEM-AP), if you perform an AP write transaction to the Data Read/Write (DRW) Register with either pushed compare or pushed verify active:

- The DP holds the data value from the AP write transaction in the pushed compare logic, see Figure 2-2 on page 2-9.
- The AP reads from the address indicated by the MEM-AP Transfer Address Register (TAR), see *The Transfer Address Register (TAR)* on page 11-7.
- The value returned by this read is compared with the value held in the pushed compare logic, and the STICKYCMP bit is set depending on the result. The comparison is masked as required by the MASKLANE bits. The logic used for this comparison is shown in Figure 3-1. For more information about masking see *MASKLANE and the bit masking of the pushed compare and pushed verify operations* on page 6-14.

From this description, notice that 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.

---

**Note**

---

- Performing an AP read transaction with pushed compare or pushed verify enabled causes UNPREDICTABLE behavior.
  - On a Serial Wire DP, this means that performing an AP read transaction with pushed compare or pushed verify active returns an UNPREDICTABLE value, and the read has UNPREDICTABLE side-effects, although the wire-level protocol remains coherent.
- 

### 3.2.1 Example uses of pushed verify and pushed find operation on a MEM-AP

You can use pushed verify 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 it sets the STICKYCMP bit in the CRL/STAT Register to 1. This operation is described in more detail in *Example use of pushed verify operation on a MEM-AP* on page 8-22.

You can use pushed find to search system memory for a given value. However, this feature is most useful when performed using the AP Transaction Counter, described in *The Transaction Counter* on page 3-8. Again, this operation is described in more detail in Chapter 8 *The Memory Access Port (MEM-AP)*, in the section *Example using the Transaction Counter for a pushed find operation on a MEM-AP* on page 8-23.

### 3.3 The Transaction Counter

A Debug Port must include an AP Transaction Counter, TRNCNT. The Transaction Counter enables a debugger to make a single AP transaction request that generates a sequence of AP transactions. With a MEM-AP access, the AP transaction sequence might generate a sequence of accesses to the connected memory system.

Examples of the use of the Transaction Counter are:

- For a code download or memory fill operation. For memory fill, the Transaction Counter can be used to repeatedly write a single data value supplied in the initial AP transaction request. The MEM-AP includes a mechanism that auto-increments the access address after each AP access. This means that a series of AP accesses under the control of the transaction counter write the supplied data value to a sequence of memory addresses. For more information see *Packed transfers and the Transaction Counter* on page 8-19.
- With pushed compare or pushed verify operation enabled, the Transaction Counter can be used when reading from the DRW register, to perform a fast search or verify of an area of memory. This use is mentioned in *Example uses of pushed verify and pushed find operation on a MEM-AP* on page 3-7, and is described in more detail in *Example using the Transaction Counter for a pushed find operation on a MEM-AP* on page 8-23.

A field in the DP Control/Status Register, bits [23:12], maps onto the Transaction Counter. Writing a value other than zero to this field generates multiple AP transactions, see *The Control/Status Register, CTRL/STAT* on page 6-10. For example, writing 0x001 to this field generates two AP transactions, and writing 0x002 generates three transactions,.

The Transaction Counter does not auto-reload when it reaches zero.

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

- the Transaction Counter is zero
- the Sticky Error flag, in the Control/Status Register, is set to 1
- the Sticky Compare flag, in the Control/Status Register, is set to 1.

If a sequence of operations is terminated because the Sticky Error or Sticky Compare flag was set to 1, the Transaction Counter remains at the value from the last successful transaction. This means that software can recover the location of the error, or determine where the compare or verify operation terminated.

## 3.4 System and Debug power and Debug reset control

The Debug Port provides:

- Four control bits for system and debug power control. The use of these is described in:
  - *The DAP power domains model*
  - *Power control requirements and operation* on page 3-10
  - *Emulation of power-down* on page 3-12
  - *Emulation of power control* on page 3-12.
- Two control bits for debug reset control, see *Debug Reset Control* on page 3-14.

These control bits are programmable by the debugger, and drive signals into the connected system. These signals are intended as hints or stimuli into existing power and reset controllers.

The Debug Interface does not replace the system power and reset controllers, and the Debug Interface specification does not place any requirements on the operation of the system power and reset controllers. However, this section provides a model of how these signals might be used.

### ———— Note ————

The ARM Debug Interface v5 does not provide support for system reset control.

### 3.4.1 The DAP power domains model

The DAP model supports multiple power domains. These provide support for debug components that can be powered off.

Three power domains are modelled:

#### **Always-on power domain**

This must be powered on for the debugger to connect to the device.

#### **System power domain**

This includes all the system components.

#### **Debug power domain**

This includes all of the debug subsystem.

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

The DP registers reside in the Always-on power domain, on the external interface side of the DP. Therefore, they can always be driven, enabling power-up requests to be made to a system power controller. The power and reset control bits are part of the DP Control/Status register, see *The Control/Status Register, CTRL/STAT* on page 6-10. See *Debug Reset Control* on page 3-14 for more information about the reset control bits in this register. Four bits of the register provide power control signals:

**Bit [28], CDBGPWRUPREQ**

**CDBGPWRUPREQ** is the signal from the debug interface to the power controller, used to request the system power controller to fully power-up and enable clocks in the debug power domain.

**Bit [29], CDBGPWRUPACK**

**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.

**Bit [30], CSYSPWRUPREQ**

**CSYSPWRUPREQ** is the signal from the debug interface to the power controller, used to request the system power controller to fully power-up and enable clocks in the system power domain.

**Bit [31], CSYSPWRUPACK**

**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.

In most situations, debuggers power-up the complete SoC. However, if a debugger is being used to investigate an energy management issue, it might want to power-up only the debug domain. To enable this possibility, 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 on.

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 core to be powered down and debugged while maintaining power to the DP.

### 3.4.2 Power control requirements and operation

The following description applies to both:

- system domain power up, using the **CSYSPWRUPREQ** and **CSYSPWRUPACK** signals
- debug domain power up, using the **CDBGPWRUPREQ** and **CDBGPWRUPACK** signals.

Therefore, in the description:

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



The rules for the operation of power-up requests and acknowledgements are:

- To initiate power-on, **CxxxPWRUPREQ** must be asserted HIGH by the Debug Port.
  - If the corresponding power domain is powered down or in a low-power retention state, the power controller must power up and restore clocks to the domain when it detects **CxxxPWRUPREQ** asserted HIGH. When the domain is powered up, the controller must assert **CxxxPWRUPACK** HIGH.
  - If the corresponding power domain is already powered up and being clocked when the power controller detects **CxxxPWRUPREQ** asserted HIGH, the controller must still respond with **CxxxPWRUPACK** HIGH. However, there is no effect on the powered-up domain.
- Tools can only initiate a DAP transfer when both **CxxxPWRUPREQ** and **CxxxPWRUPACK** are asserted HIGH for one of the pairs of power control signals. When both **CxxxPWRUPREQ** and **CxxxPWRUPACK** are asserted HIGH, the corresponding power domain is powered on.
- The removal of power to a domain is requested by the Debug Port deasserting **CxxxPWRUPREQ**, that is, by the DP taking **CxxxPWRUPREQ** LOW.  
 The power controller deasserts **CxxxPWRUPACK** when it has accepted the request to power-down the domain.  
**CxxxPWRUPACK** being taken LOW by the power controller does not indicate that the domain has been powered down, it only indicates that the power controller has recognized the request to remove power. In effect, **CxxxPWRUPACK** acts as a logical AND of the corresponding **CxxxPWRUPREQ** and **CACTIVE** signals.
- **CxxxPWRUPACK** must default to the LOW state, and only go HIGH on receipt of a **CxxxPWRUPREQ** request.
- On detecting the deassertion of **CxxxPWRUPREQ**, indicated by the signal going LOW, 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 power-down has been requested, by the deassertion of **CxxxPWRUPREQ**, tools must wait until **CxxxPWRUPACK** is LOW before making a new request for power-up. **CxxxPWRUPACK** going LOW indicates that the power controller has recognized the original power-down request.  
 This requirement ensures that the power control handshaking mechanism is not violated.

#### ————— Note —————

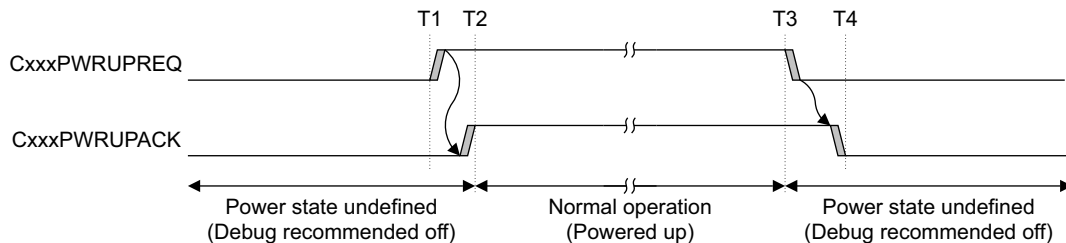
The AMBA v3 AXI low-power clock control signals **CSYSREQ** and **CSYSACK** have similar behavior to the ARM Debug Interface (ADI) **CSYSPWRUPREQ** and **CSYSPWRUPACK** signals. However:

- the AXI signals request entry to a low-power state, and acknowledge the request
- the ADI signals request full-power state, and acknowledge the request.

Figure 3-2 on page 3-12 shows the timing of the power control signals.

**Note**

All AP transactions must be initiated between times T2 and T3 for **CDBGPWRUPREQ** and **CDBGPWRUPACK**, as shown in Figure 3-2.



**Figure 3-2 Power-up request and acknowledgement timing**

### 3.4.3 Emulation of power-down

When **CxxxPRWUPREQ** from the ADI is asserted HIGH for a domain, if the power controller receives a conflicting request for the domain from another source it must *emulate* the power-down request for the domain. This applies if the power controller receives, from the other source, either of:

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

This requirement makes it possible to debug a system where one domain powers up and down dynamically. An example of a system that requires this is an IEM-enabled ARM core.

During emulation of the power-down request, the power controller carries out all of the expected power control handshaking, but does not actually remove power from the domain.

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

### 3.4.4 Emulation of power control

Where the system to which an ARM Debug Interface is connected does not support the ADI power control model, the required signals must be emulated or generated from other signals. Three possible cases are described here.

### System without power controller support for the ADI control scheme

If the connected system can not support the ADI power-up and power-down control scheme, then **CxxxPRWUPACK** must be connected to **CxxxPRWUPREQ**. With these connections, whenever the ADI requests power-up, or removes a power-up request, it receives the appropriate acknowledge immediately. This power control emulation is shown in Figure 3-3.

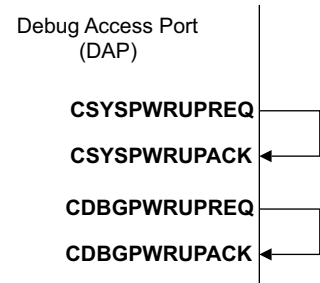


Figure 3-3 Emulation of power-up control

### System power controller does acknowledge power-up requests

If the system power controller does not provide ACK responses to power-up requests, but supplies an **ACTIVE** signal when a domain is powered-up, the **CxxxPRWUPACK** signals must be generated. This is shown in Figure 3-4.

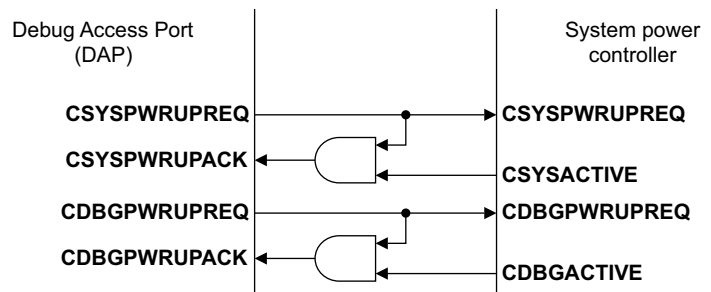


Figure 3-4 Generation of ACK signals from REQ and ACITVE signals

### System power controller does not support separate power domains

If the system power controller does not support separate debug and system power domains, then the two **CxxxPRWUPREQ** power-up request signals must be ORed together to provide a single power-up request to the controller. However, the **CxxxPRWUPACK** signals must be generated, so that the DAP sees the correct response to asserting a **CxxxPRWUPREQ** signal. This is shown in Figure 3-5 on page 3-14.

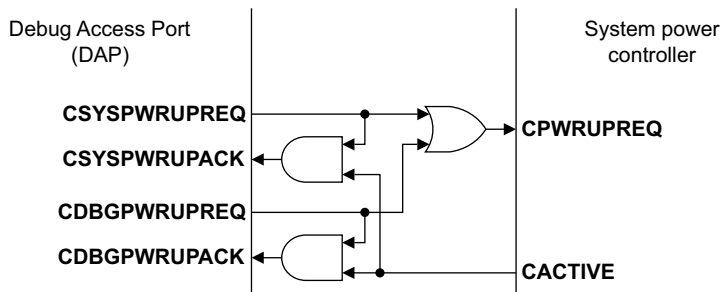


Figure 3-5 Signal generation for a single power domain

### 3.4.5 Debug Reset Control

The DP Control/Status register provides two bits, bits [27:26], for reset control of the debug domain, see *The Control/Status Register, CTRL/STAT* on page 6-10. The debug domain controlled by these signals covers the internal DAP and the connection between the DAP and the debug components, for example the debug bus. The two bits provide a debug reset request, **CDBGIRSTREQ**, and a reset acknowledge, **CDBGIRSTACK**. The associated signals provide a connection to a system reset controller.

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 3-6 shows the reset request and acknowledge timing.

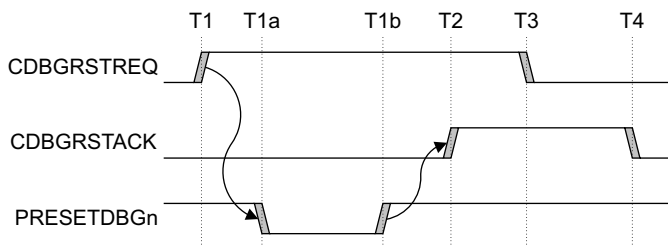


Figure 3-6 Reset request and acknowledge timing

In Figure 3-6:

1. At time T1, the debugger writes to the CDBGIRSTREQ bit, bit [26], of the DP Control/Status Register. This initiates the reset request.

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

#### ———— Note ————

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

2. At time T2, the system reset controller acknowledges that the reset of the debug domain has completed. The **CDBGIRSTACK** signal sets the corresponding bit, bit [27], in the DP Control/Status Register.
3. At time T3, the debugger checks the DP Control/Status Register and finds that the reset has completed. Therefore, it writes to the Control/Status register, to clear the CDBGIRSTREQ bit to 0. This removes the reset request signal.
4. At time T4, the system reset controller recognizes that **CDBGIRSTREQ** is no longer asserted, and deasserts **CDBGIRSTACK**.

———— **Caution** ————

If **CDBGIRSTREQ** is removed before the reset controller asserts **CDBGIRSTACK**, the behavior is UNPREDICTABLE.

—————

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

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



# Chapter 4

## The JTAG Debug Port (JTAG-DP)

This chapter describes the implementation of the *JTAG Debug Port* (JTAG-DP), and in particular the *Debug Test Access Port* (DBGTAP) *State Machine* (DBGTAPSM) and Scan Chains. It is only relevant to an ARM Debug Interface implementation that use a JTAG Debug Port. In this case, the JTAG-DP provides the external connection to the ADI, and all interface accesses are made using the scan chains, driven by the DBGTAPSM.

Additional information about JTAG-DPs is given in the following chapters:

- Chapter 3 *Common Debug Port (DP) features*
- Chapter 6 *Debug Port Registers*.

This chapter contains the following sections:

- *The Debug TAP State Machine introduction* on page 4-2
- *The scan chain interface* on page 4-3
- *IR scan chain and IR instructions* on page 4-8
- *DR scan chain and DR registers* on page 4-12.

## 4.1 The Debug TAP State Machine introduction

The *Debug TAP State Machine* (DBGTAPSM) controls the operation of a JTAG-DP. In particular, it controls the scan chain interface that provides the external physical interface to the ADI 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*. This chapter describes both the DBGTAPSM and its scan chain interface.

Figure 2-2 on page 2-9 shows an ARM Debug Interface with a JTAG-DP, including the basic operation of the scan chain interface.

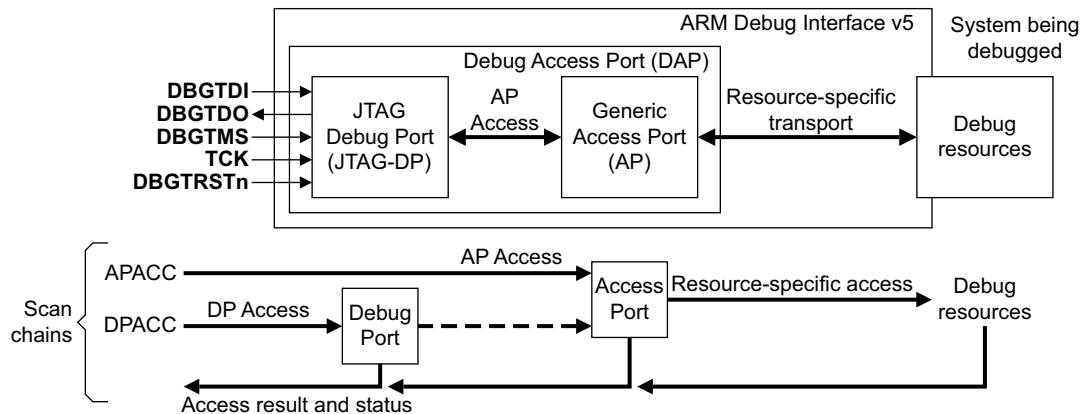


## 4.2 The scan chain interface

When a Debug Access Port (DAP) is implemented with a JTAG-DP, the wire-level interface is through scan chains, and the DAP comprises:

- a Debug TAP State Machine (DBGTAPSM)
- an Instruction Register (IR) and associated IR scan chain, used to control the behavior of the JTAG-DP and the currently-selected data register
- a number of Data Registers (DRs) and associated DR scan chains, that interface to:
  - the registers in the DAP
  - the debug registers in the device or debug component being accessed through the ADI.

Figure 2-2 on page 2-9 shows how the scan chains provide access to the different levels of the DAP architecture, and this is summarized in Figure 4-1.



**Figure 4-1 Mapping of the JTAG-DP scan chains onto the logical levels of the ARM Debug Interface**

### 4.2.1 Physical connection to the JTAG-DP

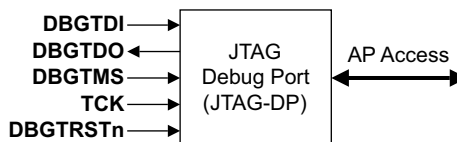
The physical connection to the JTAG-DP is closely based on the JTAG model. The connections are listed in Table 4-1. This table also shows the names of the equivalent signals in a JTAG implementation.

**Table 4-1 JTAG-DP signal connections**

JTAG-DP signal name	JTAG equivalent signal name	Direction	Required?	Description
<b>DBGTDI</b>	<b>TDI</b>	Input	Yes	Debug Data In
<b>DBGTDO</b>	<b>TDO</b>	Output	Yes	Debug Data Out
<b>TCK</b>	<b>TCK</b>	Input	Yes	Debug Clock
<b>DBGTMS</b>	<b>TMS</b>	Input	Yes	Debug Mode Select
<b>DBGTRSTn</b>	<b>TRST*</b>	Input	Optional	Debug TAP Reset

An implementation might also include a return clock signal, **RTCK**. However, ARM Limited recommends that **RTCK** is not implemented on an ARM Debug Interface.

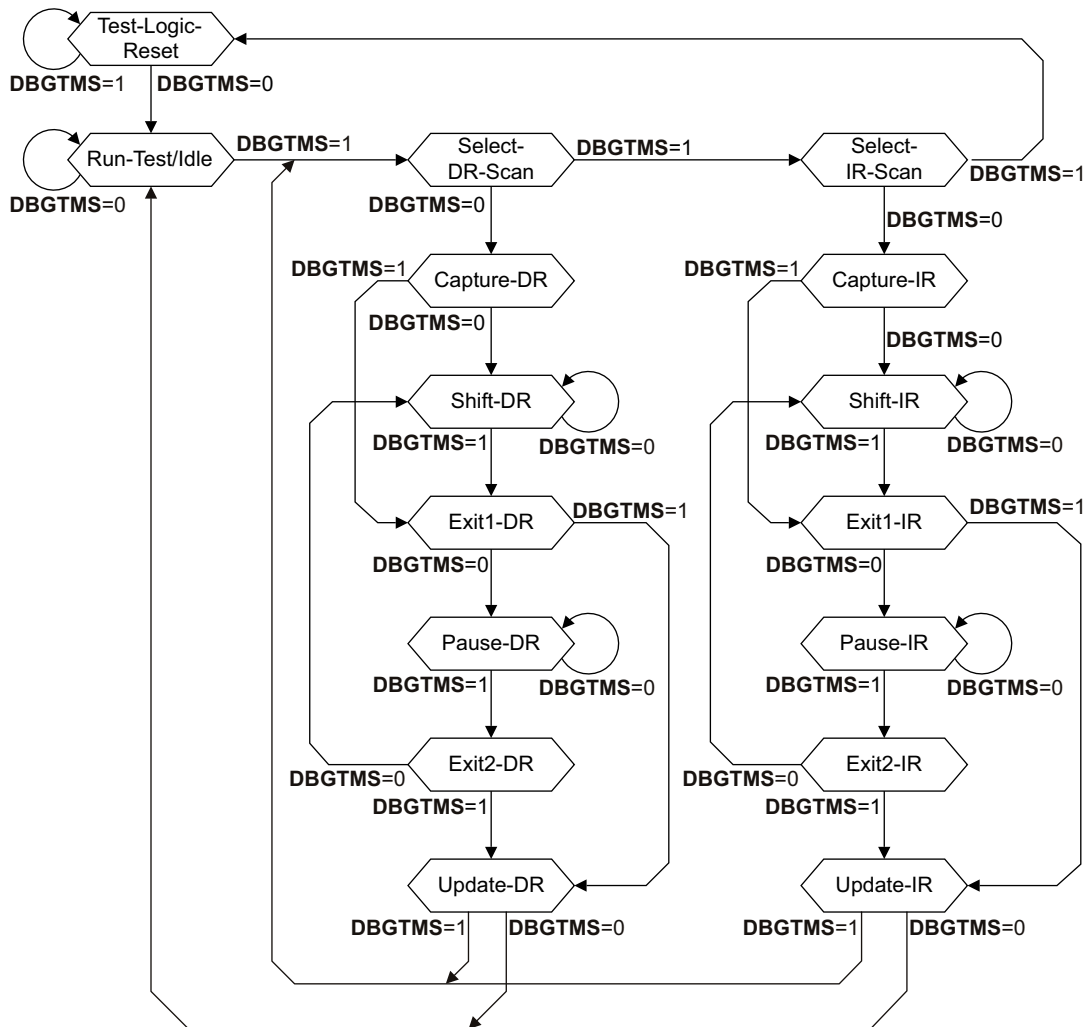
The recommended physical connection to the JTAG-DP is shown in Figure 4-2.



**Figure 4-2 JTAG-DP physical connection**

## 4.2.2 The Debug TAP State Machine (DBGTAPSM)

Figure 4-3 shows 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 4-3 The Debug TAP State Machine (DBGTAPSM)**

### 4.2.3 Basic operation of the DBGTAPSM

The **DBGTDI** signal into the DAP is the start of the scan chain, and the **DBGTDO** signal out of the DAP is the end of the scan chain.

Referring to the Debug TAP State Machine (DBGTAPSM) shown in Figure 4-3 on page 4-5:

- When the DBGTAPSM goes through the Capture-IR state, a value is transferred onto the Instruction Register (IR) scan chain. The IR scan chain is connected between **DBGTDI** and **DBGTDO**.
- While the DBGTAPSM is in the Shift-IR state, and for the transition from Capture-IR to Shift-IR, the IR scan chain advances one bit for each tick of **TCK**. This means that on the first tick:
  - the LSB of the IR is output on **DBGTDO**
  - bit [1] of the IR is transferred to bit [0]
  - bit [2] is transferred to bit [1]
  - .
  - .
  - .
  - the value on **DBGTDI** is transferred to the MSB of the IR.
- When the DBGTAPSM goes through the Update-IR state, the value scanned into the scan chain is transferred into the Instruction Register.
- When the DBGTAPSM goes through the Capture-DR state, a value is transferred from one of a number of Data Registers (DRs) onto one of a number of DR scan chains, connected between **DBGTDI** and **DBGTDO**.

The value held in the Instruction Register determines which Data Register, and associated DR scan chain, is selected.

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 scanned into the scan chain is transferred into the Data Register
- When the DBGTAPSM is in the Run-Test/Idle state, no special actions occur. Debuggers can use this as a true resting state.

———— **Note** ————

This is a change from the behavior of previous versions of the ARM Debug Interface based on the IEEE JTAG standard. From ARM Debug Interface v5 there is no requirement for debuggers to gate **TCK** to obtain a true rest state.

The behavior of the IR and DR scan chains is described in more detail in *IR scan chain and IR instructions* on page 4-8 and *DR scan chain and DR registers* on page 4-12.

The **DBGTRSTn** signal only resets the state machine. **DBGTRSTn** asynchronously takes the DBGTAPSM to the Test-Logic-Reset state. As shown in Figure 4-3 on page 4-5, 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 might take the state machine through one of the Update states, with the resulting side effects.

Within the DAP:

- the DP registers are only reset on a power-on reset
- the AP registers are reset on a power-on reset, and also by the Debug Reset Control described in *Debug Reset Control* on page 3-14.

## 4.3 IR scan chain and IR instructions

This section describes the JTAG-DP Instruction Register (IR), accessed through the IR scan chain.

### 4.3.1 The JTAG-DP Instruction Register (IR)

**Purpose** Holds the current DAP Controller instruction.

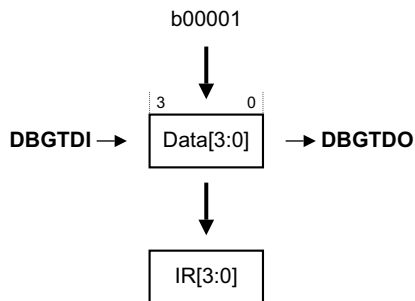
**Length** 4 bits.

#### Operating mode

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 b0001 is loaded into this shift section. This is shifted out, least significant bit first, during Shift-IR. As this happens, 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.

On debug logic reset, IDCODE becomes the current instruction, see *The JTAG-DP Device ID Code Register (IDCODE)* on page 4-13. Debug logic reset is initiated by the **DBGTRSTn** signal.

**Order** Figure 4-4 shows the operation of the Instruction Register.



**Figure 4-4 JTAG-DP Instruction Register operation**

This register is mandatory in the IEEE 1149.1 standard.

### 4.3.2 Required Instruction Register (IR) instructions

The description of the JTAG-DP Instruction Register shows how a 4-bit instruction is transferred into the IR. This instruction determines the physical Data Register that the JTAG-DP Data Register maps onto, as described in *DR scan chain and DR registers* on page 4-12. The standard IR instructions are listed in Table 4-2, and recommended IMPLEMENTATION DEFINED extensions to this instruction set are described in *implementation defined extensions to the IR instruction set* on page 4-10.

Unused IR instruction values are Reserved and select the Bypass register, described in *The JTAG-DP Bypass Register (BYPASS)* on page 4-12.

**Table 4-2 Standard IR instructions**

IR instruction value	Data register	DR scan length	Instruction and reference for description
b0xxx	-	-	<i>implementation defined extensions to the IR instruction set</i> on page 4-10
b1000	ABORT	35	<i>The JTAG-DP Abort Register (ABORT)</i> on page 4-21
b1001	-	-	Reserved
b1010	DPACC	35	<i>The JTAG-DP DP and AP Access Registers (DPACC and APACC)</i> on page 4-14
b1011	APACC	35	
b110x	-	-	Reserved
b1110	IDCODE	32	<i>The JTAG-DP Device ID Code Register (IDCODE)</i> on page 4-13
b1111	BYPASS	1	<i>The JTAG-DP Bypass Register (BYPASS)</i> on page 4-12

### 4.3.3 IMPLEMENTATION DEFINED extensions to the IR instruction set

The eight IR instructions b0000 to b0111 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. The instructions required to do this are listed in Table 4-3. All these instructions select the boundary scan data register.

#### ————— **Note** —————

ARM Limited recommends that:

- separate JTAG TAPs are used for boundary scan and debug
- the instructions listed in Table 4-3 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 4-3 Recommended implementation defined IR instructions for IEEE 1149.1-compliance**

IR instruction value	Instruction	Required by IEEE 1149.1?
b0000	EXTEST	See note in main text.
b0001	SAMPLE	Yes
b0010	PRELOAD	Yes
b0011	Reserved	-
b0100	INTEST <sup>a</sup>	No
b0101	CLAMP <sup>a</sup>	No
b0110	HIGHZ <sup>a</sup>	No
b0111	CLAMPZ <sup>a</sup>	No. See <i>The CLAMPZ instruction</i> on page 4-11.

a. Reserved, if the instruction is not implemented.

If you require a boundary scan implementation then you must implement the instructions that are shown as required by IEEE 1149.1. The other IR instruction values listed in Table 4-3 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, mandates that instruction b0000 is EXTEST. However, in the more recent edition, 1149.1-2001, this requirement is removed and it is recommended that instruction b0000 is Reserved. See the IEEE specification for more details.

---

The IEEE 1149.1 specification also requires the IDCODE and BYPASS instructions. These are included in Table 4-2 on page 4-9.

**The CLAMPZ instruction**

CLAMPZ is not an IEEE 1149.1 instruction.

If implemented, when the CLAMPZ instruction is selected all the 3-state outputs are placed in their inactive state, but the data supplied to the outputs is derived from the scan cells.

CLAMPZ can be implemented to ensure that, during production test, each output can be disabled when its value is 0 or 1. This encoding must be used for CLAMPZ if this function is required.

## 4.4 DR scan chain and DR registers

There are five physical DR registers:

- the BYPASS and IDCODE Registers, as defined by the IEEE 1149.1 standard
- the DPACC and APACC Access Registers
- an ABORT register, used to abort a transaction.

There is a scan chain associated with each of these registers. As described in *IR scan chain and IR instructions* on page 4-8, the value in the IR Register determines which of these scan chains is connected to the **DBGTDI** and **DBGTDO** signals.

### 4.4.1 The JTAG-DP Bypass Register (BYPASS)

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

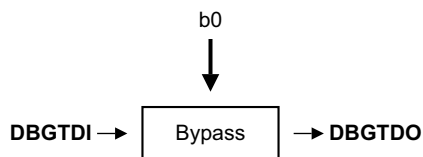
**Length** 1 bit.

#### Operating mode

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
- Nothing happens at the Update-DR state. The shifted-in data is ignored.

**Order** Figure 4-5 shows the operation of the Bypass Register.



**Figure 4-5 JTAG-DP Bypass Register operation**

This register is mandatory in the IEEE 1149.1 standard.

#### 4.4.2 The JTAG-DP Device ID Code Register (IDCODE)

**Purpose** Device identification. The Device ID Code value enables a debugger to identify the Debug Port to which it is connected. Different Debug Ports have different Device ID Codes, so that a debugger can distinguish between them.

This is the JTAG-DP implementation of the Identification Code Register, see *The Identification Code Register, IDCODE* on page 6-8.

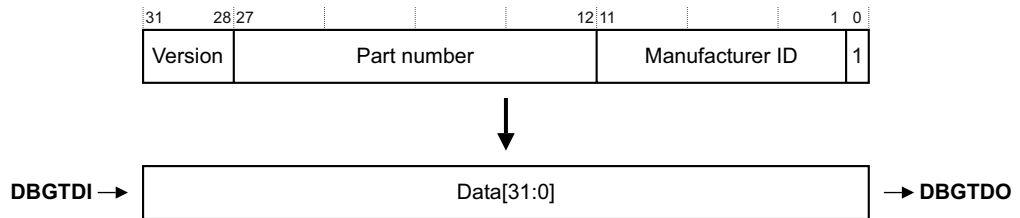
**Length** 32 bits.

**Operating mode**

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.

**Order** Figure 4-6 shows the operation of the Device ID Code Register.



**Figure 4-6 JTAG-DP Device ID Code Register operation**

### 4.4.3 The JTAG-DP DP and AP Access Registers (DPACC and APACC)

The DPACC and APACC scan chains have the same format.

**Purpose** Initiate a DP or AP access, to access a DP or AP register. The DPACC and APACC are used for read and write accesses to registers.

The DPACC scan chain is used to access the CTRL/STAT, SELECT and RDBUFF DP registers, see *JTAG-DP register map* on page 6-3.

The APACC scan chain is used to access registers in the currently-selected AP register bank, see:

- *Summary of MEM-AP registers* on page 11-2 for details of accessing MEM-AP registers
- *Summary of JTAG-AP registers* on page 12-2 for details of accessing JTAG-AP registers.

**Length** 35 bits.

#### Operating mode

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. Two ACK responses are implemented, and these are summarized in Table 4-4.

**Table 4-4 DPACC and APACC ACK responses**

Response	ACK[2:0] encoding	See:
OK/FAULT	b010	<i>The OK/FAULT response to a DPACC or APACC access</i> on page 4-15
WAIT	b001	<i>The WAIT response to a DPACC or APACC access</i> on page 4-16

All other ACK encodings are Reserved.

- In the Shift-DR state, this data is shifted out, least significant bit first. As shown in Figure 4-7 on page 4-15, the first three bits of data shifted out are ACK[2:0]. As the returned data is shifted out to **DBGTDO**, new data is shifted in from **DBGTDI**. This is described in *The OK/FAULT response to a DPACC or APACC access* on page 4-15.
- Operation in the Update-DR depends on whether the ACK[2:0] response was OK/FAULT or WAIT. The two cases are described in:
  - *Update-DR operation following an OK/FAULT response* on page 4-15
  - *Update-DR operation following a WAIT response* on page 4-17.

**Order** Figure 4-7 on page 4-15 shows the operation of the DP and AP Access Registers.

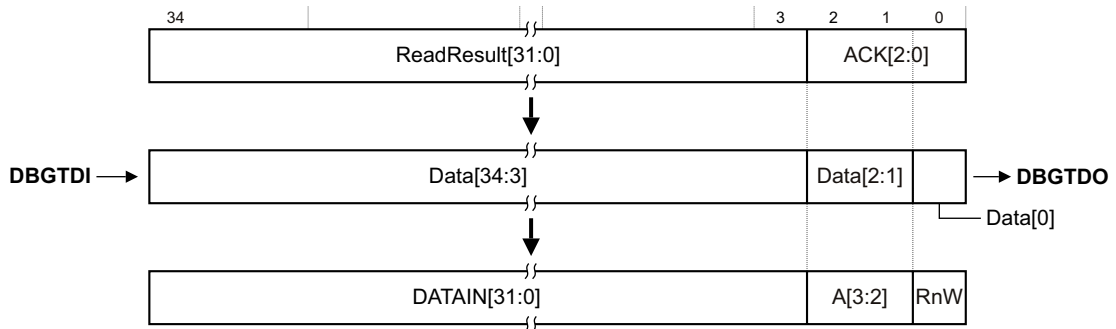


Figure 4-7 Operation of JTAG-DP DP Access and AP Access Registers

### The 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 was faulted. You must read the CTRL/STAT register to find whether the transaction was successful, see *The Control/Status Register, CTRL/STAT* on page 6-10:

- If the previous transaction was a read that completed successfully, then 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, then the captured ReadResult[31:0] is UNPREDICTABLE, and if Data[34:3] is shifted out it must be discarded.

### Update-DR operation following an OK/FAULT response

The values shifted into the scan chain form a request to read or write a register:

- if the current IR instruction is DPACC then **DBGTDI** and **DBGTDO** connect to the DPACC scan chain, and the request is to read or write a DP register
- if the current IR instruction is APACC then **DBGTDI** and **DBGTDO** connect to the APACC scan chain, and the request is to read or write an AP register.

In either case:

- If RnW is shifted in as 0, the request is to write the value in DATAIN[31:0] to the addressed register.
- If RnW is shifted in as 1, the request is to read the value of the addressed register. The value in DATAIN[31:0] is ignored. You must read the scan chain again to obtain the value read from the register.

The required register is addressed:

- In the case of a DPACC access to read a DP register, by the value shifted into A[3:2]. See *JTAG-DP register map* on page 6-3 for the addressing details.

- In the case of an APACC access to read an AP register, by the combination of:
  - the value shifted into A[3:2]
  - the current value of the SELECT register in the DP, see *The AP Select Register, SELECT* on page 6-15.

For more details of the register addressing, see:

- Table 11-2 on page 11-3, if you are accessing a MEM-AP register
- Table 12-2 on page 12-3, if you are accessing a JTAG-AP register.

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 is benign. That is, it has no effect on the operation of the DBGTAPSM, see *The Read Buffer, RDBUFF* on page 6-17. The section *Target response summary* on page 4-18 gives more information about how one DPACC or APACC scan returns the result from the previous scan.

If the current IR instruction is APACC, causing an APACC access:

- If any sticky flag is set to 1 in the DP CTRL/STAT Register, the transaction is discarded. The next scan returns an OK/FAULT response. For more information see *Sticky flags and DP error responses* on page 3-2, and *The Control/Status Register, CTRL/STAT* on page 6-10.
- If pushed compare or pushed verify operations are enabled then the scanned-in value of RnW must be 0, otherwise behavior is UNPREDICTABLE. On Update-DR, a read request is issued, and the returned value compared against DATAIN[31:0]. The STICKYCMP flag in the DP CTRL/STAT register is updated based on this comparison. For more information see *Pushed compare and pushed verify operations* on page 3-5.

Pushed operations are enabled using the TRNMODE field of the DP CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 6-10 for more information.

- The AP access does not complete until the AP signals it as completed. For example, if you access a Memory Access Port (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.

## The 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. Accesses to the DP are stalled, by returning WAIT, until any previous AP transaction has completed.

Normally, if software detects a WAIT response, it retries the same transfer. This enables the protocol to process data as quickly as possible. However, if the software has retried a transfer a number of 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 signals to the active AP that it should terminate the transfer it is currently attempting, to permit access to other parts of the debug system. An AP might not be able to terminate a transfer on its ASIC interface. However, on receiving an ABORT, the AP must free its interface to the DP.

### ***Update-DR operation following a WAIT response***

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

### **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. When this happens, if the Overrun Detect flag is set to 1, the Sticky Overrun flag, STICKYORUN, is set to 1. See *The Control/Status Register, CTRL/STAT* on page 6-10 for more information about the Overrun Detect and Sticky Overrun flags.

As long as the previous transaction remains 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 set to 1, and to clear the flag to 0 after gathering any required information about the overrun condition. See *Overrun detection* on page 3-3 for more information.

### **Minimum response times**

As explained in *The OK/FAULT response to a DPACC or APACC access* on page 4-15, 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. However, the second access generates a WAIT response if the requested register access has not completed.

The DBGTTAPSM clock, **TCK**, is asynchronous to the internal clock of the system being debugged, and the time required for an access to complete includes clock cycles in both domains. However, the timing between the Update-DR state and the Capture-DR state only includes **TCK** cycles. Referring to Figure 4-3 on page 4-5, 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
- a path through Run-Test/Idle and Select-DR-Scan.

If the second path is followed, the state machine can spend any number of **TCK** cycles spinning in the Run-Test/Idle state. This means it is possible to vary 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 Limited recommends that debuggers should be able to adapt to any IMPLEMENTATION DEFINED limit.

In addition, when accessing AP registers, or accessing a connected device through an AP, there might be other variable response delays in the system. A debugger that can adapt to these delays, avoiding wasted WAIT scans, operates more efficiently and provides higher maximum data throughput.

### Target response summary

As described in *The OK/FAULT response to a DPACC or APACC access* on page 4-15, 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 4-5 on page 4-19, for cases where the previous scan was a DPACC access
- Table 4-6 on page 4-20, for cases where the previous scan was an APACC access

#### ————— **Note** —————

The target responses shown in Table 4-5 on page 4-19 are independent of the operation being performed in the current DPACC or APACC scan. In this table, *Read result* is the data shifted out as Data[34:3], and ACK is decoded from the data shifted out as Data[2:0].



Table 4-5 JTAG-DP target response summary, when previous scan<sup>a</sup> was a DPACC access

Previous scan <sup>a</sup> , at Update-DR state			Current scan, at Capture-DR state			Notes
R/W	A[3:2] <sup>b</sup>	Sticky? <sup>c</sup>	AP state <sup>d</sup>	Read result	ACK	
X	bXX	X	Busy	UNP <sup>e</sup>	WAIT	Might cause Sticky Overrun flag to be set to 1. <sup>f</sup>
R	b00	X	Not Busy	UNP <sup>e</sup>	OK/ FAULT	Returns UNPREDICTABLE value.
	b01			CTRL/STAT		Returns CTRL/STAT value.
	b10			SELECT		Returns SELECT value.
	b11			0x00000000		Returns RDBUFF value, always zero.
W	b00	X	Not Busy	UNP <sup>e</sup>	OK/ FAULT	Behavior UNPREDICTABLE.
	b01					Value has been written to CTRL/STAT.
	b10					Value has been written to SELECT.
	b11					Writes to RDBUFF always ignored.

- The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates made following a WAIT response are discarded.
- A[3:2] in the DPACC access.
- The *Sticky?* column indicates whether any Sticky flag was set to 1 in the DP CTRL/STAT register, see *The Control/Status Register; CTRL/STAT* on page 6-10.
- The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.
- UNP = UNPREDICTABLE.
- If the Overrun Detect flag is set to 1 then this access and response sequence causes the Sticky Overrun flag to be set to 1. See *The Control/Status Register; CTRL/STAT* on page 6-10.

Table 4-6 JTAG-DP target response summary, when previous scan<sup>a</sup> was an APACC access

Previous scan <sup>a</sup> , at Update-DR state			Current scan, at Capture-DR state			Notes
R/W	A[3:2] <sup>b</sup>	Sticky? <sup>c</sup>	AP state <sup>d</sup>	Read result	ACK	
X	bXX	X	Busy	UNP <sup>e</sup>	WAIT	Might cause Sticky Overrun flag to be set to 1. <sup>f</sup>
R	bXX	No	Ready	See Notes	OK/ FAULT	See footnote <sup>g</sup> .
			Error	UNP <sup>e</sup>		Sticky Error flag is set to 1.
W	bXX	No	Ready	UNP <sup>e</sup>	OK/ FAULT	See footnote <sup>h</sup> .
			Error	UNP <sup>e</sup>		Sticky Error flag is set to 1.
X	bXX	Yes	X	UNP <sup>e</sup>	OK/ FAULT	Previous transaction was discarded.

- The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates made following a WAIT response are discarded.
- A[3:2] in the APACC access.
- The *Sticky?* column indicates whether any Sticky flag was set to 1 in the DP CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 6-10.
- The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.
- UNP = UNPREDICTABLE.
- If the Overrun Detect flag is set to 1 then this access and response sequence causes the Sticky Overrun flag to be set to 1. See *The Control/Status Register, CTRL/STAT* on page 6-10.
- If Pushed Verify or Pushed Compare is enabled, the behavior is UNPREDICTABLE. Otherwise, returns the value of the AP Register addressed on the previous scan.
- If Pushed Verify or Pushed Compare is enabled, the previous transaction performed the required pushed operation, that might have set the Sticky Compare flag to 1, see *Pushed compare and pushed verify operations* on page 3-5. Otherwise, the data captured at the previous scan has been written to the AP register requested.

## Host response summary

The ACK column, for the *Current scan*, at *Capture-DR* state section of Table 4-5 on page 4-19 and Table 4-6 on page 4-20, shows the responses the host might receive after initiating a DPACC or APACC access. Table 4-7 indicates the normal action of a host in response to each of these ACKs.

**Table 4-7 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 AP Abort Register to enable access to the AP.
Read or Write	Invalid ACK	Assume a target or line error has occurred and treat as a fatal error.

### 4.4.4 The JTAG-DP Abort Register (ABORT)

**Purpose** Access the AP Abort Register in the DP, to force an AP abort.  
This is the JTAG-DP implementation of the AP Abort Register, see *The AP Abort Register, ABORT* on page 6-6.

**Length** 35 bits.

#### Operating mode

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 is used to access the AP Abort Register.

The debugger must scan the value 0x00000008 into this scan chain. This value:

- writes the RnW bit as 0
- writes the A[3:2] field as b00
- writes 1 into bit [0], the DAPABORT bit, of the AP Abort Register.

#### Caution

The effect of writing any other value into this scan chain is UNPREDICTABLE.

**Order** Figure 4-8 on page 4-22 shows the operation of the ABORT scan chain.

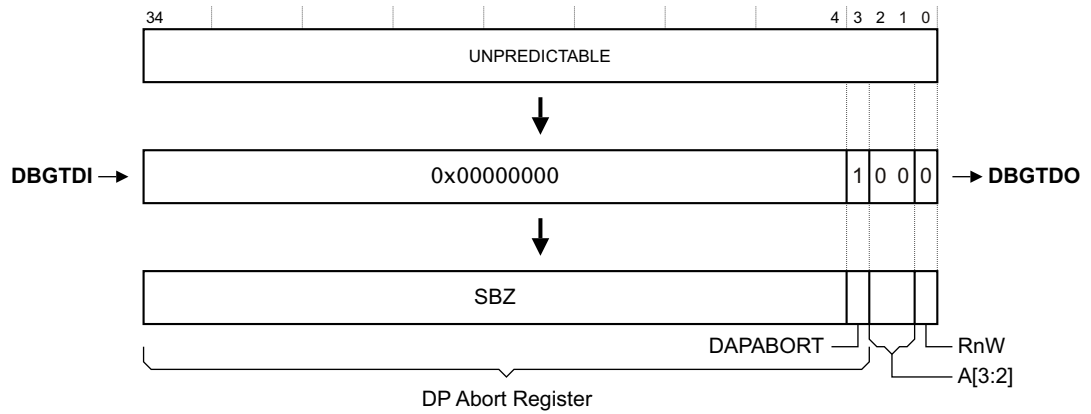


Figure 4-8 JTAG-DP ABORT scan chain operation

# Chapter 5

## The Serial Wire Debug Port (SW-DP)

This chapter describes the implementation of the *Serial Wire Debug Port* (SW-DP), including the DAP Serial Wire Debug interface. It is only relevant if your ARM Debug Interface implementation uses a SW-DP. In this case, the SW-DP provides the external connection to the Debug Interface, and all interface accesses are made using the Serial Wire Debug protocol that is summarized in this chapter.

Additional information about SW-DPs is given in the following chapters:

- Chapter 3 *Common Debug Port (DP) features*
- Chapter 6 *Debug Port Registers*.

This chapter contains the following sections:

- *Introduction to the DAP Serial Wire Debug Port* on page 5-2
- *Introduction to the ARM Serial Wire Debug (SWD) protocol* on page 5-3
- *Serial Wire Debug protocol operation* on page 5-5
- *Protocol description* on page 5-10.

## 5.1 Introduction to the DAP Serial Wire Debug Port

This chapter only gives an architectural description of the ARM *Serial Wire Debug Port* (SW-DP), and in particular the Serial Wire Debug interface that provides the physical connection to an ARM Debug Interface. It describes:

- the *Serial Wire Debug* (SWD) protocol
- how this protocol provides access to the DP registers
- how the SW-DP provides *Access Port ACCesses* (APACCs).

You can implement an ARM Serial Wire Debug interface with either a synchronous or an asynchronous serial connection. However, this chapter does not describe the physical characteristics of the SWD interface, such as:

- asynchronous clocking, if required
- multiple connections, if required
- signal sampling rates.

---

### Note

Contact ARM Limited if you require more detailed information about the implementation of the ARM Serial Wire Debug Interface.

---

### 5.1.1 Operational requirement for synchronous SW-DP serial interface

If a SW-DP is implemented with a synchronous serial interface, the host must continue to clock the interface for a number of cycles after the data phase of any data transfer. This ensures that the transfer can be clocked through the SW-DP. This means that after the data phase of any transfer the host must do one of the following:

- immediately start a new SW-DP operation
- continue to clock the SW-DP serial interface until the host starts a new SW-DP operation
- after clocking out the data parity bit, continue to clock the SW-DP serial interface until it has clocked out at least 8 more clock rising edges, before stopping the clock.

The protocol used for the SW-DP operations is described in the remainder of this chapter.

## 5.2 Introduction to the ARM Serial Wire Debug (SWD) protocol

The ARM Serial Wire Debug Interface uses a single bi-directional data connection. It is IMPLEMENTATION DEFINED whether the serial interface:

- transfers data asynchronously, for minimum pin count
- provides a separate clock connection, and transfers data synchronously.

Each sequence of operations 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 ORUNDETECT flag is set to 1 in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 6-10.

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 Overrun Detect bit in the DP CTRL/STAT Register is set to 1, then a data transfer phase is required on all responses, including WAIT and FAULT. For more information, see *Sticky overrun behavior* on page 5-12.

For details of the CTRL/STAT Register see *The Control/Status Register, CTRL/STAT* on page 6-10.

---

## 5.2.1 Parity in the SWD protocol

In the SWD protocol, a simple parity check is applied to all packet request and data transfer phases. Even parity is used:

### Packet requests

The parity check is made over the APnDP, RnW and A[2:3] bits. If, of these four bits:

- the number of bits set to 1 is odd, then the parity bit is set to 1
- the number of bits set to 1 is even, then the parity bit is set to 0.

### Data transfers (WDATA and RDATA)

The parity check is made over the 32 data bits, WDATA[0:31] or RDATA[0:31]. If, of these 32 bits:

- the number of bits set to 1 is odd, then the parity bit is set to 1
- the number of bits set to 1 is even, then the parity bit is set to 0.

The packet request parity bit is shown in each of the diagrams in this section, from Figure 5-1 on page 5-7 to Figure 5-7 on page 5-13. It appears on the wire immediately after the A[2:3] bits.

The WDATA parity bit is shown in Figure 5-1 on page 5-7 and in Figure 5-7 on page 5-13. It appears on the wire immediately after the WDATA[31] bit.

The RDATA parity bit is shown in Figure 5-2 on page 5-8 and in Figure 5-6 on page 5-13. It appears on the wire immediately after the RDATA[31] bit.

### ————— Note —————

The ACK[0:2] bits are never included in the parity calculation. Debuggers must remember this when parity checking the data from a read operation, when the debugger receives a continuous stream of 36 bits, as shown in Figure 5-2 on page 5-8:

- bits 0 to 2 are ACK[0:2]
- bits 3 to 34 are RDATA[0:31]
- bit 35 is the parity bit.

The parity check must be applied to bits 3 to 34 of this block of data, and the result compared with bit 35, the parity bit.

---



## 5.3 Serial Wire Debug protocol operation

This section gives an overview of the bi-directional operation of the protocol. It illustrates each of the possible sequences of operations on the Serial Wire Debug interface data connection.

The illustrations in this section, and the descriptions in *Protocol description* on page 5-10, describe a synchronous implementation of the Serial Wire Debug interface. An asynchronous implementation has additional park, resume and idle states at turnaround.

The sequences of operations illustrated here are:

- *Successful write operation (OK response)* on page 5-7
- *Successful read operation (OK response)* on page 5-7
- *WAIT response to Read or Write operation request* on page 5-8
- *FAULT response to Read or Write operation request* on page 5-9
- *Protocol error sequence* on page 5-9.

The terms used in the illustrations are described in *Key to illustrations of operations*.

### 5.3.1 Key to illustrations of operations

The illustrations of the different possible operations use the following terms:

<b>Start</b>	A single start bit, with value 1.
<b>APnDP</b>	A single bit, indicating whether the Debug Port or the Access Port Access Register is to be accessed. This bit is 0 for an DPACC access, or 1 for a APACC access.
<b>RnW</b>	A single bit, indicating whether the access is a read or a write. This bit is 0 for an write access, or 1 for a read access.
<b>A[2:3]</b>	Two bits, giving the A[3:2] address field for the DP or AP Register Address: <ul style="list-style-type: none"> <li>• For a DPACC access, the A[3:2] value is the address of the register in the SW-DP register map, see <i>SW-DP Register Map</i> on page 6-5.</li> <li>• For an APACC access, the register being addressed depends on the A[3:2] value and the value held in the SELECT register. For details of the addressing see:               <ul style="list-style-type: none"> <li>— Table 11-2 on page 11-3 for accesses to a MEM-AP register</li> <li>— Table 12-2 on page 12-3 for accesses to a JTAG-AP register.</li> </ul>               For details of the SELECT register see <i>The AP Select Register, SELECT</i> on page 6-15.             </li> </ul>
<hr/> <p style="text-align: center;"><b>Note</b></p> <hr/> <p>The A[3:2] value is transmitted Least Significant Bit (LSB) first on the wire. This is why it appears as A[2:3] on the diagrams.</p> <hr/>	
<b>Parity</b>	A single parity bit for the preceding packet. See <i>Parity in the SWD protocol</i> on page 5-4.

<b>Stop</b>	A single stop bit. In the synchronous SWD protocol this is always 0.
<b>Park</b>	A single bit. The host does not drive the line for this bit, and the line is pulled HIGH by the SWD interface hardware. The target reads this bit as 1.
<b>Trn</b>	Turnaround. This is a period during which the line is not driven and the state of the line is undefined. The length of the turnaround period is controlled by the TURNROUND field in the Wire Control Register, see <i>The Wire Control Register, WCR (SW-DP only)</i> on page 6-18. The default setting is a turnaround period of one clock cycle.

---

**Note**

All the examples given in this chapter show the default turnaround period of one cycle.

---

There are additional turnaround periods in the asynchronous SWD protocol.

<b>ACK[0:2]</b>	A three-bit target-to-host response.
-----------------	--------------------------------------

---

**Note**

The ACK value is transmitted LSB-first on the wire. This is why it appears as ACK[0:2] on the diagrams.

---

**WDATA[0:31]**

32 bits of write data, from host to target.

---

**Note**

The WDATA value is transmitted LSB-first on the wire. This is why it 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. This is why it appears as RDATA[0:31] on the diagrams.

---

**Note**

The diagrams in this section are included to show the operation of the Serial Wire Debug protocol. They are not timing diagrams for the protocol. Contact ARM Limited if you require more information about timings of the serial connection to the SW-DP.

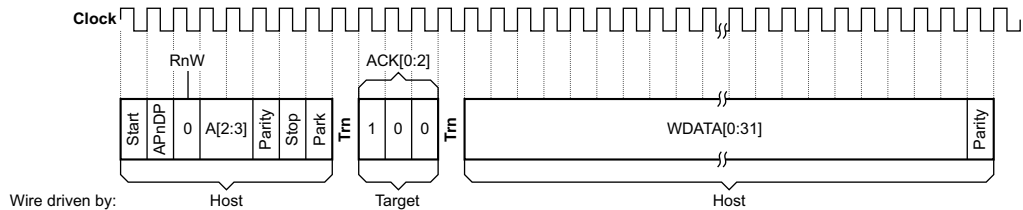
### 5.3.2 Successful write operation (OK response)

A successful write operation consists of three phases:

- an eight-bit write packet request, from the host to the target
- a three-bit OK acknowledge response, from the target to the host
- a 33-bit data write phase, from the host to the target.

By default, there are single-cycle turnaround periods between each of these phases. See the description of **T<sub>rn</sub>** in *Key to illustrations of operations* on page 5-5 for more information.

A successful write operation is shown in Figure 5-1.



**Figure 5-1 Serial Wire Debug successful write operation**

#### Note

- The OK response shown in Figure 5-1 only 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.
- There is no turnaround phase after the data phase. The host is driving the line, and can start the next operation immediately.

### 5.3.3 Successful read operation (OK response)

A successful read operation consists of three phases:

- an eight-bit read packet request, from the host to the target
- a three-bit OK acknowledge response, from the target to the host
- a 33-bit data read 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 the description of **T<sub>rn</sub>** in *Key to illustrations of operations* on page 5-5 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.

A successful read operation is shown in Figure 5-2 on page 5-8.

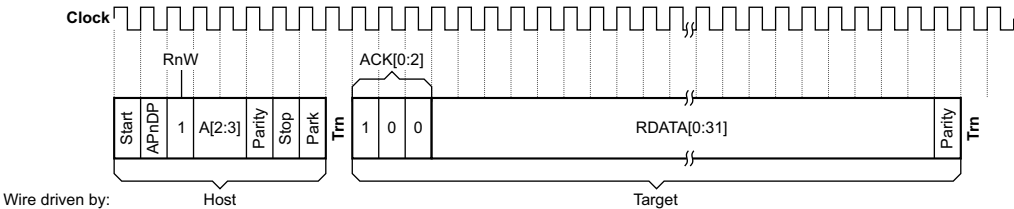


Figure 5-2 Serial Wire Debug successful read operation

### 5.3.4 WAIT response to Read or Write operation request

A WAIT response to a read or write packet request consists of two phases:

- an eight-bit read or write packet request, from the host to the target
- a three-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 the description of **Trn** in *Key to illustrations of operations* on page 5-5 for more information.

A WAIT response to a read or write packet request is shown in Figure 5-3.

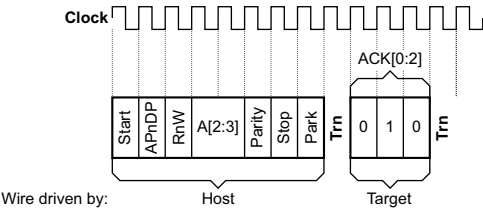


Figure 5-3 Serial Wire Debug WAIT response to a packet request

#### Note

If Overrun Detection is enabled then a data phase is required on a WAIT response. For more information see *Sticky overrun behavior* on page 5-12.

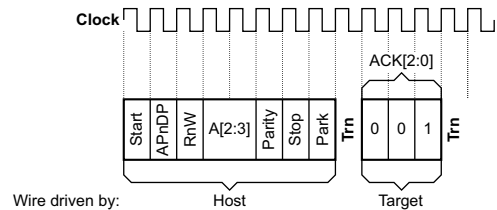
### 5.3.5 FAULT response to Read or Write operation request

A FAULT response to a read or write packet request consists of two phases:

- an eight-bit read or write packet request, from the host to the target
- a three-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 the description of **Trn** in *Key to illustrations of operations* on page 5-5 for more information.

A FAULT response to a read or write packet request is shown in Figure 5-4.



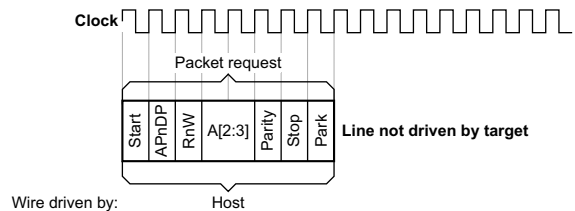
**Figure 5-4 Serial Wire Debug FAULT response to a packet request**

#### Note

If Overrun Detection is enabled then a data phase is required on a FAULT response. For more information see *Sticky overrun behavior* on page 5-12.

### 5.3.6 Protocol error sequence

A protocol error occurs when a host issues a packet request but the target fails to return any acknowledge response. This is illustrated in Figure 5-5.



**Figure 5-5 Serial Wire Debug protocol error after a packet request**

## 5.4 Protocol description

This section provides additional information on the DAP Serial Wire Debug operations that are introduced in *Serial Wire Debug protocol operation* on page 5-5.

### 5.4.1 Connection and line reset sequence

The serial interface to the SW-DP must use a connection sequence, to ensure that hot-plugging the serial connection does not result in unintentional transfers. The connection sequence ensures that the SW-DP is synchronized correctly to the header that is used to signal a connection. It consists of a sequence of 50 clock cycles with data = 1, that is, with the serial data signal asserted HIGH by the debugger.

This connection sequence is also used as a line reset sequence, see *Protocol Error responses* on page 5-13. The protocol requires that any run of 50 consecutive 1s on the data input is detected as a line reset, regardless of the state of the protocol.

After the host has transmitted a line request sequence to the SW-DP, it must read the IDCODE register. The SW-DP returns an OK response to this read. For more information see:

- *The Identification Code Register, IDCODE* on page 6-8
- *Successful read operation (OK response)* on page 5-7.

The requirement that the host reads the IDCODE register to exit the training state gives confirmation that correct packet frame alignment has been achieved.

### 5.4.2 The OK response

When the SW-DP receives a packet request from the debug host, it *must* respond immediately. If the SW-DP is ready for the data phase of the transfer, and there is no error condition, it issues an OK response. This is indicated by an acknowledge phase of b001.

#### ———— Note ————

- As shown in *Serial Wire Debug protocol operation* on page 5-5, there is always a turnaround between the end of the packet request from the host and the start of the acknowledgement from the SW-DP target. In the synchronous SWD protocol the default turnaround is exactly one serial clock cycle, but see the description of **Trn** in *Key to illustrations of operations* on page 5-5 for more information.  
  
There is a turnaround whenever there is a change in the direction of data transfer over the serial SWD connection. If an operation that is described as immediate involves a change in the data transfer direction then the operation must start immediately after the turnaround.
- All SWD transfers are made LSB-first. Therefore, the OK response of b001 appears on the wire as 1, followed by 0, followed by 0, as shown in Figure 5-1 on page 5-7 and Figure 5-2 on page 5-8.

If the host requested a write access it must start the write transfer immediately after receiving the OK acknowledgement from the target. This behavior is the same whether the write is to the DP or to an AP. However, the SW-DP can buffer AP writes, as described in *SW-DP write buffering* on page 5-14.

If the host requested a read access to the DP then the SW-DP sends the read data immediately after the acknowledgement. Because there is no change in the data transfer direction between the acknowledgement and the read data there is not any turnaround between these phases. This is shown in Figure 5-2 on page 5-8.

Read accesses to the AP are *posted*. This means that the result of the access is returned on the *next* transfer. If the next access you have to make is not another AP read then you must insert a read of the DP RDBUFF Register to obtain the posted result, see *The Read Buffer, RDBUFF* on page 6-17.

When you have to make a series of AP reads you only have to insert one read of the RDBUFF Register:

- On the first AP read access, the read data returned is UNPREDICTABLE. You must discard this result.
- If you immediately make another AP read access this returns the result of the previous AP read.
- You can repeat this for any number of AP reads.
- Issuing the last AP read packet request returns the last-but-one AP read result.
- You must then read the DP RDBUFF Register to obtain the last AP read result.

## Operation and use of the READOK flag and RESEND register

On a SW-DP, the DP CTRL/STAT register includes a READOK flag, bit [6]. This register is described in *The Control/Status Register, CTRL/STAT* on page 6-10.

The READOK flag is updated on every AP read access, and on every RDBUFF read request. The flag is updated when the packet header is decoded, and reflects the ACK[2:0] value returned:

- on an OK response, ACK[2:0] = b001, the READOK bit is set to 1
- on a WAIT or FAULT response the READOK bit is set to 0.

DP register accesses other than RDBUFF reads do not affect the READOK flag.

### ————— Note —————

As stated earlier in this section, the ACK[2:0] value appears LSB-first on the physical connection, and for this reason this value is shown as ACK[0:2] on the protocol diagrams.

This means that if a host receives a corrupted ACK response to an AP or RDBUFF read request it can check whether the read actually completed correctly. The host reads the DP CTRL/STAT Register to find the value of the READOK flag:

- If the flag is set to 1 then the read was performed correctly. The host can use a RESEND request to obtain the read result, see *The Read Resend Register, RESEND (SW-DP only)* on page 6-21.
- If the flag is set to 0 then the read was not successful. The host must retry the original AP or RDBUFF read request.

### 5.4.3 The WAIT response

If the SW-DP is not able to process the request from the debugger immediately it must issue a WAIT response. However, a WAIT response must not be issued to the following requests. The DP must always process these three requests immediately:

- a read of the IDCODE register, see *The Identification Code Register, IDCODE* on page 6-8
- a read of the CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 6-10
- a write to the ABORT register, see *The AP Abort Register, ABORT* on page 6-6.

With any request other than those listed, the DP issues a WAIT response, with no data phase, if it cannot process the request. This happens:

- if a previous AP or DP access is outstanding
- if the new request is an AP read request and the result of the previous AP read is not yet available.

---

#### Note

When overrun detection is enabled a WAIT response must include a data phase. See *Sticky overrun behavior* for more information.

---

Normally, when a debugger receives a WAIT response it retries the same operation. This enables it 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, if appropriate, the debugger might write to the ABORT register. This signals to the active AP that it must terminate the transfer that it is currently attempting. An AP implementation might be unable to terminate a transfer on its ASIC interface. However, on receiving an 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.

### 5.4.4 The FAULT response

A SW-DP must not issue a FAULT response to an access to the IDCODE, CTRL/STAT or ABORT registers. For any other access, the DP issues a FAULT response if any sticky flag is set to 1 in the CTRL/STAT Register. See *Sticky overrun behavior* for more information about the sticky overrun flag, and see *The Control/Status Register, CTRL/STAT* on page 6-10 for a description of the register.

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. see *Overrun detection* on page 3-3.

In a SW-DP, the sticky error flags are cleared to 0 by writing bits in the ABORT register, see *The AP Abort Register, ABORT* on page 6-6.

### 5.4.5 Sticky overrun behavior

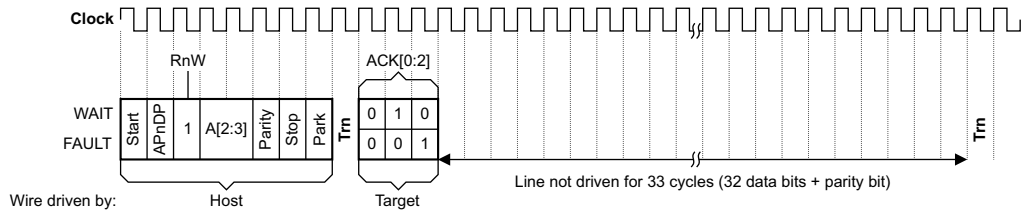
If a SW-DP receives a transaction request when the previous transaction has not completed it returns a WAIT response. If overrun detection is enabled in the CTRL/STAT Register, the STICKYORUN flag is set to 1 in that register. For more information see *The Control/Status Register, CTRL/STAT* on page 6-10. Subsequent transactions generate FAULT responses, because a sticky flag is set to 1.



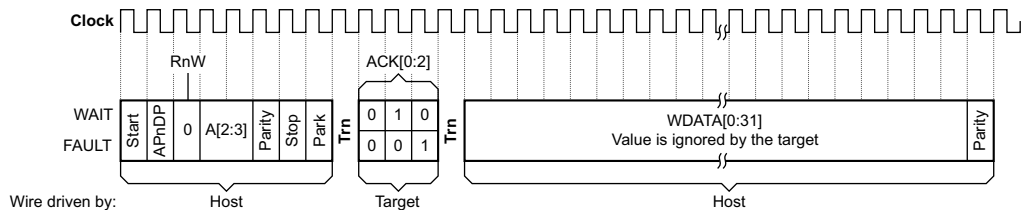
When overrun detection is enabled, WAIT and FAULT responses require a data phase:

- If the transaction is a read the data in the data phase is UNPREDICTABLE. 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 5-6 shows the WAIT or FAULT response to a read operation when overrun detection is enabled, and Figure 5-7 shows the response to a write operation when overrun detection is enabled.



**Figure 5-6 Serial Wire WAIT or FAULT response to a read operation when overrun detection is enabled**



**Figure 5-7 Serial Wire WAIT or FAULT response to a write operation when overrun detection is enabled**

#### 5.4.6 Protocol Error responses

If the SW-DP detects a parity error in the packet request it does not reply to the request. For more information about the parity checks in the SW-DP protocol see *Parity in the SWD protocol* on page 5-4.

When the host receives no reply to its request, it must back off, in case the DP has lost frame synchronization for some reason. After this, it can issue a new transfer request. In this situation it *must* read the IDCODE register, see *The Identification Code Register, IDCODE* on page 6-8. This is mandated by this specification because a successful read of the IDCODE register confirms that the target is operational.

If there is no response at the second attempt the debugger must force a line reset to ensure frame synchronization and valid operation. This is necessary because the DP is in a state where it will only respond to a line reset. After the line reset the debugger must read the IDCODE register before it attempts any other operations. For more information see *Connection and line reset sequence* on page 5-10.

If the transfer that resulted in the original protocol error response was a write you can assume that no write occurred. If the original transfer was a read it is possible that the read was issued to an AP. Although this is unlikely, you must consider this possibility because reads are pipelined and the DP might implement a write buffer.

———— **Note** ————

After a line reset, an asynchronous SW-DP connection requires retraining. Contact ARM Limited if you require more information about asynchronous SW-DP operation.

—————

### 5.4.7 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 it issues an OK response to a write request if it can accept the write into its write buffer. This 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. It 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 then, if it has any writes buffered it *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 then the WDATAERR flag is set to 1 in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 6-10. A buffered write is abandoned if:

- A sticky flag is set to 1 by a previous transaction.
- A DP read of the IDCODE or CTRL/STAT Register is made. Because the DP is not permitted to stall reads of these registers, it must:
  - perform the IDCODE or CTRL/STAT Register access immediately
  - discard any buffered writes, because otherwise they would be performed out-of-order
  - set the WDATAERR flag to 1.
- A DP write of the ABORT Register is made. Again, this is because the DP cannot stall an ABORT Register access.

This means that if you make a series of AP write transactions, it might not be possible to determine which transaction failed from examining the ACK responses. However it might be possible to use other enquiries to find which write failed. For example, if you are using the *auto-address increment* (AddrInc) feature of a Memory Access Port, then you can read the Transfer Address Register to find the address of the last successful write transaction. See *The Transfer Address Register (TAR)* on page 11-7 for more information.

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 IDCODE 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.

---

**Note**

---

If Pushed Verify or Pushed Compare is enabled, AP write transactions are converted into AP reads. These are then treated in the same way as other AP read operations. See *Pushed compare and pushed verify operations* on page 3-5 for details of these operations.

---

If you have to perform a DP read of the IDCODE or CTRL/STAT Register or a DP write to the ABORT Register immediately after a sequence of AP writes, you must first perform an access that the DP is able to stall. In this way you ensure that the write buffer is emptied before performing the DP register access. If this is not done, WDATAERR might be set to 1, and the buffered writes 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.

This means that in many cases the requirement for an additional instruction can be avoided.

---

### 5.4.8 Summary of target responses

Table 5-1 on page 5-16 summarizes the target SW-DP response to all possible debugger DP read operation requests.

Table 5-2 on page 5-17 summarizes the target SW-DP response to all possible debugger AP read operation requests.

Table 5-3 on page 5-17 summarizes the target SW-DP response to all possible debugger DP write operation requests, assuming the WDATA parity check is good.

Table 5-4 on page 5-18 summarizes the target SW-DP response to all possible debugger AP write operation requests, assuming the WDATA parity check is good.

Fault conditions that are not shown in these tables are described in *Fault conditions not included in the target response tables* on page 5-18

**Table 5-1 Target response summary for DP read transaction requests**

<b>A[3:2]</b>	<b>Sticky flag set to 1?</b>	<b>AP Ready?</b>	<b>SW-DP (target) response</b>	
			<b>ACK</b>	<b>Register addressed and action</b>
b00	X <sup>a</sup>	X <sup>a</sup>	OK	IDCODE. Respond with register value.
b01	X <sup>a</sup>	X <sup>a</sup>	OK	CRTL/STAT or WCR. Respond with register value <sup>b</sup> .
b10	No	Yes	OK	RESEND. Respond by resending the last read value sent to the host. This value is the result of one of: <ul style="list-style-type: none"> <li>the most recent AP read</li> <li>the most recent DP RDBUF read.</li> </ul>
b11	No	Yes	OK	RDBUFF. Respond with the value from the previous AP read, and set READOK flag in CTRL/STAT Register to 1.
b10	No	No	WAIT	RESEND. No data phase, unless overrun detection is enabled <sup>c</sup> .
b10	Yes	X	FAULT	RESEND. No data phase, unless overrun detection is enabled <sup>c</sup> .
b11	No	No	WAIT	RDBUFF. No data phase, unless overrun detection is enabled <sup>c</sup> . Set READOK flag in CTRL/STAT Register to 0.
b11	Yes	X	FAULT	RDBUFF. No data phase, unless overrun detection is enabled <sup>c</sup> . Set READOK flag in CTRL/STAT Register to 0.

- a. The SW-DP must always give an OK response to a read of the IDCODE, CTRL/STAT or WCR Register.
- b. Which value is returned depends on the value of the CTRLSEL bit in the SELECT Register, see *The AP Select Register, SELECT* on page 6-15.
- c. See *Sticky overrun behavior* on page 5-12 for details of data phase when overrun detection is enabled.

Table 5-2 Target response summary for AP read transaction requests

A[3:2]	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
			ACK	Action
bXX	No	Yes	OK	Normally <sup>a</sup> , return value from previous AP read <sup>b</sup> and set READOK flag in CTRL/STAT Register to 1. Initiate AP read of addressed register <sup>c</sup> .
bXX	No	No	WAIT	No data phase, unless overrun detection is enabled <sup>d</sup> . Set READOK flag in CTRL/STAT Register to 0.
bXX	Yes	X	FAULT	No data phase, unless overrun detection is enabled <sup>d</sup> . Set READOK flag in CTRL/STAT Register to 0.

- a. If Pushed Verify or Pushed Compare is enabled, behavior is UNPREDICTABLE.  
b. On the first of a sequence of AP reads, the value returned in the data phase is UNPREDICTABLE.  
c. The AP register is addressed by the value of A[3:2] together with the value of the APBANKSEL field in the SELECT Register in the DP, see *The AP Select Register, SELECT* on page 6-15.  
d. See *Sticky overrun behavior* on page 5-12 for details of data phase when overrun detection is enabled.

Table 5-3 Target response summary for DP write transaction requests

A[3:2]	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
			ACK	Action
b00	X	X	OK	Write WDATA value to ABORT Register.
Not b00	No	Yes <sup>a</sup>	OK	Write WDATA value to DP register indicated by A[3:2].
Not b00	No	No	WAIT	No data phase, unless overrun detection is enabled <sup>b</sup> .
Not b00	Yes	X	FAULT	No data phase, unless overrun detection is enabled <sup>b</sup> .

- a. Writes might be accepted when other transactions are still outstanding. These writes might be abandoned subsequently. See *SW-DP write buffering* on page 5-14 for more information.  
b. See *Sticky overrun behavior* on page 5-12 for details of data phase when overrun detection is enabled.

**Table 5-4 Target response summary for AP write transaction requests**

A[3:2]	Sticky flag set to 1?	SW-DP (target) response		
		AP Ready?	ACK	Action
bXX	No	Yes <sup>a</sup>	OK	Normally <sup>b</sup> , write WDATA value to the indicated AP register <sup>c</sup> .
bXX	No	No	WAIT	No data phase, unless overrun detection is enabled <sup>d</sup> .
bXX	Yes	X	FAULT	No data phase, unless overrun detection is enabled <sup>d</sup> .

- Writes might be accepted when other transactions are still outstanding. These writes might be abandoned subsequently. See *SW-DP write buffering* on page 5-14 for more information.
- If Pushed Verify or Pushed Compare is enabled, the write is converted to a read of the addressed AP register, and the value returned by this read is compared with the supplied WDATA value, see *Pushed compare and pushed verify operations* on page 3-5 for more information. For an outline of how AP registers are addressed see footnote <sup>c</sup> to this table.
- The AP register is addressed by the value of A[3:2] together with the value of the APBANKSEL field in the SELECT Register in the DP, see *The AP Select Register, SELECT* on page 6-15.
- See *Sticky overrun behavior* on page 5-12 for details of data phase when overrun detection is enabled.

### 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 5-1 on page 5-16 to Table 5-4:

#### Protocol fault

If there is a protocol fault in the operation request then the target does not respond to the request at all. This means that when the host expects an ACK response, it finds that the line is not driven.

#### 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 5-3 on page 5-17. When the parity check is performed and fails, the WDATAERR flag is set to 1 in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 6-10.

### 5.4.9 Summary of host responses

Every access by a debugger to a SW-DP starts with an operation request. *Summary of target responses* on page 5-15 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 a SW-DP, it expects to receive a 3-bit acknowledgement, as listed in the ACK columns of Table 5-1 on page 5-16 to Table 5-4 on page 5-18. Table 5-5 summarizes how the debugger must respond to this acknowledgement, for all possible cases.

**Table 5-5 Summary of host (debugger) responses to the SW-DP acknowledge**

Operation requested	ACK received	Host response	
		Data phase	Additional action
R	OK	Capture RDATA from target and check for valid parity <sup>a</sup> and protocol.	Might have to re-issue original read request or use the RESEND register if a parity or protocol fault occurs and are unable to flag data as invalid <sup>b</sup> .
W	OK	Send WDATA.	Validity of this transfer will be confirmed on next access.
X	WAIT	No data phase, unless overrun detection is enabled <sup>c</sup> .	Normally, repeat the original operation request. See <i>The WAIT response</i> on page 5-12 for more information.
X	FAULT	No data phase, unless overrun detection is enabled <sup>c</sup> .	Can send new headers, but only an access to DP register addresses b0X will give a valid response.
X	No ACK	Back off because of possible data phase.	Can attempt IDCODE Register read. Otherwise reset connection and retrain. See <i>Protocol Error responses</i> on page 5-13.
R	Invalid ACK	Back off because of possible data phase.	Can check CTRL/STAT Register to see if the response sent was OK.
W	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.

- See *Parity in the SWD protocol* on page 5-4 for details of the parity checking.
- The host debugger might support corrupted reads, or it might have to re-try the transfer.
- If overrun detection is enabled, a data phase is required. See *Sticky overrun behavior* on page 5-12 for a description of the behavior on read and write operations.





# Chapter 6

## Debug Port Registers

This chapter describes the *Debug Port* (DP) registers, for both the JTAG-DP and the SW-DP. More information about Debug Ports is given in the following chapters:

- Chapter 3 *Common Debug Port (DP) features*
- Chapter 4 *The JTAG Debug Port (JTAG-DP)*
- Chapter 5 *The Serial Wire Debug Port (SW-DP)*.

This chapter includes the following sections:

- *Debug Port registers overview* on page 6-2
- *Debug Port (DP) register descriptions* on page 6-6.

## 6.1 Debug Port registers overview

Every ARM Debug Interface includes a single Debug Port (DP). This is one of:

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

Although the two Debug Port implementations are different, their register sets are generally similar, and are described together in this chapter. In the section *Debug Port (DP) register descriptions* on page 6-6, the detailed register descriptions always make it clear how the register implementation differs for a JTAG-DP and for a SW-DP.

This section summarizes the Debug Port registers, and gives the register maps for a JTAG-DP and for a SW-DP. It contains the following sub-sections:

- *Debug Port (DP) registers summary*
- *JTAG-DP register map* on page 6-3
- *SW-DP Register Map* on page 6-5.

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

### 6.1.1 Debug Port (DP) registers summary

Table 6-1 summarizes the DP registers, and lists which registers are implemented on a JTAG-DP and which are implemented on a SW-DP.

**Table 6-1 Summary of Debug Port registers**

Name	Description	JTAG-DP	SW-DP	For description see section
ABORT	AP Abort Register	Yes	Yes	<i>The AP Abort Register, ABORT</i> on page 6-6
IDCODE	ID Code Register	Yes	Yes	<i>The Identification Code Register, IDCODE</i> on page 6-8
CTRL/STAT	DP Control/Status Register	Yes	Yes	<i>The Control/Status Register, CTRL/STAT</i> on page 6-10
SELECT	Select Register	Yes	Yes	<i>The AP Select Register, SELECT</i> on page 6-15
RDBUFF	Read Buffer	Yes	Yes	<i>The Read Buffer, RDBUFF</i> on page 6-17
WCR	Wire Control Register	No	Yes	<i>The Wire Control Register, WCR (SW-DP only)</i> on page 6-18

**Table 6-1 Summary of Debug Port registers (continued)**

Name	Description	JTAG-DP	SW-DP	For description see section
RESEND	Read Resend Register	No	Yes	<i>The Read Resend Register; RESEND (SW-DP only)</i> on page 6-21
ROUTESEL	Reserved	No	Optional	See footnote <sup>a</sup>

- a. The specification of the SW-DP provides for an optional ROUTESEL register. However the register is Reserved in the ARM Debug Interface v5.0 specification. Therefore, no other details of this register are given.

### 6.1.2 JTAG-DP register map

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* on page 6-4.

Table 6-2 shows the JTAG-DP register map.

**Table 6-2 JTAG-DP register map**

IR contents	Address <sup>a</sup>	Access	For description see:	Notes
IDCODE	_b	RO	<i>The Identification Code Register, IDCODE</i> on page 6-8	-
DPACC	0x0	RAZ/WI	-	Reserved. Read UNPREDICTABLE, writes UNPREDICTABLE. Hosts must not access this register.
DPACC	0x4	R/W	<i>The Control/Status Register, CTRL/STAT</i> on page 6-10	-
DPACC	0x8	R/W	<i>The AP Select Register, SELECT</i> on page 6-15	-
DPACC	0xC	RAZ/WI	<i>The Read Buffer, RDBUFF</i> on page 6-17	-

Table 6-2 JTAG-DP register map (continued)

IR contents	Address <sup>a</sup>	Access	For description see:	Notes
ABORT	0x0	WO <sup>c</sup>	<i>The AP Abort Register, ABORT on page 6-6</i>	-
ABORT	0x4 - 0xC	-	-	See footnote <sup>d</sup> .

- a. 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 b00.
- b. There is no address associated with IDCODE accesses. See *Accessing the JTAG-DP registers*.
- c. The value read on the ABORT scan chain is UNPREDICTABLE.
- d. The result of accessing the ABORT scan chain with the address field not set to 0x0 is UNPREDICTABLE.

### Accessing the JTAG-DP registers

The JTAG-DP registers are only accessed when the Instruction Register (IR) for the DAP access contains the IDCODE, DPACC or ABORT instruction. In detail, the register accesses for each instruction are:

- IDCODE**      The IDCODE scan chain has no address field, and accesses the IDCODE register.
- DPACC**        The DPACC scan chain accesses registers at addresses 0x0 to 0xC, although register address 0x0 is Reserved, and the RDBUFF register at 0xC is always Read As Zero on a JTAG-DP. These registers are shown in the illustration of the JTAG-DP in Figure 2-2 on page 2-9.
- ABORT**        For a write access with address 0x0, the ABORT scan chain accesses the AP 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 4 *The JTAG Debug Port (JTAG-DP)*.

### 6.1.3 SW-DP Register Map

For most register addresses on the SW-DP, different registers are addressed on read and write accesses. In addition, the CTRLSEL bit in the SELECT Register determines which register is accessed at address 0x04.

Table 6-3 shows the SW-DP register map.

**Table 6-3 SW-DP register map**

Address <sup>a</sup>	CTRLSEL <sup>b</sup>	Access <sup>c</sup>	Required?	For description see:
0x0	X	R	Yes	<i>The Identification Code Register, IDCODE</i> on page 6-8
		W	Yes	<i>The AP Abort Register, ABORT</i> on page 6-6
0x4	0	R/W	Yes	<i>The Control/Status Register, CTRL/STAT</i> on page 6-10
	1	R/W	No	<i>The Wire Control Register, WCR (SW-DP only)</i> on page 6-18
0x8	X	R	Yes	<i>The Read Resend Register, RESEND (SW-DP only)</i> on page 6-21
		W	Yes	<i>The AP Select Register, SELECT</i> on page 6-15
0xC	X	R	Yes	<i>The Read Buffer, RDBUFF</i> on page 6-17
		W	No	See footnote <sup>d</sup> .

- The A[3:2] field of the SW-DP request provides bits [3:2] of the address. Bits [1:0] of the address are always b00.
- CTRLSEL bit in the SELECT register, see *The AP Select Register, SELECT* on page 6-15.
- Entries in the Access column indicate whether the SWD protocol makes a read or a write access to the given address.
- The specification of the SW-DP provides for an optional ROUTESEL register. However the register is Reserved in the ARM Debug Interface v5.0 specification. Therefore, no other details of this register are given.

### 6.1.4 Accesses to Reserved addresses

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

- Figure 2-2 on page 2-9, for accesses to JTAG-DP registers
- Figure 8-1 on page 8-4, for accesses to MEM-AP registers
- Figure 9-1 on page 9-3, for accesses to JTAG-AP registers

There are a number of Reserved addresses in these register maps. With DAP accesses:

- reads of Reserved addresses return zero (RAZ)
- writes to Reserved addresses are ignored (WI).

## 6.2 Debug Port (DP) register descriptions

This section gives a detailed description of each of the DP registers. Each description clearly states whether the register is implemented for the JTAG-DP and for the SW-DP, and any differences in the implementation.

### 6.2.1 The AP Abort Register, ABORT

The AP Abort Register is always present on all DP implementations. Its main purpose is to force a DAP abort, and on a SW-DP it is also used to clear error and sticky flag conditions.

**JTAG-DP** It is at address 0x0 when the Instruction Register (IR) contains ABORT.

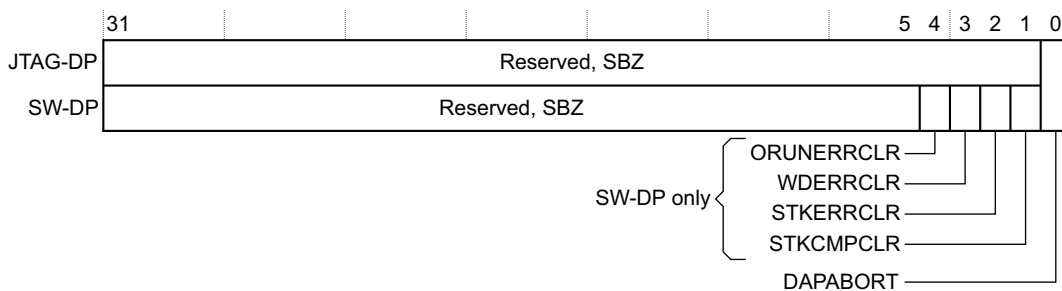
**SW-DP** It is at address 0x0 on write operations when the APnDP bit = 0. Access to the AP Abort Register is not affected by the value of the CTRLSEL bit in the Select Register.

It is:

- A write-only register.
- Always accessible, and returns an OK response if a valid transaction is received.

AP Abort Register accesses always complete on the first attempt.

Figure 6-1 shows the register bit assignments.



**Figure 6-1 AP Abort Register bit assignments**

Table 6-4 lists the bit functions of the AP Abort Register.

**Table 6-4 AP Abort Register bit assignments**

Bits	Function	Description
[31:5]	-	Reserved, SBZ.
[4] <sup>a</sup>	ORUNERRCLR <sup>a</sup>	Write 1 to this bit to clear the STICKYORUN overrun error flag <sup>b</sup> to 0.
[3] <sup>a</sup>	WDERRCLR <sup>a</sup>	Write 1 to this bit to clear the WDATAERR write data error flag <sup>b</sup> to 0.

**Table 6-4 AP Abort Register bit assignments (continued)**

<b>Bits</b>	<b>Function</b>	<b>Description</b>
[2] <sup>a</sup>	STKERRCLR <sup>a</sup>	Write 1 to this bit to clear the STICKYERR sticky error flag <sup>b</sup> to 0.
[1] <sup>a</sup>	STKCMPLCLR <sup>a</sup>	Write 1 to this bit to clear the STICKYCMP sticky compare flag <sup>b</sup> to 0.
[0]	DAPABORT	Write 1 to this bit to generate a DAP abort. This aborts the current AP transaction. Do this only if the debugger has received WAIT responses over an extended period.

a. Implemented on SW-DP only. On a JTAG-DP this bit is Reserved, SBZ.

b. In the Control/Status register, see *The Control/Status Register, CTRL/STAT* on page 6-10.

## DAP Aborts

Writing 1 to bit [0] of the AP Abort Register generates a DAP abort, causing the current AP transaction to abort. This also terminates the Transaction Counter, if it was active.

From a software perspective, this is a fatal operation. It discards any outstanding and pending transactions, and leaves the AP in an unknown state. However, on a SW-DP, the sticky error bits are not cleared to 0.

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.

After a DAP abort is requested, new transactions can be accepted by the DP. However, an AP access to the AP that was aborted might result in additional WAIT responses. Other APs can be accessed, although the state of the system might make it impossible to continue with debug.

### Caution

On a JTAG-DP, for the AP Abort Register:

- bit [0], DAPABORT, is the only bit that is defined
- the effect of writing any value other than 0x00000001 is UNPREDICTABLE.

## Clearing error and sticky compare flags to 0, SW-DP only

When a debugger, connected to a SW-DP, checks the Control/Status register and finds that an error flag is set to 1, or that the sticky compare flag is set to 1, it must write to the AP Abort Register to clear the error or sticky compare flag to 0. Table 6-4 on page 6-6 lists the flags that might be set to 1 in the Control/Status register, and shows which bit of the AP Abort Register is used to clear each of the flags to 0. You can use a single write of the AP Abort Register to clear multiple flags to 0, if this is necessary.

After clearing the flag to 0, you might have to access the DP and AP registers to find what caused the flag to be set to 1. Typically:

- For the STICKYCMP or STICKYERR flag, you must find which location was accessed to cause the flag to be set to 1.
- For the WDATAERR flag, you must resend the corrupted data.
- For the STICKYORUN flag, you must find which DP or AP transaction caused the overflow. You then have to repeat your transactions from that point.

## 6.2.2 The Identification Code Register, IDCODE

The Identification Code Register is always present on all DP implementations. It provides identification information about the ARM Debug Interface.

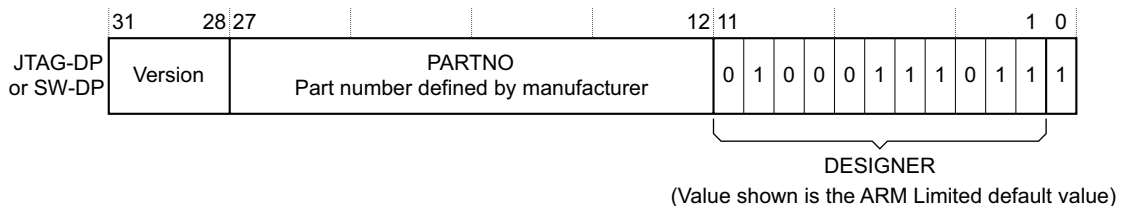
**JTAG-DP** It is accessed using its own scan chain, see *The JTAG-DP Device ID Code Register (IDCODE)* on page 4-13.

**SW-DP** It is at address 0x0 on read operations when the APnDP bit = 0. Access to the Identification Code Register is not affected by the value of the CTRLSEL bit in the Select Register.

It is:

- a read-only register
- always accessible.

Figure 6-2 shows the register bit assignments.



**Figure 6-2 Identification Code Register bit assignments**

Table 6-5 on page 6-9 lists the bit functions of the Identification Code Register.



**Table 6-5 Identification Code Register bit assignments**

Bits	Function	Description
[31:28]	Version	Version code. The meaning of this field is IMPLEMENTATION DEFINED.
[27:12]	PARTNO	Part Number for the DP. This value is provided by the designer of the Debug Port and <i>must not</i> be changed. Current DPs designed by ARM Limited have the following PARTNO values: <b>JTAG-DP</b> 0xBA00 <b>SW-DP</b> 0xBA10
[11:1]	DESIGNER	JEDEC Designer ID, an 11-bit JEDEC code that identifies the designer of the ADI implementation, see <i>The JEDEC Designer ID</i> . The ARM Limited default value for this field, shown in Figure 6-2 on page 6-8, is 0x23B. Designers can change the value of this field. However, if the DAP is used for boundary scan then this field <i>must</i> be set to the JEDEC Manufacturer ID assigned to the manufacturer.
[0]	-	Always 1.

## The JEDEC Designer ID

This field is bits [11:1] of the Identification Code Register. The JEDEC Designer ID is also described as the JEP-106 manufacturer identification code, and can be subdivided into two fields, as shown in Table 6-6.

**Table 6-6 JEDEC JEP-106 manufacturer ID code, with ARM Limited values**

JEP-106 field	Bits <sup>a</sup>	ARM Limited registered value
Continuation code	4 bits, [11:8]	b0100, 0x4
Identity code	7 bits, [7:1]	b0111011, 0x3B

a. Field width, in bits, and the corresponding bits in the Identification Code Register.

JEDEC codes are assigned by the JEDEC Solid State Technology Association, see *JEP106M, Standard Manufacture's Identification Code*.

Normally, this field identifies the designer of the ADI implementation, rather than the system architect or the device manufacturer. However, if the DAP is used for boundary scan then the field must be set to the JEDEC Manufacturer ID assigned to the manufacturer of the device.

### 6.2.3 The Control/Status Register, CTRL/STAT

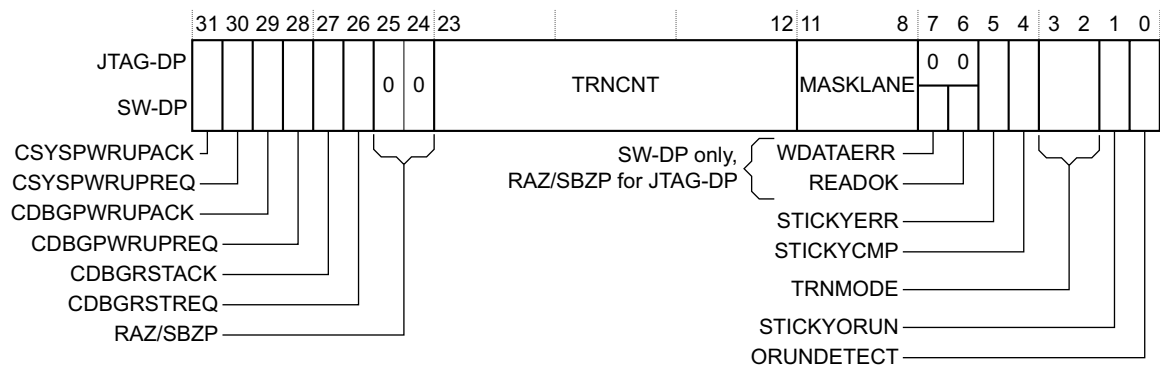
The Control/Status Register is always present on all DP implementations. Its provides control of the DP, and status information about the DP.

**JTAG-DP** It is at address 0x4 when the Instruction Register (IR) contains DPACC.

**SW-DP** It is at address 0x4 on read and write operations when the APnDP bit = 0 and the CTRLSEL bit in the Select Register is set to 0. For information about the CTRLSEL bit see *The AP Select Register, SELECT* on page 6-15.

It is a read-write register, although some bits are read-only. It is IMPLEMENTATION DEFINED whether some fields in the register are supported, and Table 6-7 shows which fields are required in all implementations.

Figure 6-3 shows the register bit assignments.



**Figure 6-3 Control/Status Register bit assignments**

Table 6-7 lists the bit functions of the Control/Status Register.

**Table 6-7 Control/Status Register bit assignments**

Bits	Access	Function	Description	Required?	
				JTAG-DP	SW-DP
[31]	RO	CSYSPWRUPACK	System power-up acknowledge. Indicates the state <sup>a</sup> of the <b>CSYSPWRUPACK</b> signal <sup>b</sup> .	Yes, or emulated <sup>c</sup>	
[30]	R/W	CSYSPWRUPREQ	System power-up request. This bit drives <sup>a</sup> the <b>CSYSPWRUPREQ</b> signal <sup>b</sup> . After a power-on reset this bit is 0.	Yes, or emulated <sup>c</sup>	

Table 6-7 Control/Status Register bit assignments (continued)

Bits	Access	Function	Description	Required?	
				JTAG-DP	SW-DP
[29]	RO	CDBGPWRUPACK	Debug power-up acknowledge. Indicates the state <sup>a</sup> of the <b>CDBGPWRUPACK</b> signal <sup>b</sup> .	Yes, or emulated <sup>c</sup>	
[28]	R/W	CDBGPWRUPREQ	Debug power-up request. This bit drives <sup>a</sup> the <b>CDBGPWRUPREQ</b> signal <sup>b</sup> . After a power-on reset this bit is 0.	Yes, or emulated <sup>c</sup>	
[27]	RO	CDBGGRSTACK	Debug reset acknowledge. Indicates the state <sup>a</sup> of the <b>CDBGGRSTACK</b> signal <sup>b</sup> .	Yes, or emulated <sup>c</sup>	
[26]	R/W	CDBGGRSTREQ	Debug reset request. This bit drives <sup>a</sup> the <b>CDBGGRSTREQ</b> signal <sup>b</sup> . After a power-on reset this bit is 0.	Yes, or emulated <sup>c</sup>	
[25:24]	-	-	Reserved, RAZ/SBZP	-	-
[21:12]	R/W	TRNCNT	Transaction counter. For more information see <i>The Transaction Counter</i> on page 3-8. The power-on reset value of this field is UNPREDICTABLE.		Yes
[11:8]	R/W	MASKLANE	Indicates the bytes to be masked in pushed compare and pushed verify operations. See <i>MASKLANE and the bit masking of the pushed compare and pushed verify operations</i> on page 6-14. The power-on reset value of this field is UNPREDICTABLE.		Yes

Table 6-7 Control/Status Register bit assignments (continued)

Bits	Access	Function	Description	Required?	
				JTAG-DP	SW-DP
[7]	RO <sup>d</sup>	WDATAERR <sup>d</sup>	<p>This bit is set to 1 if a Write Data Error occurs. This happens if:</p> <ul style="list-style-type: none"> <li>there is a parity or framing error on the data phase of a write</li> <li>a write that has been accepted by the DP is then discarded without being submitted to the AP.</li> </ul> <p>For more information see <i>Protocol errors, SW-DP only</i> on page 3-4.</p> <p>This bit can only be cleared to 0 by writing 1 to the WDERRCLR field of the AP Abort Register, see <i>The AP Abort Register, ABORT</i> on page 6-6.</p> <p>After a power-on reset this bit is 0.</p>	No <sup>d</sup>	Yes
[6]	RO <sup>d</sup>	READOK <sup>d</sup>	<p>This bit is set to 1 if the response to the previous AP or RDBUFF read was OK. It is cleared to 0 if the response was not OK.</p> <p>This flag always indicates the response to the last AP read access, see <i>Operation and use of the READOK flag and RESEND register</i> on page 5-11 for more information.</p> <p>After a power-on reset this bit is 0.</p>	No <sup>d</sup>	Yes
[5]	RO <sup>e</sup>	STICKYERR	<p>This bit is set to 1 if an error is returned by an AP transaction. For more information see <i>Sticky flags and DP error responses</i> on page 3-2.</p> <p>See <i>Clearing the sticky error flags in the Control/Status Register</i> on page 6-15 for details of how to clear this flag to 0.</p> <p>After a power-on reset this bit is 0.</p>		Yes

Table 6-7 Control/Status Register bit assignments (continued)

Bits	Access	Function	Description	Required?	
				JTAG-DP	SW-DP
[4]	RO <sup>e</sup>	STICKYCMP	This bit is set to 1 when a match occurs on a pushed compare or a pushed verify operation. For more information see <i>Pushed compare and pushed verify operations</i> on page 3-5.  See <i>Clearing the sticky error flags in the Control/Status Register</i> on page 6-15 for details of how to clear this flag to 0. After a power-on reset this bit is 0.	Yes	
[3:2]	R/W	TRNMODE	This field sets the transfer mode for AP operations, see <i>Transfer mode (TRNMODE)</i> , bits [3:2] on page 6-14. After a power-on reset the value of this field is UNPREDICTABLE.	Yes	
[1]	RO <sup>e</sup>	STICKYORUN	If overrun detection is enabled, this bit is set to 1 when an overrun occurs. <sup>f</sup> For more information see <i>Overrun detection</i> on page 3-3.  See <i>Clearing the sticky error flags in the Control/Status Register</i> on page 6-15 for details of how to clear this flag to 0. After a power-on reset this bit is 0.	Yes	
[0]	R/W	ORUNDETECT	This bit is set to 1 to enable overrun detection. For more information see <i>Overrun detection</i> on page 3-3. After a power-on reset this bit is 0.	Yes	

- a. For these bits, the corresponding signal is LOW when the bit = 0 and HIGH when the bit = 1.
- b. For more information about these bits and the associated signals see *System and Debug power and Debug reset control* on page 3-9.
- c. For more information see *Emulation of power control* on page 3-12.
- d. Implemented on SW-DP only. On a JTAG-DP this bit is Reserved, RAZ/SBZP.
- e. RO on SW-DP. On a JTAG-DP, this bit can be read normally, and writing 1 to this bit clears the bit to 0.
- f. See bit [0] of this register for details of enabling overrun detection.

## MASKLANE and the bit masking of the pushed compare and pushed verify operations

The MASKLANE field, bits [11:8] of the CTRL/STAT Register, is only relevant if the Transfer Mode is set to pushed verify or pushed compare operation, see *Transfer mode (TRNMODE), bits [3:2]*.

In the pushed operations, the word supplied in an AP write transaction is compared with the current value at the target AP address. For more information see *Pushed compare and pushed verify operations* on page 3-5. The MASKLANE field lets you specify that the comparison is made using only certain bytes of the values. Each bit of the MASKLANE field corresponds to one byte of the AP values. Therefore, each bit is said to control one byte lane of the compare operation.

Table 6-8 shows how the bits of MASKLANE control the comparison masking.

**Table 6-8 Control of pushed operation comparisons by MASKLANE**

MASKLANE <sup>a</sup>	Meaning	Mask used for comparisons <sup>b</sup>
b1XXX	Include byte lane 3 in comparisons.	0xFF-----
bX1XX	Include byte lane 2 in comparisons.	0x--FF----
bXX1X	Include byte lane 1 in comparisons.	0x---FF--
bXXX1	Include byte lane 0 in comparisons.	0x-----FF

a. Bits [11:8] of the CTRL/STAT Register.

b. Bytes of the mask shown as -- are determined by the other bits of MASKLANE.

If MASKLANE is set to b0000 or to b1111 then the comparison is made on the complete word. In this case the mask is 0xFFFFFFFF.

## Transfer mode (TRNMODE), bits [3:2]

This field sets the transfer mode for AP operations. Table 6-9 lists the permitted values of this field, and their meanings.

**Table 6-9 Transfer Mode, TRNMODE, bit definitions**

TRNMODE <sup>a</sup>	AP Transfer mode
b00	Normal operation.
b01	Pushed verify operation.
b10	Pushed compare operation.
b11	Reserved.

a. Bits [3:2] of the CTRL/STAT Register.

In normal operation, AP transactions are passed to the AP for processing, as described in *Accessing Access Ports* on page 2-8.

In pushed verify and pushed compare operations, the DP compares the value 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. These operations are described in *Pushed compare and pushed verify operations* on page 3-5.

## Clearing the sticky error flags in the Control/Status Register

In the Control/Status Register, the sticky error flags are:

- STICKYERR, bit [5]
- STICKCMP, bit [4]
- STICKORUN, bit [1]

The descriptions of these bits in Table 6-7 on page 6-10 indicate the condition that sets each bit to 1. When one of these bits is set to 1, a write of 0 to that bit is ignored. The method of clearing these bits to 0 is different for the JTAG-DP and the SW-DP:

### On a JTAG-DP

Write 1 to the appropriate bit of this register. For example, if the STICKYERR flag, bit [5], is set to 1 then you must write 1 to bit [5] to clear the flag to 0.

**On a SW-DP** Write 1 to the appropriate CLR field of the AP Abort Register, see *The AP Abort Register, ABORT* on page 6-6:

- to clear STICKYERR to 0, write 1 to the STKERRCLR field, bit [2]
- to clear STICKYCMP to 0, write 1 to the STKCMPCLR field, bit [1]
- to clear STICKYORUN to 0, write 1 to the ORUNERRCLR field, bit [4].

## 6.2.4 The AP Select Register, SELECT

The AP Select Register is always present on all DP implementations. Its main purpose is to select the current *Access Port* (AP) and the active four-word register bank within that AP. On a SW-DP, it also selects the Debug Port address bank.

**JTAG-DP** It is at address 0x8 when the Instruction Register (IR) contains DPACC, and is a read/write register.

**SW-DP** It is at address 0x8 on write operations when the APnDP bit = 0, and is a write-only register. Access to the AP Select Register is not affected by the value of the CTRLSEL bit.

Figure 6-4 on page 6-16 shows the register bit assignments.

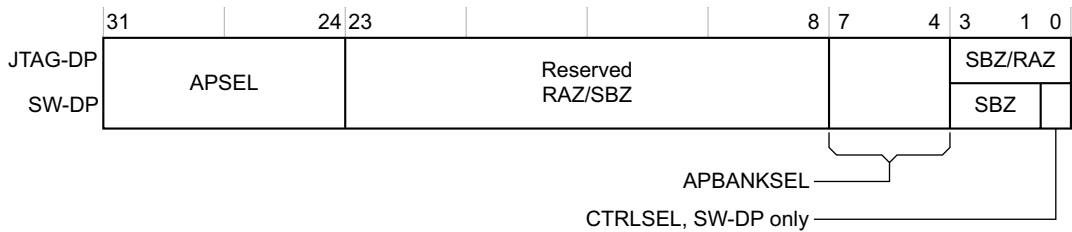


Figure 6-4 Select Register, SELECT, bit assignments

Table 6-10 lists the bit functions of the Select Register.

Table 6-10 Bit assignments for the AP Select Register, SELECT

Bits	Function	Description
[31:24]	APSEL	Selects the current AP. The power-on reset value of this field is UNPREDICTABLE. <sup>a</sup>
[23:8]	-	Reserved. RAZ/SBZ <sup>a</sup> .
[7:4]	APBANKSEL	Selects the active four-word register bank on the current AP. For more information see <i>Accessing Access Ports</i> on page 2-8. The power-on reset value of this field is UNPREDICTABLE. <sup>a</sup>
[3:1]	-	Reserved. RAZ/SBZ <sup>a</sup> .
[0] <sup>b</sup>	CTRLSEL <sup>b</sup>	SW-DP Debug Port address bank select, see <i>CTRLSEL, SW-DP only</i> on page 6-17. After a power-on reset this field is 0. However the register is WO and you cannot read this value.

- a. On a SW-DP the register is write-only and therefore you cannot read the field value.  
b. Implemented on SW-DP only. On a JTAG-DP this bit is Reserved, RAZ/SBZ.

If APSEL is set to a non-existent AP then all AP transactions return zero on reads and are ignored on writes. See *Accesses to Reserved addresses* on page 6-5.

**Note**

Every ARM Debug Interface implementation must include at least one AP.



## CTRLSEL, SW-DP only

The CTRLSEL field, bit [0], controls which DP register is selected at address b01 on a SW-DP. Table 6-11 shows the meaning of the different values of CTRLSEL.

**Table 6-11 CTRLSEL field bit definitions**

CTRLSEL <sup>a</sup>	DP Register at address b01
0	CTRL/STAT, see <i>The Control/Status Register; CTRL/STAT</i> on page 6-10.
1	WCR, see <i>The Wire Control Register; WCR (SW-DP only)</i> on page 6-18.

a. Bit [0] of the SELECT Register.

### 6.2.5 The Read Buffer, RDBUFF

The 32-bit Read Buffer is always present on all DP implementations. However, there are significant differences in its implementation on JTAG-DPs and SW-DPs.

<b>JTAG-DP</b>	It is at address 0xC when the Instruction Register (IR) contains DPACC, and is a Read As Zero, Writes ignored (RAZ/WI) register.
<b>SW-DP</b>	It is at address 0xC on read operations when the APnDP bit = 0, and is a read-only register. Access to the Read Buffer is not affected by the value of the CTRLSEL bit in the SELECT Register.

#### Read Buffer implementation and use on a JTAG-DP

On a JTAG-DP, the Read Buffer is always Read As Zero, and writes to the Read Buffer address are ignored.

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

#### Read Buffer implementation and use on a SW-DP

On a SW-DP, performing a read of the Read Buffer captures data from the AP, presented as the result of a previous read, without initiating a new AP transaction. This means that reading the Read Buffer returns the result of the last AP read access, without generating a new AP access.

After you have read the Read Buffer, its contents are no longer valid. The result of a second read of the Read Buffer is UNPREDICTABLE.

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

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

This second access, to the AP or the DP depending on which option you use, stalls until the result of the original AP read is available.

### 6.2.6 The Wire Control Register, WCR (SW-DP only)

The Wire Control Register is always present on any SW-DP implementation. Its purpose is to select the operating mode of the physical serial port connection to the SW-DP.

The WCR is not present on JTAG-DPs.

On a SW-DP, the WCR is a read/write register at address 0x4 on read and write operations when the CTRLSEL bit in the Select Register is set to 1. For information about the CTRLSEL bit see *The AP Select Register, SELECT* on page 6-15.

#### ————— Note —————

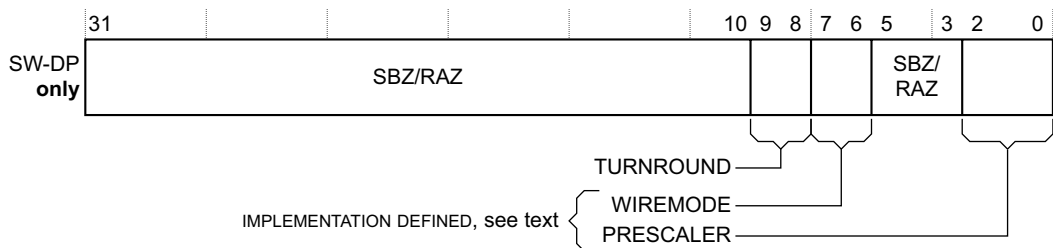
When the CTRLSEL bit is set to 1, to enable access to the WCR, the DP Control/Status Register is not accessible.

Many features of the Wire Control Register are IMPLEMENTATION DEFINED. In particular, it is IMPLEMENTATION DEFINED whether a SW-DP implementation supports:

- only asynchronous operation
- only synchronous operation
- both synchronous and asynchronous operation.

These implementation choices are reflected in the WCR Register.

Figure 6-5 shows the register bit assignments.



**Figure 6-5 Wire Control Register bit assignments (SW-DP only)**

Table 6-12 lists the bit functions of the Wire Control Register.

**Table 6-12 Bit assignments for the Wire Control Register (SW-DP only)**

Bits	Access	Function	Description
[31:10]	-	-	Reserved. RAZ/SBZ.
[9:8]	R/W	TURNROUND	Turnaround tri-state period, see <i>Turnaround tri-state period, TURNROUND, bits [9:8]</i> . After a power-on reset this field is b00.
[7:6]	IMP, <sup>a</sup>	WIREMODE	Selects or indicates the operating mode for the wire connection to the DP, see <i>Wire operating mode, WIREMODE, bits [7:6]</i> on page 6-20. The power-on reset value of this field is IMPLEMENTATION DEFINED.
[5:3]	-	-	Reserved. RAZ/SBZ.
[2:0]	IMP, <sup>a</sup>	PRESCALER	Defines the prescaler for generating the clock used to oversample the Serial Wire Debug interface in asynchronous mode, see <i>Sampling clock prescaler, PRESCALER, bits [2:0]</i> on page 6-20. The power-on reset value of this field is IMPLEMENTATION DEFINED.

a. IMPLEMENTATION DEFINED. See the field description for more information.

### Turnaround tri-state period, TURNROUND, bits [9:8]

This field defines the turnaround tri-state period. This turnaround period provides for pad delays when using a high sample clock frequency. See the protocol description in Chapter 5 *The Serial Wire Debug Port (SW-DP)* for more information. Table 6-13 lists the permitted values of this field, and their meanings.

**Table 6-13 Turnaround tri-state period field, TURNROUND, bit definitions**

TURNROUND <sup>a</sup>	Turnaround tri-state period
b00	1 data period <sup>b</sup> .
b01	2 data periods <sup>b</sup> .
b10	3 data periods <sup>b</sup> .
b11	4 data periods <sup>b</sup> .

a. Bits [9:8] of the WCR Register.

b. A *data period* is the period of a single data bit on the serial wire interface. With synchronous operation this is the same as one Serial Wire clock period.

### Wire operating mode, WIREMODE, bits [7:6]

This field indicates or controls the operating mode, synchronous or asynchronous, of the serial wire connection to the SW-DP. This means that the meaning and access type of this bit is IMPLEMENTATION DEFINED:

#### If the implementation only supports asynchronous operation

WIREMODE is a Read-only field, that always returns the value b00. Writes to this field are ignored.

#### If the implementation only supports synchronous operation

WIREMODE is a Read-only field, that always returns the value b01. Writes to this field are ignored.

#### If the implementation supports both asynchronous and synchronous operation

WIREMODE is a Read/Write field, and Table 6-14 lists the permitted values of the field, and their meanings.

**Table 6-14 Wire operating mode, WIREMODE, bit definitions**

WIREMODE <sup>a</sup>	Wire operating mode
b00	Asynchronous (oversampling).
b01	Synchronous (no oversampling).
b1X	Reserved.

a. Bits [7:6] of the WCR Register.

The oversampling rate in asynchronous operation is IMPLEMENTATION DEFINED.

#### **Note**

Changing the operation of a SW-DP from synchronous mode to asynchronous mode returns the Wire Manager to its reset state. This means the serial interface requires retraining.

### Sampling clock prescaler, PRESCALER, bits [2:0]

This field defines a prescaler that is used to generate a sampling clock, for oversampling the Serial Wire interface when operating in asynchronous mode. This sampling clock rate determines the data rate for the interface.

The interpretation of the value of this field, and the reset value of the field, are IMPLEMENTATION DEFINED. However:

- a PRESCALER value of b000 must select the highest data rate,
- the data rate must decrease as the PRESCALER value increases
- the reset value of PRESCALER must select a data rate of 8MBaud or less.

If an implementation only supports synchronous operation then this field is not supported, and bits [2:0] are RAZ/SBZ.

### 6.2.7 The Read Resend Register, RESEND (SW-DP only)

The Read Resend Register is always present on any SW-DP implementation. Its purpose is to enable the read data to be recovered from a corrupted debugger transfer, without repeating the original AP transfer.

The RESEND Register is not present on JTAG-DPs.

On a SW-DP, the RESEND Register is a 32-bit read-only register at address b10 on read operations. Access to the Read Resend Register is not affected by the value of the CTRLSEL bit in the SELECT Register.

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

Reading the RESEND register enables the read data to be recovered from a corrupted SW transfer without having to re-issue the original read request or generate a new access to the connected debug memory system.

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



# Chapter 7

## Common Access Port (AP) features

An ARM Debug Interface can include multiple Access Ports. ARM Limited provides two AP definitions, but other designers might implement additional APs. This chapter gives an overview of ADI Access Ports, and describes the features that must be implemented in every ADI Access Port. Additional information about APs is given in the following chapters:

- Chapter 8 *The Memory Access Port (MEM-AP)*
- Chapter 9 *The JTAG Access Port (JTAG-AP)*

This chapter contains the following sections:

- *Overview of Access Ports (APs)* on page 7-2
- *The identification model for Access Ports* on page 7-3
- *Selecting and accessing an AP* on page 7-4.

The common register that must be implemented by all ADI Access Ports is described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

## 7.1 Overview of Access Ports (APs)

There are two types of Access Port defined by this ADIV5 specification:

- the Memory Access Port (MEM-AP)
- the JTAG Access Port (JTAG-AP).

An ARM Debug Interface might have multiple Access Ports, and these can be a mixture of types.

### ————— **Note** —————

This Architecture Specification also permits an ARM Debug Interface to include additional Access Port types. Such an AP must implement the common AP register described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*. A debugger must be able to recognize any AP, and must ignore any AP that it does not understand.

If an ARM Debug Interface has only one Access Port this can be a Memory Access Port, a JTAG Access Port, or an additional Access Port, as described in this note. This specification does not require an ADI to include one of the APs that it defines.

The ADIV5 specification requires every Access Port to follow a common identification model. This requirement applies to the MEM-AP and JTAG-AP implementations defined by ARM Limited, to any future AP implementations by ARM Limited, and to any Access Ports that might be implemented by any third party.

It is also required that any Access Port supports ADI accesses from the implemented Debug Port, as described in *Accessing Access Ports* on page 2-8. A summary of how to access an AP is given in *Selecting and accessing an AP* on page 7-4.

The common identification model for APs is summarized in *The identification model for Access Ports* on page 7-3.

There are no other requirements for APs in the ADIV5 specifications. All features provided by an AP can be IMPLEMENTATION DEFINED.



## 7.2 The identification model for Access Ports

This section summarizes the registers that must be implemented by all Access Ports (APs). This requirement applies to the APs that are defined by ARM Limited:

- Memory Access Ports, MEM-APs
- JTAG Access Ports, JTAG-APs.

However, the ARM Debug Interface Architecture Specification permits additional APs to be defined, and any such AP must comply with the identification model given in this section.

The identification model requires every AP to implement an Identification Register:

- the format of the Identification Register is defined by the ADIV5 specification
- the Identification Register must be implemented at offset 0xFC in the register space.

AP register mapping, and the format of the Identification Register, are described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

## 7.3 Selecting and accessing an AP

A full description of how APs are selected and accessed is given in *Accessing Access Ports* on page 2-8. This section summarizes that information.

In any APACC access, to a MEM-AP, to a JTAG-AP or to an AP not defined by this specification:

- Values held in the AP Select Register in the DP determine:
  - Which AP is accessed.
  - Which four-register bank of AP registers is accessed. This information is passed as the A[7:4] field of the MEM-AP access.

For more information see *The AP Select Register, SELECT* on page 6-15.

- The A[3:2] field of the APACC access determines which AP register, within the selected four-register bank, is accessed.
- The RnW bit of the APACC access determines whether the AP register access is a read access or a write access.

For more information about APACC accesses from a JTAG-DP, see *The JTAG-DP DP and AP Access Registers (DPACC and APACC)* on page 4-14.

For more information about APACC accesses from a SW-DP, see *Serial Wire Debug protocol operation* on page 5-5 and *Protocol description* on page 5-10.

APACC accesses to a MEM-AP are shown in Figure 8-1 on page 8-4.

APACC accesses to a JTAG-AP are shown in Figure 9-1 on page 9-3.

### 7.3.1 Stalling accesses

Access Port interfaces can support stalling accesses. These enable the AP to be connected to slow devices, such as a memory system or a long JTAG scan chain. In this way, AP accesses can be pended by the DAP, and do not have to complete within a fixed number of cycles. This is important because in many cases an AP access cannot complete until the associated memory access or JTAG scan has completed. For more information see:

- *Stalling accesses* on page 8-11, for stalling accesses to a MEM-AP
- *Stalling accesses* on page 9-6, for stalling accesses to a JTAG-AP.

# Chapter 8

## The Memory Access Port (MEM-AP)

This chapter describes the implementation of the Memory Access Port (MEM-AP), and how a MEM-AP provides the Debug Port connection to a debug component.

Additional information about MEM-APs is given in the following chapters:

- Chapter 7 *Common Access Port (AP) features*
- Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*
- Chapter 11 *Memory Access Port (MEM-AP) Registers*.

This chapter contains the following sections:

- *Overview of the function of a Memory Access Port (MEM-AP)* on page 8-2
- *MEM-AP functions* on page 8-8
- *MEM-AP examples of pushed verify and pushed compare* on page 8-22
- *MEM-AP implementation requirements* on page 8-24.

## 8.1 Overview of the function of a Memory Access Port (MEM-AP)

A MEM-AP provides a DAP with access to a memory subsystem. Another way of describing the operation of a MEM-AP is that:

- a debug component implements a memory-mapped abstraction of a set of resources
- the MEM-AP provides AP access to those resources.

However, an access to a MEM-AP might only access a register within the MEM-AP, without generating a memory access.

### 8.1.1 The programmer's model for debug register access

The programmer's 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 the architecture specifications permits a debug component to implement multiple 4KB blocks, most components implement a single block. For more information, see the *ARM Architecture Reference Manual Debug Supplement*.

#### Debug Register Files

A 4KB block of address space accessible from an Access Port can be referred to as a Debug Register File. A single Access Port 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 Memory Access Port 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 then 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.

ROM Tables are described in Chapter 14 *ROM Tables*.

### 8.1.2 Selecting and accessing the MEM-AP

Figure 8-1 on page 8-4 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 Embedded Trace Macrocell (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. This is summarized in *Selecting and accessing an AP* on page 7-4.

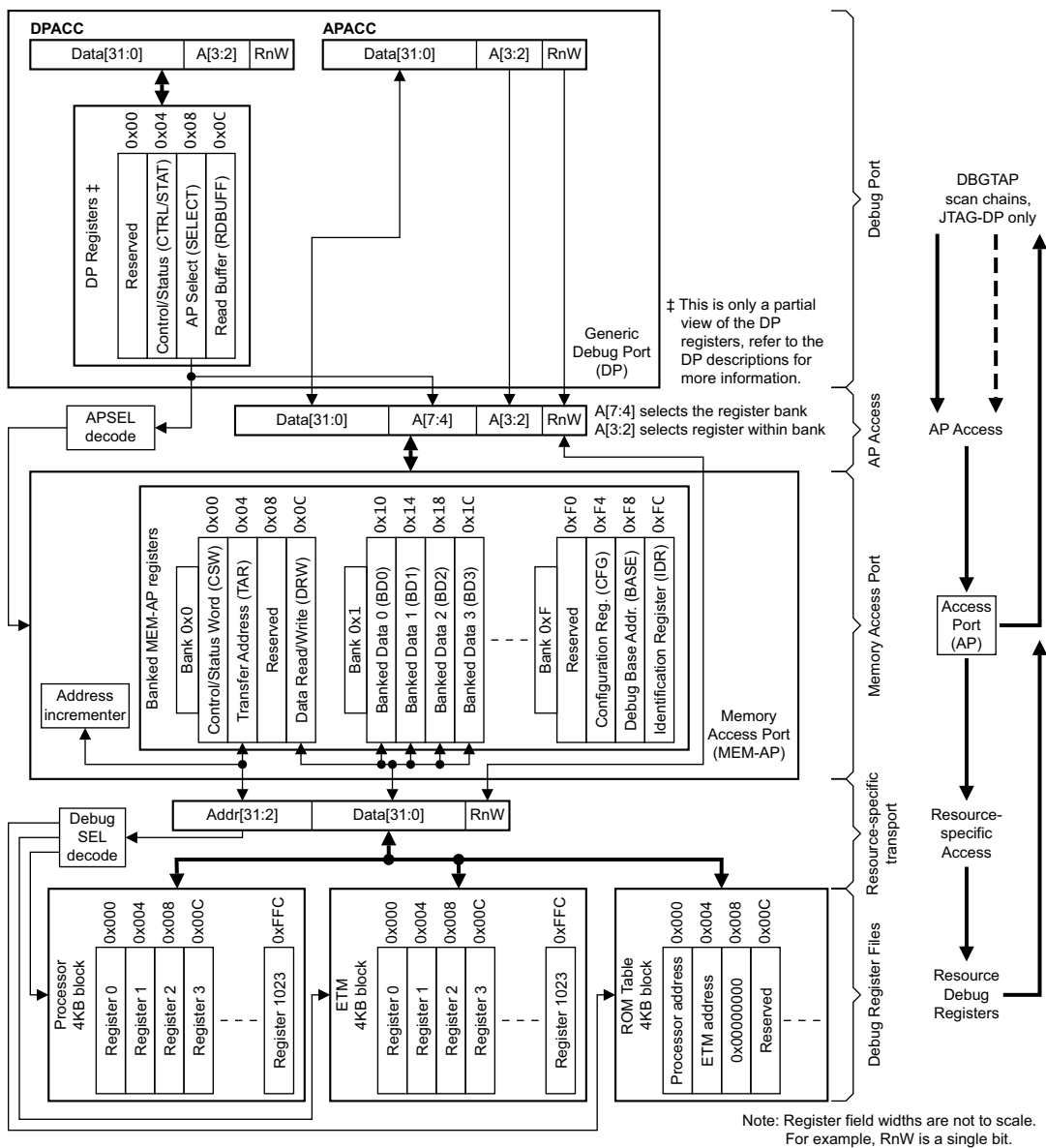


Figure 8-1 MEM-AP connecting the DP to debug components

### 8.1.3 The MEM-AP registers

The MEM-AP registers, and the memory map of the MEM-AP, are described in detail in Chapter 11 *Memory Access Port (MEM-AP) Registers*. However, you require a basic knowledge of the functions of these registers to understand the operation of the MEM-AP. The MEM-AP registers are shown in Figure 8-1 on page 8-4. In outline, these registers are:

#### 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, 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 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 provide direct read or write access to a block of four words of memory, starting at the address specified in the TAR:

- accessing BD0 accesses  $(\text{TAR}[31:4] \ll 4)$  in memory
- accessing BD1 accesses  $((\text{TAR}[31:4] \ll 4) + 0x4)$  in memory
- accessing BD2 accesses  $((\text{TAR}[31:4] \ll 4) + 0x8)$  in memory
- accessing BD3 accesses  $((\text{TAR}[31:4] \ll 4) + 0xC)$  in memory.

#### Configuration Register, CFG

The CFG Register hold information about the configuration of the MEM-AP. In particular, it indicates whether accesses to the connected memory system are big-endian or little-endian.

#### 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 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 Chapter 11 *Memory Access Port (MEM-AP) Registers*.

---

### 8.1.4 MEM-AP register accesses and memory accesses

---

**Note**

---

In this section, an access to the debug resources is described as a memory access. This naming is explained in *The conventional model of the ADI* on page 1-5.

---

This section summarizes all of the possible APACC accesses to a MEM-AP. This means it covers accesses to each of the MEM-AP registers. These accesses can be divided into two groups, summarized in the following sections:

- *Accesses that do not require a memory access*
- *Accesses that initiate a memory access.*

#### Accesses that do not require 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.

APACC accesses to these registers complete immediately.

#### 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 held in the TAR. See *The Data Read/Write Register (DRW)* on page 11-8 and *The Transfer Address Register (TAR)* on page 11-7.
- Accesses to one of the Banked Data Registers, BD0 to BD1. The address used for the memory access depends on which Banked Data Register is accessed:
  - accessing BD0 accesses  $(TAR[31:4] \ll 4)$  in the debug component address map
  - accessing BD1 accesses  $((TAR[31:4] \ll 4) + 0x4)$  in the debug component address map
  - accessing BD2 accesses  $((TAR[31:4] \ll 4) + 0x8)$  in the debug component address map
  - accessing BD3 accesses  $((TAR[31:4] \ll 4) + 0xC)$  in the debug component address map.



Memory accesses through the Banked Data Registers are always 32-bit, so bits [1:0] of the access address are always b00.

———— **Note** ————

The A[3:2] field of the APACC access specifies which of the Banked Data Registers is being accessed. This means that the A[3:2] field value specifies bits [3:2] of the debug component address map address accessed.

For more information see *Banked Data Registers 0 to 3 (BD0 to BD3)* on page 11-9 and *The Transfer Address Register (TAR)* on page 11-7.

In some cases, a single AP transaction initiates more than one memory access. The two cases where this happens are:

- If the transaction counter is set. See *The Transaction Counter* on page 3-8.
- If packed transfers are enabled and the transfer size is smaller than word. See *Packed transfers* on page 8-16.

For more information see *Packed transfers and the Transaction Counter* on page 8-19.

If an AP transaction initiates one or more memory accesses, the AP transaction does not complete until one of the following occurs:

- All of 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, see *The AP Abort Register, ABORT* on page 6-6.

### 8.1.5 Accesses to Reserved addresses

Figure 8-1 on page 8-4 shows the register memory maps for the DP and the AP within the DAP. There are a number of Reserved addresses in these maps. With DAP accesses:

- reads of Reserved addresses return zero (RAZ)
- writes to Reserved addresses are ignored (WI).

## 8.2 MEM-AP functions

This section describes the functions of a MEM-AP. These functions are controlled by the MEM-AP registers, as described in Chapter 11 *Memory Access Port (MEM-AP) Registers*.

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) on page 8-9*
- *Stalling accesses on page 8-11.*

The following sections describe functions that an MEM-AP implementation might support. In other words, it is IMPLEMENTATION DEFINED whether a particular MEM-AP supports these features.

- *Variable access size for memory accesses on page 8-13*
- *Endianness and byte lanes on page 8-14*
- *Packed transfers on page 8-16*
- *Slave memory port and software access control on page 8-19*
- *Additional implementation defined features of a MEM-AP on page 8-20.*

### ———— Note —————

Some of the IMPLEMENTATION DEFINED functions are inter-dependent. These dependencies are summarized in *MEM-AP implementation requirements* on page 8-24.

### 8.2.1 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 Debug Access Port (DAP), used when the DAP is implemented as a separate component in the system. **DEVICEEN** is normally tied HIGH, so that it is asserted even when the Debug Enable signal, **DBGGEN**, is LOW. This means that the MEM-AP can be programmed even when debug is disabled.

The current value of the **DEVICEEN** signal is shown by the read-only DeviceEn flag, bit [6] of the CSW Register, see *Control/Status Word (CSW) Register* on page 11-5.

The DeviceEn flag shows whether the MEM-AP is able to issue transactions to the memory system to which it is connected. When DeviceEn is 0, no transactions can be issued to any address. This means that any access to the Data Read/Write Register or to any of the Banked Data Registers:

- Immediately causes the **STICKYERR** flag in the Debug Port Control/Status Register to be set to 1, see *The Control/Status Register, CTRL/STAT* on page 6-10.
- Has no other effect. The access does not cause a MEM-AP transaction.

If there is no **DEVICEEN** signal for a device then the DeviceEn flag must Read-as-One.

## 8.2.2 Auto-incrementing the Transfer Address Register (TAR)

As indicated in *The MEM-AP registers* on page 8-5, the Transfer Address Register (TAR) holds an address in the address map of the debug resource connected to the MEM-AP. This address is used as:

- the address in the debug component memory map of read or write accesses initiated by a read or write of the Data Read/Write Register
- the base address used to determine the address in the debug component memory map of read or write accesses initiated by a read or write of one of the Banked Data Registers, as described in *Accesses that initiate a memory access* on page 8-6.

You can configure the MEM-AP to auto-increment the TAR on every read or write access to the Data Read/Write Register. Auto-incrementing is controlled by the AddrInc field of the CSW Register, see *Control/Status Word (CSW) Register* on page 11-5.

### ———— Note ————

Accesses to the Banked Data Registers never cause the TAR to auto-increment. The AddrInc field has no effect on accesses to the Banked Data Registers.

The permitted values of the AddrInc field, and their meanings, are summarized in Table 8-1.

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.

**Table 8-1 Summary of AddrInc field values**

AddrInc value	Description	Support required?
b00	Auto-increment off	Always.
b01	Increment single	Always.
b10	Increment packed	If Packed transfers are supported. See <i>Packed transfers</i> on page 8-16. When Packed transfers are not supported, value b10 is Reserved.
b11	Reserved	-

In more detail, the possible settings of this field are:

### **Auto-increment off**

No auto-incrementing occurs. The value in the TAR is unchanged after any Data Read/Write Register access.

### Increment single

After a successful Data Read/Write Register 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 8-13.

#### ————— Note —————

It is IMPLEMENTATION DEFINED whether a MEM-AP supports transfer sizes other than Word. If a MEM-AP only supports word transfers then, when Increment single is selected, the TAR always increments by four after a successful DRW transaction.

Automatic address increment is only guaranteed to operate on the bottom 10-bits of the address held in the TAR. Auto address incrementing of bit [10] and beyond is IMPLEMENTATION DEFINED. This means that auto address incrementing at a 1KB boundary is IMPLEMENTATION DEFINED. For example, if TAR[31:0] is set to 0x14A4, and the access size is word, successive accesses to the DRW increment TAR to 0x14A8, 0x14A, and so on, up to the end of the 1KB range at 0x17FC. At this point, the auto-increment behavior on the next DRW access is IMPLEMENTATION DEFINED.

### Increment packed

If packed transfers are supported, setting AddrInc to b10, Increment packed, enables packed transfer operation. Packed transfers are described in more detail in *Packed transfers* on page 8-16. In brief, with packed transfers multiple half-word or byte memory accesses are packed into a single word APACC access.

It is IMPLEMENTATION DEFINED whether a MEM-AP supports packed transfers, but:

- an implementation that supports transfers smaller than word must support packed transfers
- packed transfers cannot be supported on a MEM-AP that only supports whole-word transfers.

When packed transfer operation is enabled and the transfer size is smaller than 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 Size set to halfword (16 bits), 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 8-16, the two halfword values from the two reads are packed into a single 32-bit word that is returned through the APACC.
- With packed accesses enabled and Size set to byte, a single DRW write operation generates four 8-bit data bus transfers, and the TAR is incremented by 0x1 after each of these transfers.

### 8.2.3 Stalling accesses

A MEM-AP must support stalling accesses, to enable connection to slow devices such as a slow memory system. This means that an access can be issued by the DP but does not have to complete within a fixed number of cycles. This is important because a MEM-AP access to the DRW Register, or to one of the Banked Data Registers BD0 to BD3, does not complete until the required memory access completes.

An example of the importance of stalling accesses is given by the mode of operation, specified by the ARMv7 Debug Architecture, 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. The process is:
  - Perform an AP write to DRW with the first instruction to execute:
    - the AP is ready, so an OK/FAULT ACK is returned at the Capture-DR state
    - at the Update-DR state the DP initiates a transfer to the AP.
  - The TAR is addressing the ITR on the processor, and the AP access is a write to the DRW. Therefore, the AP initiates a write to the ITR through its connection to the processor.
  - The core accepts the transfer, because the processor is idle and the instruction complete flag, InstrCompl, is set to 1.
  - The transfer completes.
  - The core starts to execute the instruction from the ITR. InstrCompl = 0.

———— **Note** ————

The ACK of OK/FAULT is issued *before* the transfer is accepted by the core.

3. The debugger writes a second instruction to the ITR:
  - It performs an AP write to DRW with the next instruction to execute:
    - the AP is ready, so an OK/FAULT ACK is returned at the Capture-DR state
    - at the Update-DR state the DP initiates a transfer to the AP.
  - The TAR has not changed, therefore, the AP initiates a second write to the ITR through its connection to the processor
  - The core is still executing the first instruction (InstrCompl = 0) and cannot accept the transfer.
  - The transfer can not complete, and the AP remains busy.

---

**Note**

---

The ACK value returned is 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:
  - It performs an AP write to DRW with the next instruction to execute:
    - The AP is not ready, so a WAIT ACK is returned at the Capture-DR state.
    - At the Update-DR state the DP discards the AP access request, because the AP was not ready at Capture-DR.
  - The debugger might now retry the AP write a number of times, but as long as the first instruction has not completed, the DP returns a WAIT ACK at the Capture-DR state
5. At some point the core completes the first instruction:
  - InstrCompl becomes 1.
  - The External Debug Interface on the core is now ready to accept the second instruction.
  - The AP transfer, from stage 3, is accepted by the core, and the second instruction is written to the ITR.
  - The core starts to execute the second instruction. InstrCompl = 0, again.
  - Because the AP transfer is complete, the AP returns to the ready state.
6. The debugger retries writing the third instruction to the ITR:
  - It performs an AP write to DRW with the third instruction:
    - the AP is ready, so an OK/FAULT ACK is returned at the Capture-DR state
    - at the Update-DR state the DP initiates a transfer to the AP.
  - The TAR has not changed, therefore, the AP initiates another write to the ITR through its connection to the processor.
  - The response to the AP write attempt depends on whether the processor has finished processing the last instruction written to the ITR:
    - If the processor is idle, with InstrCompl = 1, 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 = 0. The processor cannot accept the transfer and the AP remains busy. The debug session continues from stage 4.

## 8.2.4 Response to debug component errors

On an error response from a debug component, the MEM-AP returns an error to the DP. As a result of this, the DP sets the STICKYERR flag, see *Sticky flags and DP error responses* on page 3-2.

### 8.2.5 Variable access size for memory accesses

It is IMPLEMENTATION DEFINED whether an MEM-AP supports memory access sizes smaller than word (32-bit).

When a MEM-AP implementation supports different sized accesses, the implementation *must* support word, half-word and byte accesses, and the access size is controlled by the Size field of the CSW Register, see *Control/Status Word (CSW) Register* on page 11-5.

———— **Note** ————

The ARM Debug Interface specification does not require a MEM-AP to support accesses smaller than word. However if a MEM-AP can access other memory, such as system memory, ARM Limited recommends that it supports other access sizes.

For more information see *MEM-AP implementation requirements* on page 8-24.

Table 8-2 summarizes the access size options.

**Table 8-2 Size field values when the MEM-AP supports different access sizes**

Size value, CSW[2:0]	Memory access size	Support required?
b000	Byte (8 bits)	No
b001	Halfword (16 bits)	No
b010	Word (32-bit)	Yes <sup>a</sup>
b011 - b111	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 b010.

When a Size other than word is used, the resulting data access is returned in byte lanes. See *Endianness and byte lanes* on page 8-14 for more information.

———— **Caution** ————

If an implementation supports access sizes other than word then the Size field *must* be set to word before making any access to the Banked Data Registers. Behavior is UNPREDICTABLE if a Banked Data Register is accessed with Size set to any value other than word.

## 8.2.6 Endianness and byte lanes

When a MEM-AP implementation supports memory accesses of different sizes it is IMPLEMENTATION DEFINED whether the memory accesses are big-endian or little-endian. The value of the CFG register indicates whether the implementation is big-endian or little-endian, see *Configuration Register (CFG)* on page 11-10.

---

### Note

- Word-invariant big-endian memory addressing schemes, sometimes referred to as BE-32 addressing, are not supported.
  - If the MEM-AP implementation only supports word (32-bit) data bus accesses, the value of the CFG register has no significance to the AP. However, the register value is still required by the debugger.
- 

## Data byte-laning

With a MEM-AP that supports memory transfers of less than 32-bits, when packed transfers are not enabled, the data transfers between the DRW and the debug component are byte-laned as described in this section. This byte-laning depends on:

- the memory transfer size, specified by the Size field in the CSW, see *Variable access size for memory accesses* on page 8-13
- the bottom two bits of the TAR, TAR[1:0], see *The Transfer Address Register (TAR)* on page 11-7
- the endianness of the MEM-AP.

---

### Note

Packed transfers also use byte laning for byte and halfword transfers. This is described in *Packed transfers* on page 8-16.

---

Table 8-3 shows how the DRW data is byte-laned:

**Table 8-3 Byte-laning of memory accesses from the DRW**

		Access data	
CSW[2:0], Size	TAR[1:0]	Little-endian	Big-endian
b000, byte	b00	DRW[7:0]	DRW[31:24]
	b01	DRW[15:8]	DRW[23:16]
	b10	DRW[23:16]	DRW[15:8]
	b11	DRW[31:24]	DRW[7:0]



**Table 8-3 Byte-laning of memory accesses from the DRW (continued)**

CSW[2:0], Size	TAR[1:0]	Access data	
		Little-endian	Big-endian
b001, halfword	b00	DRW[15:0]	DRW[31:16]
	b10	DRW[31:16]	DRW[15:0]
	bX1	IMPLEMENTATION DEFINED	
b010, word	b00	DRW[31:0]	DRW[31:0]
	b1X, bX1	IMPLEMENTATION DEFINED	

The IMPLEMENTATION DEFINED behavior shown in Table 8-3 on page 8-14 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 STICKYERR bit in the Debug Port Control/Status Register is set to 1, see *The Control/Status Register, CTRL/STAT* on page 6-10.
- The access is made to the unaligned address specified in TAR[31:0], and the result packed as if packed transfers were enabled, see *Packed transfers* on page 8-16. 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 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.

## 8.2.7 Packed transfers

If packed transfers are supported they are enabled by setting the auto address increment field, *AddrInc*, in the CSW register to b10, Increment packed. See *Auto-incrementing the Transfer Address Register (TAR)* on page 8-9.

When packed transfers are enabled, each access to the DRW results in one of the following actions, depending on the value of the *Size* field in the field in the CSW, see *Variable access size for memory accesses* on page 8-13:

- when *Size* = b010, word, there is a single word (32-bit) access
- when *Size* = b001, halfword, there are two halfword (16-bit) accesses
- when *Size* = b000, byte, there are four byte (8-bit) accesses.

### Note

It is IMPLEMENTATION DEFINED whether a MEM-AP supports packed transfers. However, if a MEM-AP supports access sizes smaller than word it must also support packed transfers.

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 those described in Table 8-3 on page 8-14. This is shown in the following examples:

- Example 8-1 on page 8-17, *Halfword packed write operation on a little-endian MEM-AP* on page 8-17
- Example 8-2 on page 8-17, *Halfword packed write operation on a big-endian MEM-AP* on page 8-17
- Example 8-3 on page 8-17, *Byte packed write operation on a little-endian MEM-AP* on page 8-17
- Example 8-4 on page 8-18, *Halfword packed read operation on a little-endian MEM-AP* on page 8-18.

### Example 8-1 Halfword packed write operation on a little-endian MEM-AP

This example describes a single word (32-bit) write access to the DRW on an little-endian MEM-AP with the following settings:

- Size, CSW[2:0] = b001, to give halfword (16-bit) memory accesses
- AddrInc, CSW[5:4] = b10, to give packed transfer operation
- TAR[31:0] = 0x00000000, to define the base address of the access.

Two write transfers are made. The little-endian *halfword* entries in Table 8-3 on page 8-14 define the byte-laning for these accesses. The accesses are made in the following order:

1. TAR[1]=0, so write DRW[15:0] to address 0x00000000.  
After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes 0x00000002.
2. TAR[1]=1, so write DRW[31:16] to address 0x00000002.  
After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes 0x00000004.

### Example 8-2 Halfword packed write operation on a big-endian MEM-AP

This example shows a similar transfer to Example 8-1, but on a big-endian MEM-AP. The operation is a single word (32-bit) write access to the DRW, with the following settings:

- Size, CSW[2:0] = b001, to give halfword (16-bit) memory accesses
- AddrInc, CSW[5:4] = b10, to give packed transfer operation
- TAR[31:0] = 0x00000000, to define the base address of the access.

Two write transfers are made. The big-endian *halfword* entries in Table 8-3 on page 8-14 define the byte-laning for these accesses. The accesses are made in the following order:

1. TAR[1]=0, so write DRW[31:16] to address 0x00000000.  
After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes 0x00000002.
2. TAR[1]=1, so write DRW[15:0] to address 0x00000002.  
After this transfer, the value in the TAR is increased by the transfer size, 2, and becomes 0x00000004.

### Example 8-3 Byte packed write operation on a little-endian MEM-AP

This example describes a single word (32-bit) write access to the DRW on an little-endian MEM-AP with the following settings:

- Size, CSW[2:0] = b000, to give byte (8-bit) memory accesses

- AddrInc, CSW[5:4] = b10, to give packed transfer operation
- TAR[31:0] = 0x00000002, to define the base address of the access.

Four write transfers are made. The little-endian *byte* entries in Table 8-3 on page 8-14 define the byte-laning for these accesses. The accesses are made in the following order:

1. TAR[1:0]=b10, so write DRW[23:16] 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]=b11, so write DRW[31:24] 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]=b00, so write DRW[7:0] 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]=b01, so write DRW[15:8] to address 0x00000005.  
After this transfer, the value in the TAR is increased by the transfer size, 1, and becomes 0x00000006.
- 

#### Example 8-4 Halfword packed read operation on a little-endian MEM-AP

---

This example describes a single word (32-bit) read access to the DRW on an little-endian MEM-AP with the following settings:

- Size, CSW[2:0]= b001, to give halfword (16-bit) memory accesses
- AddrInc, CSW[5:4] = b10, 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 8-3 on page 8-14 define the byte-laning for these accesses. The accesses are made in the following order:

1. TAR[1]=1, 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.
2. TAR[1]=0, 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.

At this point, the complete word has been read into the DRW, and the APACC read access completes.

---

The descriptions given in these four 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.

---

**Note**

---

Packing occurs before any pushed comparisons are made. Pushed comparisons are made, and the STICKYCMP flag set to 1 if necessary, only when a complete word of data has been packed into the DRW. See *Pushed compare and pushed verify operations* on page 3-5 for a description of pushed comparisons.

---

**Packed transfers and the Transaction Counter**

The Debug Port Transaction Counter, described in *The Transaction Counter* on page 3-8, means that an external debugger can 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 transfer operation is enabled, each of the AP transactions 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, altogether 32 memory accesses are made, each of one byte.

**8.2.8 Slave memory port and software access control**

The CSW Register can include a Debug software access enable flag, DbgSwEnable, bit [31]. See *Control/Status Word (CSW) Register* on page 11-5 for details of the CSW Register.

It is IMPLEMENTATION DEFINED whether this flag is supported. There are two alternative possible uses for this flag, described in the following sub-sections:

- *Using DbgSwEnable to control a slave memory port* on page 8-20
- *Using DbgSwEnable to control software access to debug resources* on page 8-20.

If neither of these features is implemented then the DbgSwEnable bit must Read As Zero.

A MEM-AP can include a slave memory port, to enable an external bus master to access the area of memory mastered by the MEM-AP. An example of this use would be permitting the external bus master to access the debug registers of the system to which the MEM-AP is connected.

If a MEM-AP implements a slave memory port then slave memory port accesses are multiplexed with DAP accesses. Slave memory port accesses have bit [31] of the access address forced to zero. This enables debug components to distinguish between slave memory port accesses and DAP accesses.

More information about MEM-AP memory addressing is given in *The BASEADDR field, BASE[31:12]* on page 11-12.

---

**Note**

---

The DAP can emulate a slave memory port access by setting bit [31] of the Transfer Address Register to 0.

---

## Using DbgSwEnable to control a slave memory port

If a MEM-AP implements a slave memory port then the DbgSwEnable bit must be implemented so that the port can be enabled or disabled. The behavior of this bit is shown in Table 8-4.

**Table 8-4 Use of DbgSwEnable bit, bit [31], to control a slave memory port**

DbgSwEnable (bit [31])	Slave memory port
0	Disabled.
1	Enabled. This is the reset value.

## Using DbgSwEnable to control software access to debug resources

If a MEM-AP does not implement a slave memory port then the DbgSwEnable bit can be used to drive a system-level signal, **DBGSWENABLE**. This signal is used to gate software access to debug resources. For example, in a processor that complies with the ARMv7 Debug Architecture, software cannot access Extended CP14 Interface registers when the **DBGSWENABLE** signal is LOW. For more information see the *ARM Architecture Reference Manual Debug Supplement*.

Table 8-5 shows the behavior of the DbgSwEnable bit when **DBGSWENABLE** is implemented.

**Table 8-5 Use of DbgSwEnable bit, bit [31], to control the DBGSWENABLE signal**

DbgSwEnable (bit [31])	DBGSWENABLE signal
0	LOW.
1	HIGH. This is the reset value.

### 8.2.9 Additional IMPLEMENTATION DEFINED features of a MEM-AP

The definition of the CSW Register includes two optional fields that are not described elsewhere in this chapter:

#### Prot, CSW bits [30:24]

This field can be implemented to provide a bus access control mechanism. If implemented, it enables a debugger to specify protection flags for a memory access. The permitted values and their significance are IMPLEMENTATION DEFINED, because they relate to the underlying bus architecture. ARM Limited recommends that these bits reset to useful values, that might not be zero. For example:

- if the bus supports privileged and non-privileged accesses, the reset value of this field should select privileged accesses

- if the bus supports code and data accesses, the reset value should select data accesses.

**SPIDEN, CSW bit [23]**

This field can be implemented to indicate whether Secure Privileged Debug is enabled.

For details of the CSW Register, see *Control/Status Word (CSW) Register* on page 11-5.

## 8.3 MEM-AP examples of pushed verify and pushed compare

Every ADI Debug Port (DP) supports pushed operations, as described in *Pushed compare and pushed verify operations* on page 3-5. However, these operations involve interaction between the DP and an AP, because each pushed operation requires an AP read. In the case of a MEM-AP, this requires a read from the connected debug memory system. This section gives some examples of pushed operations on an DP that is connected to a MEM-AP.

### 8.3.1 Example use of pushed verify operation on a MEM-AP

You can use pushed verify to verify the contents of system memory:

- Make sure that the MEM-AP Control/Status Word (CSW) is set up to increment the Transfer Address Register (TAR) after each access. See *Control/Status Word (CSW) Register* on page 11-5.
- Write to the Transfer Address Register (TAR) to indicate the start address of the memory region that is to be verified, see *The Transfer Address Register (TAR)* on page 11-7.
- 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 supplied in the AP write transaction, and sets the STICKYCMP bit in the CRL/STAT Register if the values do not match. See *The Control/Status Register, CTRL/STAT* on page 6-10.

The TAR is incremented on each transaction.

In this way, the series of values supplied is compared against the contents of the memory region, and STICKYCMP set to 1 if they do not match.

### 8.3.2 Example use of pushed find operation on a MEM-AP

You can use pushed find to search system memory for a particular word.

- Make sure that the MEM-AP Control/Status Word (CSW) is set up to increment the TAR after each access. See *Control/Status Word (CSW) Register* on page 11-5.
- Write to the Transfer Address Register (TAR) to indicate the start address of the Debug Register region that is to be searched. See *The Transfer Address Register (TAR)* on page 11-7.
- Repeatedly write the value to be searched for as an AP write transaction. On each transaction the DP reads the location indicated by the TAR. On each DP read:
  - The value returned is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to 1.
  - The TAR is incremented.

If you use pushed find with byte lane masking you can search for one or more bytes.

*Example using the Transaction Counter for a pushed find operation on a MEM-AP* on page 8-23 describes how the Transaction Counter can be used to refine this search operation.



You can use pushed find without address incrementing to poll a single location, for example to test for a flag being set to 1 on completion of an operation.

### 8.3.3 Example using the Transaction Counter for a pushed find operation on a MEM-AP

The Transaction Counter can be used to refine the pushed find search operation described in *Example use of pushed find operation on a MEM-AP* on page 8-22. Pushed find enables you to search system memory for a particular word, and if used with byte lane masking you can search for one or more bytes. The Transaction Counter enables you use a single AP write transaction to search an area of memory.

To perform a search under the control of the Transaction Counter:

- Make sure that the MEM-AP Control/Status Word (CSW) is set up to increment the TAR after each access. See *Control/Status Word (CSW) Register* on page 11-5.
- Write to the Transfer Address Register (TAR) to indicate the start address of the Debug Register region that is to be searched. See *The Transfer Address Register (TAR)* on page 11-7.
- Write to the Transaction Counter field, TRNCNT, of the DP Control/Status Register, to indicate the required number of repeat accesses. This value defines the size of the region to be searched.
- Write the value you are searching for as an AP write transaction. The DP repeatedly reads the location indicated by the TAR. On each DP read:
  - The value returned is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to 1.
  - The TAR is incremented.
  - The Transaction Counter, TRNCNT, is decremented.

This continues until either STICKYCMP is set to 1 or TRNCNT reaches zero.

## 8.4 MEM-AP implementation requirements

The descriptions given in the section *MEM-AP functions* on page 8-8 indicate a number of 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
  - support word (32-bit), half-word (16-bit) and byte (8-bit) accesses.

No other combinations of supported access sizes are permitted. For more information see *Variable access size for memory accesses* on page 8-13.
- It is IMPLEMENTATION DEFINED whether memory accesses are big-endian or little-endian:
  - The CFG register indicates the endianness, see *Configuration Register (CFG)* on page 11-10. This register must be implemented, even if the MEM-AP only supports word accesses, because the debugger must be able to find the endianness of the data transfers.

**Note**

Even if a MEM-AP does not support memory transfers smaller than word the debugger must be able to find the endianness of the memory access so that it can synthesize halfword and byte accesses.

  - If the MEM-AP supports memory transfers smaller than word then the endianness affects the byte laning of halfword and byte transfers, see *Endianness and byte lanes* on page 8-14.
- If access sizes other than 32-bit are supported then:
  - Support for packed transfers must be implemented. For more information see *Packed transfers* on page 8-16.

# Chapter 9

## The JTAG Access Port (JTAG-AP)

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.

Additional information about JTAG-APs is given in the following chapters:

- Chapter 7 *Common Access Port (AP) features*
- Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*
- Chapter 12 *JTAG Access Port (JTAG-AP) Registers*.

This chapter contains the following sections:

- *Overview of the function of a JTAG Access Port (JTAG-AP)* on page 9-2
- *The JTAG Engine Byte Command Protocol* on page 9-10.

## **9.1 Overview of the function of a JTAG Access Port (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 Limited recommends that only one TAP is connected to each scan chain.

### **9.1.1 Selecting and accessing the JTAG-AP**

Figure 9-1 on page 9-3 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. This is summarized in *Selecting and accessing an AP* on page 7-4.

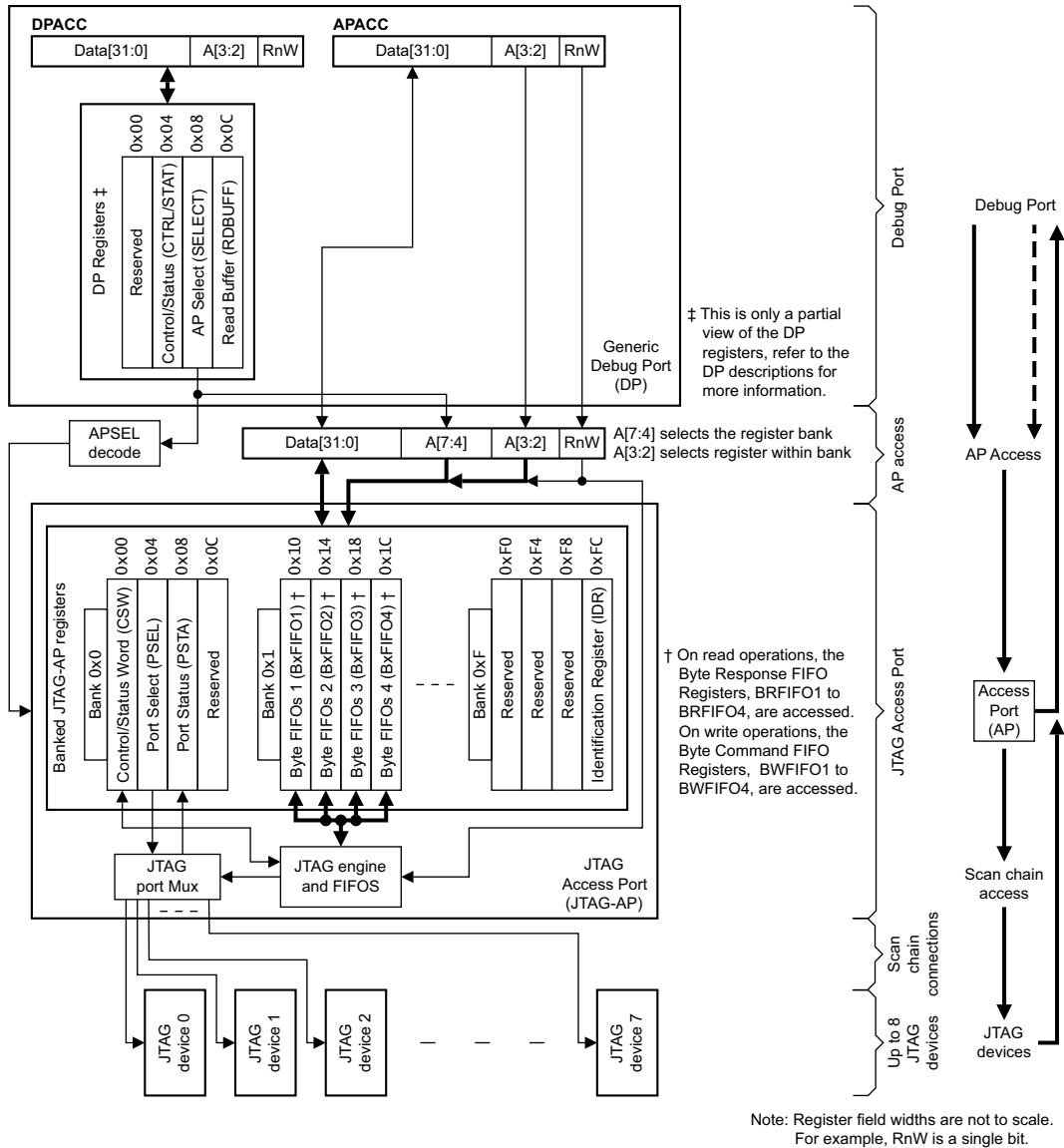


Figure 9-1 JTAG-AP connecting the DP to JTAG devices

### 9.1.2 Logical structure of the JTAG-AP

A JTAG-AP comprises:

- The JTAG Engine. This is the main processing component of the JTAG-AP. It:
  - 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, that it passes to the Response FIFO
- The JTAG Port Multiplexer. This 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 additional control and status signals for each port.

———— **Note** ————

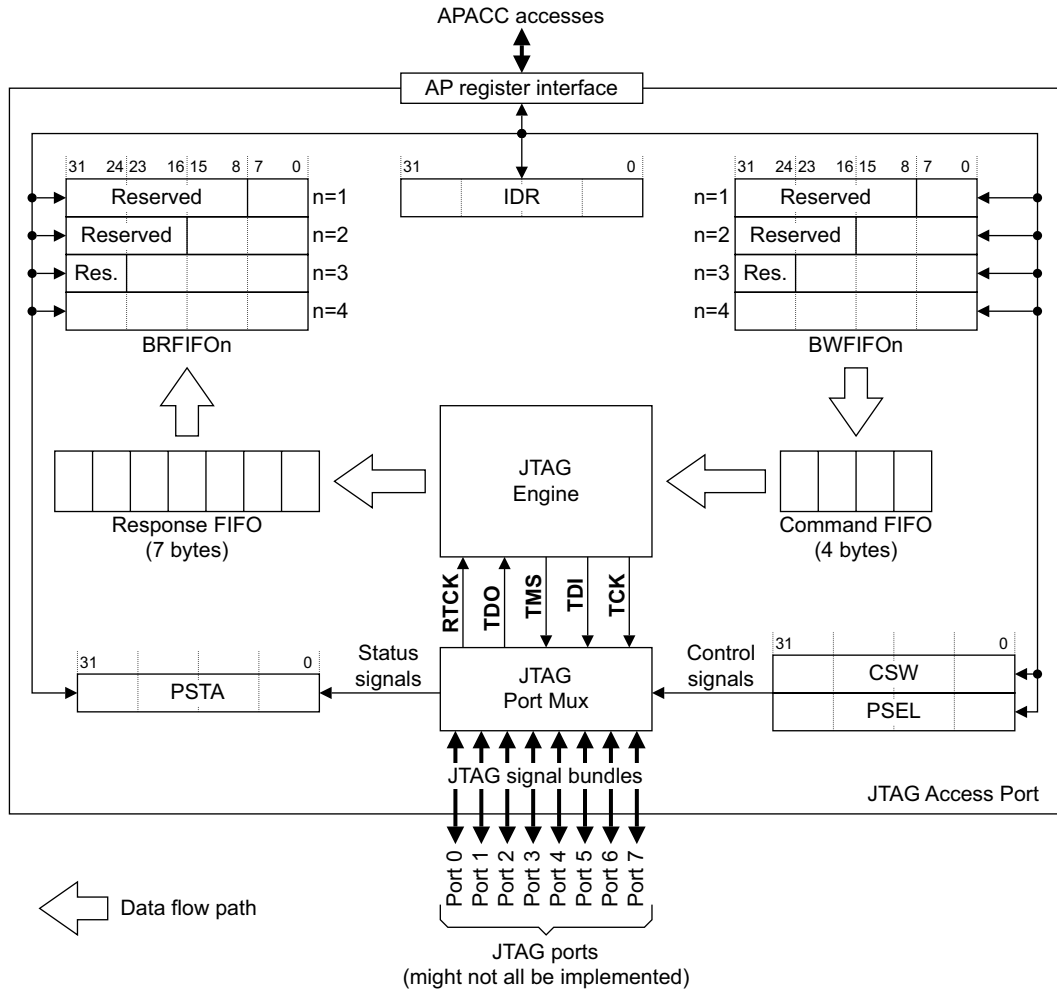
The Port Multiplexer also supports the **RTCK** (return clock) extension to the JTAG protocol, enabling the JTAG scan chains to be self-timed. The JTAG signals from and to the JTAG-AP are asynchronous to the Debug Port signals.

- Byte Command and Response FIFOs. These enable efficient use of the JTAG Engine.
- The JTAG-AP registers. These can be considered in three groups:
  - an Identification Register
  - control and status registers
  - FIFO access registers.

Figure 9-2 on page 9-5 shows this JTAG-AP structure. The diagram includes all of the JTAG-AP registers. These registers are:

- CSW, the Control/Status Word Register
- PSEL, the Port Select Register
- PSTA, the Port Status Register
- BWFIFOn, the Byte Write FIFO Registers 1 to 4
- BRFIFOn, the Byte Read FIFO Registers 1 to 4
- IDR, the JTAG-AP Identification Register.

These registers are described fully in Chapter 12 *JTAG Access Port (JTAG-AP) Registers*.



**Figure 9-2 Structure of the JTAG Access Port (JTAG-AP)**

**Note**

- The Response FIFO must be 7 bytes deep.
- The Command FIFO must be at least 4 bytes deep. Although the Command FIFO might be larger than this, up to a maximum size of 7 bytes, there is unlikely to be any advantage in having a Command FIFO that is larger than 4 bytes.

### 9.1.3 JTAG-AP register accesses and accesses to the JTAG scan chain

An external debugger accesses a JTAG component, 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. For example, referring to Figure 9-2 on page 9-5, the debugger must write to the PSEL register to select the *JTAG port* that is connected through the *JTAG Port Mux* to the *JTAG Engine*.

The description of APACC accesses to JTAG-AP registers shows that the register accesses can be divided into two groups:

- Register accesses that do not access the JTAG engine FIFOs. This group comprises all accesses to the following registers, as shown in Figure 9-2 on page 9-5:
  - CSW, see *The Control/Status Word Register, CSW* on page 12-5
  - PSEL, see *The Port Select Register, PSEL* on page 12-7
  - PSTA, see *The Port Status Register, PSTA* on page 12-9
  - IDR, see *The Identification Register, IDR* on page 10-3.

Accesses to these registers always complete immediately. These accesses do not require any communication with any connected JTAG component.

- Register accesses that access the JTAG engine FIFOs. This group comprises all accesses to the following registers, as shown in Figure 9-2 on page 9-5:
  - the Byte Response FIFO Registers, BRFIFO1 to BRFIFO4
  - the Byte Command FIFO Registers, BWFIFO1 to BWFIFO4.

These registers are described in *The Byte FIFO registers, BRFIFO1 to BRFIFO4 and BWFIFO1 to BWFIFO4* on page 12-10. Using these registers to access the JTAG state machine and JTAG scan chains is described in *The JTAG Engine Byte Command Protocol* on page 9-10.

Accesses to these registers can be stalled, see *Stalling accesses*.

#### Stalling accesses

*Stalling accesses* on page 7-4 stated that a JTAG-AP must be able to stall accesses from a Debug Port, to permit connection to a slow JTAG scan chain. A JTAG-AP can stall:

- read accesses to the Byte Response FIFO Registers, BRFIFO1 to BRFIFO4, see *Stalling Debug Port read accesses to the JTAG-AP*
- write accesses to the Byte Command FIFO Registers, BWFIFO1 to BWFIFO4, see *Stalling Debug Port write accesses to the JTAG-AP* on page 9-7.

#### Stalling Debug Port read accesses to the JTAG-AP

The JTAG-AP can stall DP read accesses to the Byte Response FIFO Registers, BRFIFO1 to BRFIFO4. Depending which of these registers is targeted, a single register read transfers between one and four bytes of data from the byte Response FIFO. The register access stalls if the FIFO does not contain enough data.



For example, if a DP initiates a read of the BRFFIFO4 register, to transfer four bytes of data from the Response FIFO, and the FIFO only contains two bytes of data, the access stalls and remains stalled until there are four bytes of data available in the Response FIFO.

The DP can read the CSW Register to find the number of bytes of data available in the Response FIFO. This value is held in the RFIFOCNT field, see *The Control/Status Word Register, CSW* on page 12-5. A read of the CSW always completes immediately.

### **Stalling Debug Port write accesses to the JTAG-AP**

The JTAG-AP can stall DP write accesses to the Byte Command FIFO Registers, BWFIFO1 to BWFIFO4. Depending which of these registers is targeted, a single register write transfers between one and four bytes of data into the byte Command FIFO. The register access stalls if the FIFO does not contain enough free space to accept all of the write data. For example, if a DP initiates a write to the BWFIFO3 register, to transfer three bytes of data into the Command FIFO, when the FIFO has only one byte free, the access stalls and remains stalled until the Command FIFO is able to accept the three bytes of data.

The DP can read the CSW Register to find the number of command bytes held in the Command FIFO and waiting to be processed by the JTAG engine. This value is held in the WFIFOCNT field, and can be used to calculate the number of free bytes in the FIFO. For more information see *The Control/Status Word Register, CSW* on page 12-5. A read of the CSW always completes immediately.

### **Normal operation of the JTAG-AP**

Operation of the JTAG-AP is described in more detail in *The JTAG Engine Byte Command Protocol* on page 9-10. This section provides a simple overview of that operation, to provide the context of the detailed description.

This simple description considers a JTAG-AP connected to a single JTAG device.

The JTAG-AP communicates with the device using the 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 9-2 on page 9-5.

The external debugger has a memory-mapped register interface to the ADI, that it uses to encapsulate a standard JTAG data flow. There are four stages to this process:

1. JTAG commands and data are encoded into the JTAG Engine Byte Command Protocol, described in *The JTAG Engine Byte Command Protocol* on page 9-10.
2. The debugger writes to the BWFIFO<sub>n</sub> registers, to transfer the encoded JTAG Engine commands and data to the JTAG Command FIFO. This stage is equivalent to sending TDI data to the JTAG device.
3. The debugger reads from the BRFFIFO<sub>n</sub> registers. These reads return JTAG TDO data that is collected in response to the encoded JTAG Engine commands.
4. The debugger decodes the actual TDO data from the response data.

The JTAG Engine provides the connection between stages 2 and 3 of this process.

There is no direct coupling between the data transfers between:

- the data transfers between the debugger and the JTAG-AP
- the data transfers between the debugger and the connected JTAG device.

In particular, the debugger does not have to send all the TDI data for a scan before it starts to read the TDO data from the scan, although it must have sent the complete command header. Indeed, for a long scan, the command and Response FIFOs might not be large enough for this approach to be possible.

Without going into the details of the JTAG Engine Byte Command Protocol, a typical debugger session might start:

- the debugger writes two bytes to BWFIFO2, to specify:
  - a TDI\_TDO scan, with 64 bits of TDI data
  - the TDO data is to be returned to the debugger
- the debugger writes a word to BWFIFO4, containing the first 32 bits of TDI data
- the debugger reads a word from BRFIFO4, to obtain the first 32 bits of TDO data
- the debugger writes another word to BWFIFO4, with the next 32 bits of TDI data
- the debugger reads another word from BRFIFO4, to obtain the next 32 bits of TDO data.

This provides a very efficient encapsulation of the JTAG scan chain. However, as described in *Stalling accesses* on page 9-6, a read of BRFIFO4 stalls if the requested data is not available. Therefore, if the device connected to the JTAG-AP has a slow clock then the debugger might want to write a number of bytes of TDI data before attempting to read the first byte of TDO data.

## Resetting connected JTAG devices or subsystems

Resets are triggered using

- the **TRST\*** signal for JTAG Test Resets
- the **nSRSTOUT** signal for subsystem resets.

These signals are controlled by the TRST\_OUT and SRST\_OUT bits of the CSW Register. A JTAG test reset might have to be clocked out for a number of **TCK** cycles with **TMS** HIGH to generate the reset. For more information see *Resetting JTAG devices* on page 12-7.

### 9.1.4 JTAG port signals

The signal bundle between the JTAG Port Multiplexer and each implemented JTAG port includes:

- the standard IEEE 1149.1 JTAG signals
- the non-IEEE 1149.1 extension **RTCK** signal
- additional port control and status signals.

Table 9-1 gives the full signal list. This list applies to each implemented port.

**Table 9-1 JTAG Access Port JTAG port signals**

Signal	Direction <sup>a</sup>	Description	Notes
<b>TCK</b>	Out	Test Clock	JTAG IEEE 1149.1 standard signals.
<b>TMS</b>	Out	Test Mode Select	
<b>TDI</b>	Out	Test Data In	
<b>TDO</b>	In	Test Data Out	
<b>TRST*</b>	Out	Test Reset	Active LOW JTAG IEEE 1149.1 standard signal.
<b>RTCK</b>	In	Return Clock	JTAG extension signal, not specified by IEEE 1149.1. If the connected JTAG device has no Return Clock, <b>RTCK</b> must be synthesized for the port. Implementation of the <b>RTCK</b> signal is IMPLEMENTATION DEFINED.
<b>nSRSTOUT</b>	Out	Subsystem Reset	Active LOW.
<b>SRSTCONNECTED</b>	In	Subsystem Reset Connected	Tie-off configuration signals to the JTAG Port Multiplexer.
<b>PORTCONNECTED</b>	In	Port Connected	
<b>PORTENABLED</b>	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.

## 9.2 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 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 between one and four bytes can be written to the FIFO in a single operation. See *The Byte Write FIFO Registers, BWFIFO1 to BWFIFO4* on page 12-11.

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 *The Byte Read FIFO Registers, BRFIFO1 to BRFIFO4* on page 12-11.

In the JTAG Engine Byte Command Protocol, all commands are one byte (8-bits). Table 9-2 summarizes the commands, and the following sections describe them in more detail. Where appropriate, the command descriptions also describe the TDO data produced by the command, and how this is encoded in the Byte Read FIFOs.

**Table 9-2 Summary of JTAG Engine Byte Command Protocol**

Bits of the Command byte								Opcode	For description see:
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]		
0	Opcode payload							TMS	<i>The encoding of the TMS packet.</i>
1	0	0	Opcode payload					TDI_TDO	<i>The encoding of the TDI_TDO packet on page 9-12.</i>
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	-

### 9.2.1 The encoding of the TMS packet

The TMS packet is a single byte. The payload of the packet holds:

- between one and five 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 introduction* on page 4-2.

Table 9-3 shows the possible encodings of a TMS packet.

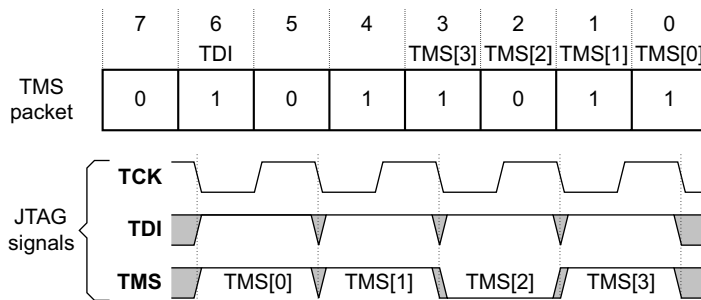
**Table 9-3 TMS packet encodings**

Command byte								Notes
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	
0	TDI	1	TMS[4]	TMS[3]	TMS[2]	TMS[1]	TMS[0]	Five bits of TMS data.
0	TDI	0	1	TMS[3]	TMS[2]	TMS[1]	TMS[0]	Four bits of TMS data.
0	TDI	0	0	1	TMS[2]	TMS[1]	TMS[0]	Three bits of TMS data.
0	TDI	0	0	0	1	TMS[1]	TMS[0]	Two bits of TMS data.
0	TDI	0	0	0	0	1	TMS[0]	One bit of TMS data.

When the JTAG Engine decodes a TMS packet, **TDI** is held at the value indicated by bit [6] while all of the TMS data bits are sent. If you have to send **TMS** bits with different **TDI** values then you must use multiple TMS packets.

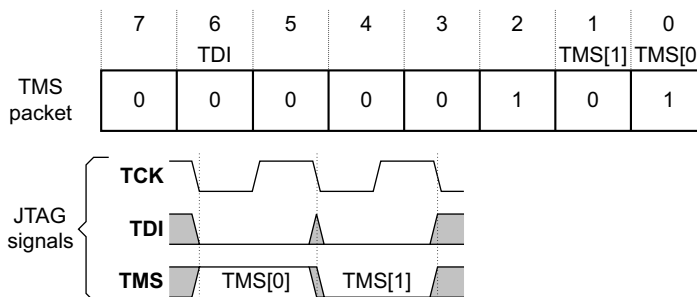
The TMS data bits are sent LSB first. This means that in each row of Table 9-3, TMS[0] is the first bit to be sent.

To send the sequence of **TMS** bits [1, followed by 1, followed by 0, followed by 1] while keeping **TDI** at 1, the TMS packet is as shown in Figure 9-3. As the diagram shows, this sequence of **TMS** signals takes four **TCK** cycles.



**Figure 9-3 TMS packet example with TDI held at 1**

To send the sequence of **TMS** bits [1, followed by 0] while keeping **TDI** at 0, the TMS packet is as shown in Figure 9-4. As shown in the diagram, this sequence of **TMS** signals takes two **TCK** cycles.



**Figure 9-4 TMS packet example with TDI held at 0**

### 9.2.2 The encoding of the TDI\_TDO packet

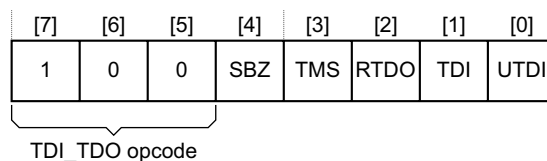
A TDI\_TDO packet is a multi-byte packet that is at least two bytes long. It comprises:

- the TDI\_TDO opcode byte
- a second byte, that contains:
  - for short packets, of fewer than seven TDI bits, the packed TDI bits
  - otherwise, the length of the packet
- if required, between one and sixteen 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 sub-type. Figure 9-5 shows the format of this byte.



**Figure 9-5 TDI\_TDO first byte (opcode) format**

Bits [3:0] are control bits that define the TDI\_TDO sub-type. Table 9-4 on page 9-13 describes all the bits of the TDI\_TDO packet first byte.

Table 9-4 TDI\_TDO first byte (opcode) format

Bit	Name	Value <sup>a</sup>	Description
[7]	TDI_TDO	1	The value of these bits indicates that this is the first byte of a TDI_TDO packet
[6]		0	
[5]		0	
[4]	-	SBZ	Reserved, Should Be Zero.
[3]	TMS	-	<p><b>TMS</b> value to use on the last cycle of the scan:  0 = <b>TMS</b> LOW on last cycle  1 = <b>TMS</b> HIGH on last cycle.  <b>TMS</b> is always LOW on all the earlier cycles of the scan.</p>
[2]	RTDO	-	<p>Read <b>TDO</b>. This bit determines whether <b>TDO</b> values returned during the scan are captured and placed in the Response FIFO:  0 = Do not capture <b>TDO</b>  1 = Capture <b>TDO</b>.</p> <p>———— <b>Caution</b> ————</p> <p>Do not set this bit to 1 if more than one JTAG port is selected and enabled. If you do, the <b>TDO</b> values captured are UNPREDICTABLE.</p>
[1]	TDI	-	<p><b>TDI</b> value to use throughout the scan if UTDI bit is set to 1:  0 = hold <b>TDI</b> signal LOW throughout the scan  1 = hold <b>TDI</b> signal HIGH throughout the scan  The value of the TDI bit is ignored if UTDI is set to 0.</p>
[0]	UTDI	-	<p>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 <b>TDI</b> signal to use throughout the scan:  0 = TDI bits for the scan are supplied in the other bytes of the TDI_TDO packet.  1 = The TDI bit, bit [1], determines the <b>TDI</b> signal to use throughout the scan.  When this bit is set to 1, no TDI data is included in the TDI_TDO packet<sup>b</sup>.</p>

- Given for bits that have a fixed value for the TDI\_TDO first byte.
- When the Packed format is used for the second byte of the packet certain bits of that byte are designated as TDI data bits, however the value of these bits is ignored if UTI = 1, see *The TDI\_TDO length byte, the second byte of the packet* on page 9-14. There is no advantage in using the packed format when UTDI = 1, but it is possible to do so.

## 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** In this format, this byte specifies the length of the scan. This can be any value between 1 and 128 bits.

This format is described in *The normal format of the TDI\_TDO length byte*.

**Packed** The packed format can be used when the required scan is between one and six cycles long. In this format, this byte contains between one and six bits of TDI data. The length of the scan is determined by the number of bits of data provided.

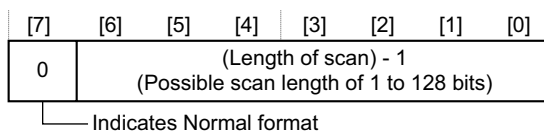
This format is described in *The packed format of the TDI\_TDO length byte* on page 9-15.

### Note

This format can be used when the UTDI bit in the first byte of the packet is set to 1. However, there is no advantage in using the packed format when UTDI = 1, because the normal format is easier to use, and the TDI\_TDO packet is two bytes long whichever format is used.

### The normal format of the TDI\_TDO length byte

If bit [7] of the second byte of the TDI\_TDO packet is zero, the byte is in the normal length byte format. In this format, bits [6:0] of the byte give the length in bits of the required scan, minus one. This format is shown in Figure 9-6.



**Figure 9-6 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 0, the TDI data for the scan is packed into additional bytes of the packet, that follow the length byte. See *The data bytes, the third and subsequent bytes of the packet* on page 9-16 for more information.
- If the UTDI bit of the first byte of the TDI\_TDO packet is 1 then no TDI data is required for the scan, and the length byte is the last byte of the packet. Whenever the UTDI bit is set to 1, the TDI\_TDO packet is always two bytes long.

See *The TDI\_TDO opcode byte, the first byte of the packet* on page 9-12 for more information about the UTDI bit.



**The packed format of the TDI\_TDO length byte**

If bit [7] of the second byte of the TDI\_TDO packet is one, the byte is in the packed length byte format. In this format:

- the length of the required scan is implied by the data in bits [6:0] of the length byte, the second byte of the TDI\_TDO packet
- if the UTDI bit of the first byte of the TDI\_TDO packet is 0, 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 two bytes long.

**Note**

The packed format of the TDI\_TDO length byte can only be used if the required scan is of six bits or less.

Figure 9-7 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 9-7 TDI\_TDO second byte (length byte), packed format**

The packed format for the TDI\_TDO second byte is summarized in Table 9-5 on page 9-16.

**Table 9-5 TDI\_TDO second byte (length byte), packed format**

Scan length	Must be zero bits	Data start flag	TDI data for scan <sup>a</sup>
6 bits	None	Bit [6] = 1	Bits [5:0]
5 bits	Bit [6] = 0	Bit [5] = 1	Bits [4:0]
4 bits	Bits [6:5] = b00	Bit [4] = 1	Bits [3:0]
3 bits	Bits [6:4] = b000	Bit [3] = 1	Bits [2:0]
2 bits	Bits [6:3] = b0000	Bit [2] = 1	Bits [1:0]
1 bit	Bits [6:2] = b00000	Bit [1] = 1	Bit [0]

a. When the UTDI bit of the first byte of the TDI\_TDO packet is 1, 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 0, the data 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 1, the data 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 b10001XXX, referring to Figure 9-7 on page 9-15 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 9-3 on page 9-11.

See *The TDI\_TDO opcode byte, the first byte of the packet* on page 9-12 for more information about the UTDI bit.

### The data bytes, the third and subsequent bytes of the packet

The TDI\_TDO packet is more than two bytes long only when both of the following are true:

- The length byte, the second byte of the packet, is in the normal format. See *The normal format of the TDI\_TDO length byte* on page 9-14.
- the UTDI bit of the first byte of the packet is 0. See *The TDI\_TDO opcode byte, the first byte of the packet* on page 9-12.

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 additional 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. This 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 set to 0. For example, a scan of 21 cycles requires three data bytes, giving a total TDI\_TDO packet size of five bytes. Figure 9-8 shows the formatting of the complete TDI\_TDO packet for this example.

	[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 9-8 TDI\_TDO formatting example. Complete packet for a scan of 21 TCK cycles**

In Figure 9-8:

**Byte 1, the opcode byte**

- Bits [7:5] are the TDI\_TDO opcode, b100.
- Bit [3] is the TMS bit. The value of 1 indicates that **TMS** must be HIGH for the last cycle of the scan.
- Bit [2] is the RTDO bit. The value of 1 indicates that TDO data must be captured during the scan.

**Byte 2, the length byte**

- Bit [7] = 0 indicates that this length byte is in normal format.
- Bits [6:0] give the value ((length of scan) - 1). This field has the value b0010100, 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 set to 0, so bits [7:5] = b000.

### 9.2.3 Response bytes from a TDI\_TDO packet

If the Read TDO (RTDO) bit, bit [2], of a TDI\_TDO packet header is set to 1, then 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 as soon as it is completed.

Figure 9-8 on page 9-17 shows a TDI\_TDO packet with RTDO = 1.

#### Caution

If more than one JTAG port is selected and enabled, the returned TDO values are UNPREDICTABLE.

The number of bytes of TDO data inserted in the Response FIFO is the scan length divided by eight, rounded up to an integer value. When the scan length is not an exact multiple of 8, zero bits are inserted to pack the last byte of returned data.

The scan stalls if the Response FIFO is full when a byte of TDO data is ready for insertion.

Figure 9-9 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.

	[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 9-9 TDI\_TDO response data formatting example. Scan of 21 TCK cycles**

If the RTDO bit is set to 0 then no response bytes are placed in the Response FIFO.

See *The TDI\_TDO opcode byte, the first byte of the packet* on page 9-12 for details of the Read TDO (RTDO) bit.

# Chapter 10

## Access Port (AP) Registers Overview and the Common AP Register

An ARM Debug Interface can include multiple Access Ports. ARM Limited provides two AP definitions, but other designers might implement additional APs. This chapter describes the required features of the register implementation for any AP. It contains the following sections:

- *Access Port (AP) registers overview* on page 10-2
- *The common Access Port register, the IDR* on page 10-3.

Descriptions of the other AP registers are given in:

- Chapter 11 *Memory Access Port (MEM-AP) Registers*
- Chapter 12 *JTAG Access Port (JTAG-AP) Registers*.

For a full description of an AP, see the following chapters:

- Chapter 8 *The Memory Access Port (MEM-AP)*, for the Memory AP
- Chapter 9 *The JTAG Access Port (JTAG-AP)*, for the JTAG AP.

In Chapter 8 and Chapter 9, the title page of the chapter lists the other chapters that contain relevant information.

## 10.1 Access Port (AP) registers overview

There are two types of Access Port defined by the ADIV5 specification:

- the Memory Access Port (MEM-AP)
- the JTAG Access Port (JTAG-AP).

An ARM Debug Interface might have multiple Access Ports, and these can be a mixture of both types.

### ————— Note —————

This Architecture Specification permits an ARM Debug Interface to include additional Access Port types. Such an AP must implement the common AP register described in this chapter. Debuggers must be able to recognize and, if necessary, ignore APs other than the MEM-AP and JTAG-AP.

There is one common register that must always be implemented by any AP. Also, the programmer's model for accessing AP registers is the same for all APs. These common features of the AP registers implementation are described in this chapter.

### 10.1.1 The Programmer's Model for Access Port (AP) registers

An *Access Port* (AP) implements 32-bit registers. These are mapped into a 256-byte address space. *Accessing Access Ports* on page 2-8 summarized the programmer's model for accessing these registers. Also:

- Figure 8-1 on page 8-4 shows the organization of the registers for a MEM-AP
- Figure 9-1 on page 9-3 shows the organization of the registers for a JTAG-AP.

The ARM Debug Interface Architecture Specification requires every AP to implement one register within this space, and that is the AP Identification Register, IDR, at offset 0xFC. This is the last register in the AP register space, and is described in *The common Access Port register, the IDR* on page 10-3.

Additional information about the AP programmer's model is given in the chapters that describe the two types of AP defined by this specification:

- In Chapter 7 *Common Access Port (AP) features*, see *Selecting and accessing an AP* on page 7-4. This description applies to both MEM-APs and JTAG-APs.
- In Chapter 8 *The Memory Access Port (MEM-AP)*, see *The programmer's model for debug register access* on page 8-2.

### 10.1.2 Accesses to Reserved addresses

There are a number of Reserved addresses within the AP register maps, for both MEM-APs and JTAG-APs. With DAP accesses:

- reads of Reserved addresses return zero (RAZ)
- writes to Reserved addresses are ignored (WI).

This behavior of accesses to Reserved addresses is a requirement of the ADIV5 specification and applies to all APs, including any implemented by companies other than ARM Limited.

## 10.2 The common Access Port register, the IDR

This section summarizes the IDR, the register that must be implemented by all Access Ports. This requirement applies to APs defined by ARM Limited:

- Memory Access Ports, MEM-APs
- JTAG Access Ports, JTAG-APs.

In addition, the ARM Debug Interface Architecture Specification permits additional APs to be defined, and any such AP must comply with the register requirements given in this chapter.

The additional register requirements for MEM-APs and a JTAG-APs are described in:

- Chapter 11 *Memory Access Port (MEM-AP) Registers*
- Chapter 12 *JTAG Access Port (JTAG-AP) Registers*.

### 10.2.1 Summary of the IDR

The IDR is the only register that must be implemented by any Access Port. This is summarized in Table 10-1.

**Table 10-1 Summary of the common Access Port (AP) register**

Address	Name	Access	Description	Reset value
0xFC	IDR	RO	AP Identification Register	IMPLEMENTATION DEFINED

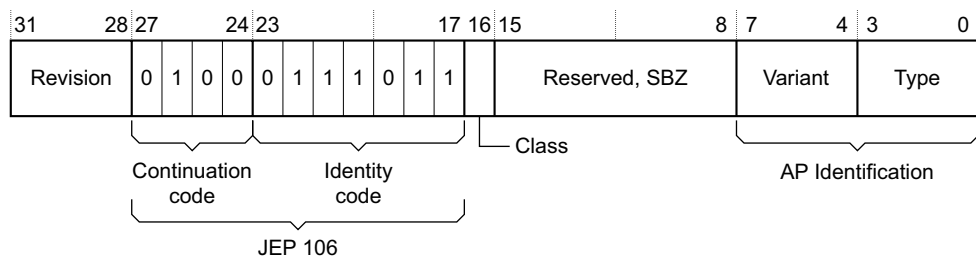
The register is described in detail in *The Identification Register, IDR*.

### 10.2.2 The Identification Register, IDR

The Identification Register identifies the Access Port. It is a read-only register, implemented in the last word of the AP register space, at offset 0xFC.

An IDR of zero indicates that no AP is present.

Figure 10-1 shows the register bit assignments for an implementation of an Access Port by ARM Limited.



**Figure 10-1 The AP Identification Register, IDR, for an ARM Limited AP implementation**

Table 10-2 lists the bit functions of the AP Identification Register.

**Table 10-2 AP Identification Register, IDR, bit assignments**

Bits	Function	Description
[31:28]	Revision	Starts at 0x0 for the first implementation of an AP design, and increments by 1 on each major or minor revision of the design <sup>a</sup> .
[27:24]	JEP-106 continuation code	JEP-106 continuation code, used to identify the <i>designer</i> of the AP, see <i>JEP-106 identity and continuation codes</i> . For an AP designed by ARM Limited this field has the value b0100, 0x4.
[23:17]	JEP106 identity code	JEP-106 identity code, used to identify the <i>designer</i> of the AP, see <i>JEP-106 identity and continuation codes</i> . For an AP designed by ARM Limited this field has the value b0111011, 0x3B.
[16]	Class	This flag can have the following values: 0: This AP is not a Memory Access Port 1: This AP is a Memory Access Port. A Memory AP must implement the registers specified in Chapter 11 <i>Memory Access Port (MEM-AP) Registers</i> , and might implement additional registers.
[15:8]	-	Reserved, SBZ.
[7:0]	AP Identification	This field identifies the AP implementation. Each AP designer must maintain their own list of implementations and associated AP Identification codes. In an AP implementation by ARM Limited this field is subdivided as: <b>[7:4] Variant</b> A field to distinguish between different APs that connect to the same bus type, or to the same connection type for non-bussed connections. <b>[3:0] Type</b> Indicates the type of bus, or other connection, that connects to the AP. See <i>ARM AP Identification types</i> on page 10-5.

a. Major design revisions introduce functionality changes, minor revisions are bug fixes.

## JEP-106 identity and continuation codes

The JEP codes are assigned by JEDEC and are usually used to identify the *manufacturer* of a device. However, in the AP Identification Register they are used to identify the *designer* of the AP. They are used in this way, also, in the DP IDCODE Register, see *The Identification Code Register, IDCODE* on page 6-8 for more information.

If you implement an ARM MEM-AP or JTAG-AP you must not change the AP Identification Register values from those given in Table 10-2.



---

**Note**

---

For backwards compatibility, debuggers must treat a JEP 106 field of zero as indicating an AP designed by ARM Limited. This encoding was used in early implementations of the Debug Access Port. In such an implementation, the Revision and Class fields also Read As Zero.

DAP implementations that comply with the ADIv5 specification must use the JEP 106 code and include Revision and Class fields.

---

**ARM AP Identification types**

Table 10-3 lists the possible values of the Type field for an AP designed by ARM Limited. It also shows the value of the Class bit that corresponds to each Type value.

**Table 10-3 ARM AP Identification types**

Type <sup>a</sup>	Connection to AP	Class <sup>b</sup>	Notes
0x0	JTAG connection	0	Variant field, bits [7:4] of IDR, must be non-zero.
0x1	AMBA AHB bus	1	
0x2	AMBA APB bus	1	
Other	Reserved	-	

a. Bits [3:0] of the IDR

b. Bit [16] of the IDR.



# Chapter 11

## Memory Access Port (MEM-AP) Registers

This chapter describes the *Memory Access Port* (MEM-AP) registers, and how they provide the Debug Port connection to a debug component. It contains the following sections:

- *Memory Access Port (MEM-AP) register summary* on page 11-2
- *MEM-AP detailed register descriptions* on page 11-5.

A MEM-AP must also implement the Identification Register, IDR, that is described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

For a full description of a MEM-AP, see Chapter 8 *The Memory Access Port (MEM-AP)*. The title page of Chapter 8 lists the other chapters that contain relevant information.

## 11.1 Memory Access Port (MEM-AP) register summary

This section:

- Lists all the MEM-AP registers, with links to the detailed description of each register. See *Summary of MEM-AP registers*.
- Gives some general information about the MEM-AP registers, see *Features of the MEM-AP registers* on page 11-4.

### 11.1.1 Summary of MEM-AP registers

Table 11-1 summarizes the MEM-AP registers, and indicates where they are described in detail. This table shows the memory map of the MEM-AP registers, and includes the Identification Register that is described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

All of the registers listed in Table 11-1 are required in every MEM-AP implementation.

**Table 11-1 Summary of Memory Access Port (MEM-AP) registers**

Address <sup>a</sup>	Access	Reset value	For description see
0x00	R/W	See register description	<i>Control/Status Word (CSW) Register</i> on page 11-5
0x04	R/W	UNPREDICTABLE	<i>The Transfer Address Register (TAR)</i> on page 11-7
0x08	RAZ/WI	-	Reserved
0x0C	R/W	Not applicable <sup>b</sup>	<i>The Data Read/Write Register (DRW)</i> on page 11-8
0x10	R/W	Not applicable <sup>b</sup>	<i>Banked Data Registers 0 to 3 (BD0 to BD3)</i> on page 11-9
0x14	R/W		
0x18	R/W		
0x1C	R/W		
0x20 - 0xF0	RAZ/WI	-	Reserved
0xF4	RO	IMPLEMENTATION DEFINED	<i>Configuration Register (CFG)</i> on page 11-10
0xF8	RO	IMPLEMENTATION DEFINED <sup>c</sup>	<i>Debug Base Address Register (BASE)</i> on page 11-11
0xFC	RO	IMPLEMENTATION DEFINED <sup>c</sup>	<i>The Identification Register, IDR</i> on page 10-3

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

b. These registers cannot be read until the memory access has completed. Therefore they do not have reset values.

c. The register descriptions give details of the reset values of *some* bits of these registers.

Reserved addresses in the register memory map Read As Zero, and writes to these addresses are ignored.

As explained in *Accessing Access Ports* on page 2-8, AP registers are accessed by specifying a four-word register bank within the AP register space, and then selecting the required register within that bank:

- the bank in the AP register space is selected by the APBANKSEL field in the Debug Port (DP) SELECT Register, see *The AP Select Register, SELECT* on page 6-15
- the register to access within the selected bank is selected by the A[3:2] field in the APACC or DPACC:
  - for accesses to a JTAG-DP, see *The JTAG-DP DP and AP Access Registers (DPACC and APACC)* on page 4-14
  - for accesses to a SW-DP, see *Serial Wire Debug protocol operation* on page 5-5.

The A[3:2] field of the APACC or DPACC specifies bits [3:2] of the register address. Bits [1:0] of the register address are always b00.

Table 11-2 shows the register bank and register offset within the bank for each of the defined MEM-AP registers.

**Table 11-2 Access information for the MEM-AP registers**

<b>MEM-AP register</b>	<b>Address</b>	<b>Register bank (APBANKSEL<sup>a</sup>)</b>	<b>Offset (A[3:2]<sup>b</sup>)</b>
CSW, Control/Status Word Register	0x00	0x0	0, b00
TAR, Transfer Address Register	0x04	0x0	1, b01
DRW, Data Read/Write Register	0x0C	0x0	3, b11
BD0, Banked Data Register 0	0x10	0x1	0, b00
BD1, Banked Data Register 1	0x14	0x1	1, b01
BD2, Banked Data Register 2	0x18	0x1	2, b10
BD3, Banked Data Register 3	0x1C	0x1	3, b11
CFG, Configuration Register	0xF4	0xF	1, b01
BASE, Base Address Register	0xF8	0xF	2, b10
IDR, Identification Register	0xFC	0xF	3, b11

a. In the DP SELECT Register, see *The AP Select Register, SELECT* on page 6-15.

b. In the APACC or DPACC:

for accesses to a JTAG-DP, see *The JTAG-DP DP and AP Access Registers (DPACC and APACC)* on page 4-14

for accesses to a SW-DP, see *Serial Wire Debug protocol operation* on page 5-5.

Accesses to other combinations of Register bank and Offset Read As Zero, and are ignored on writes.

---

**Note**

---

This chapter describes the registers that must be implemented by any MEM-AP. However, an AP might implement additional features, including additional registers.

---

### 11.1.2 Features of the MEM-AP registers

The Memory Access Port (MEM-AP) provides access to a memory subsystem, through the DAP. All accesses to a MEM-AP are made through the MEM-AP registers. However, only accesses to the Data Read/Write and Banked Data Registers initiate accesses to the connected memory system. For more details, see *MEM-AP register accesses and memory accesses* on page 8-6.

## 11.2 MEM-AP detailed register descriptions

This section describes each of the required MEM-AP registers. These registers are listed in Table 11-1 on page 11-2. That table indexes the detailed register descriptions in this section. The registers are described in the following sections:

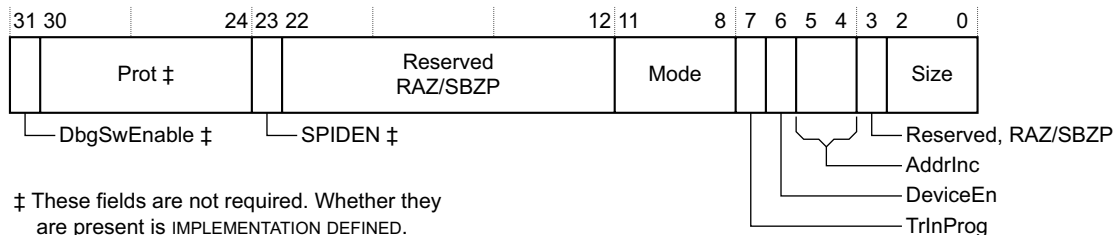
- *Control/Status Word (CSW) Register*
- *The Transfer Address Register (TAR)* on page 11-7
- *The Data Read/Write Register (DRW)* on page 11-8
- *Banked Data Registers 0 to 3 (BD0 to BD3)* on page 11-9
- *Configuration Register (CFG)* on page 11-10
- *Debug Base Address Register (BASE)* on page 11-11.

### 11.2.1 Control/Status Word (CSW) Register

The CSW Register configures and controls accesses through the MEM-AP to or from a connected memory system. The CSW is:

- At offset 0x00 in the MEM-AP register space. This means it is the first register in the first register bank of the MEM-AP register space.
- A read/write register, although some bits of the register are read-only.

Figure 11-1 shows the register bit assignments.



**Figure 11-1 MEM-AP Control/Status Word Register, CSW, bit assignments**

Table 11-3 lists the bit assignments for the CSW, and indicates which fields are required.

**Table 11-3 Bit assignments for the MEM-AP Control/Status Word Register, CSW**

Bits	Function	Access	Required?	Description
[31]	DbgSwEnable <sup>a</sup>	R/W <sup>a</sup>	No	Debug software access enable. The use of this flag is IMPLEMENTATION DEFINED, see <i>Slave memory port and software access control</i> on page 8-19.
[30:24]	Prot <sup>a</sup>	R/W <sup>a</sup>	No	Bus access protection control. This field enables the debugger to specify protection flags for a debug access. The permitted values and their significance are IMPLEMENTATION DEFINED, because they relate to the underlying bus architecture. See <i>Additional implementation defined features of a MEM-AP</i> on page 8-20 for more information.
[23]	SPIDEN <sup>a</sup>	RO <sup>a</sup>	No	Secure Privileged Debug Enabled. If this flag is implemented, the possible values are: 0: Secure Privileged Debug disabled 1: Secure Privileged Debug enabled.
[22:12]	-	-	-	Reserved. RAZ/SPZP
[11:8]	Mode	R/W	Yes	Mode of operation. These bits must be set to 0x0. All other values are Reserved. The reset value of this field is UNPREDICTABLE.
[7]	TrInProg	RO	Yes	Transfer in progress. This bit is set to 1 while a transfer is in progress on the connection to the memory system, and is clear (0) when the connection is idle. After an ABORT operation, debug software can read this bit to check whether the aborted transaction completed.
[6]	DeviceEn	RO	Yes	Device enabled. This bit is set to 1 when transactions can be issued through the MEM-AP. This bit shows the value of the <b>DEVICEEN</b> signal, that is a control input to the DAP. If <b>DEVICEEN</b> is not implemented this bit reads as one. For more information, see <i>Enabling access to the connected debug device or memory system</i> on page 8-8.



**Table 11-3 Bit assignments for the MEM-AP Control/Status Word Register, CSW (continued)**

Bits	Function	Access	Required?	Description
[5:4]	AddrInc	R/W	Yes	Address auto-increment and packing mode. This field controls whether the access address increments automatically on read and write data accesses through the Data Read/Write Register. For more information see <i>Auto-incrementing the Transfer Address Register (TAR)</i> on page 8-9. The reset value of this field is UNPREDICTABLE.
[3]	-	-	-	Reserved. RAZ/SPZP.
[2:0]	Size <sup>b</sup>	R/W <sup>b</sup>	Yes	Size of the access to perform. See <i>Variable access size for memory accesses</i> on page 8-13 The reset value of this field is UNPREDICTABLE.
		RO <sup>b</sup>		Reads as b010 to indicate that only 32-bit accesses are supported.

- a. It is IMPLEMENTATION DEFINED whether a MEM-AP supports the features controlled by these fields. Where a feature is not supported the corresponding field is Reserved, RAZ/SBZP. See the field description for more information.
- b. It is IMPLEMENTATION DEFINED whether a MEM-AP supports access sizes other than 32-bit. If it does then the Size field is R/W. If it does not then it is RO. Both cases are described in the table.

## 11.2.2 The Transfer Address Register (TAR)

The TAR holds the memory address to be accessed. The TAR is:

- At offset 0x04 in the MEM-AP register space. This means it is the second register in the first register bank of the MEM-AP register space.
- A read/write register.

The TAR holds a 32-bit address:

- When using the Data Read/Write Register (DRW), TAR[31:0] specifies the memory address to access:
  - if the MEM-AP does not support accesses smaller than word then TAR[1:0] can be implemented as Read As Zero
  - the contents of the TAR can be incremented automatically on a successful DRW access, see *Auto-incrementing the Transfer Address Register (TAR)* on page 8-9.
- When using the Banked Data Registers, BD0 to BD3, TAR[31:4] specifies the base address of the 16-byte block of memory that can be accessed through BD0 to BD3, see *Banked Data Registers 0 to 3 (BD0 to BD3)* on page 11-9 for more information.

The address held in the TAR represents an address in the memory system to which the MEM-AP is connected. It is *not* an address within the MEM-AP

Table 11-4 lists the bit assignments for the TAR.

**Table 11-4 Bit assignments for the Transfer Address Register, TAR**

Bits	Function	Access	Description
[31:0]	Address	R/W	Memory address for the current transfer. When accessing memory through the Banked Data Registers, BD0 to BD3, bits [3:0] of this address are ignored.

### 11.2.3 The Data Read/Write Register (DRW)

The DRW Register maps an AP access directly to one or more memory accesses. The AP access does not complete until the memory access, or accesses, complete. The DRW is:

- At offset 0x0C in the MEM-AP register space. This means it is the fourth register in the first register bank of the MEM-AP register space.
- A read/write register.

The DRW holds a 32-bit data value:

- in write mode, the DRW holds the value to write for the current transfer to the address specified in TAR[31:0]
- in read mode, the DRW holds the value read in the current transfer from the address specified in TAR[31:0].

Table 11-5 lists the bit assignments for the DRW.

**Table 11-5 Bit assignments for the Data Read/Write Register, DRW**

Bits	Function	Access	Description
[31:0]	Data	R/W	Data value of the current transfer.

If the MEM-AP supports data transfers of less than 32-bits then a single APACC access to the DRW might result in multiple memory accesses, depending on the value of the Size field in the CSW, see *Variable access size for memory accesses* on page 8-13. These accesses are byte-laned, as described in *Endianness and byte lanes* on page 8-14.

For more information about the use of the CSW, see:

- *Endianness and byte lanes* on page 8-14
- *Packed transfers* on page 8-16
- *Auto-incrementing the Transfer Address Register (TAR)* on page 8-9.

### 11.2.4 Banked Data Registers 0 to 3 (BD0 to BD3)

The Banked Data Registers map APACC transactions directly to memory accesses, without having to change the value in the TAR. Together, the four Banked Data Registers give access to four words of the memory space, starting at the address specified in the TAR.

The Banked Data Registers are:

- At offsets 0x10, 0x14, 0x18, and 0x1C in the MEM-AP register space. This means they are the four registers in the second register bank of the MEM-AP register space. BD0 is the first register in this bank, and BD3 is the last register in the bank.
- Read/write registers.

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 read from memory.

Table 11-6 shows how the four Banked Data Registers map onto the address space. Memory accesses through the Banked Data Registers are always 32-bit, so bits [1:0] of the memory address are always b00.

**Table 11-6 Mapping of Banked Data Registers onto memory addresses**

Register	Offset	Memory address accessed
BD0	0x10	TAR[31:4] << 4
BD1	0x14	(TAR[31:4] << 4) + 0x4
BD2	0x18	(TAR[31:4] << 4) + 0x8
BD3	0x1C	(TAR[31:4] << 4) + 0xC

Table 11-7 lists the bit assignments for a Banked Data Register.

**Table 11-7 Bit assignments for the Banked Data Registers, BD0 to BD3**

Bits	Function	Access	Description
[31:0]	Banked Data	R/W	Data value of the current transfer.

An access to a Banked Data Register initiates an access to the memory address shown in Table 11-6. The AP access does not complete until the memory access has completed. Auto address incrementing is not performed when a Banked Data Register is accessed. The value of the AddrInc field in the CSW has no effect on Banked Data Register accesses.

### Caution

If the MEM-AP supports data transfers of less than 32-bits then the Size field of the CSW, CSW[2:0], must be set to b010 before accessing the Banked Data Registers. This configures the MEM-AP for word (32-bit) memory accesses. Behavior is UNPREDICTABLE if a Banked Data Register is accessed with the Size field set to any other value.

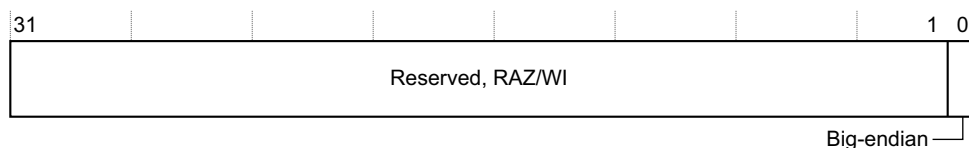
### 11.2.5 Configuration Register (CFG)

The CFG Register provides information about the configuration of the MEM-AP implementation.

The Configuration Register is:

- At offset 0xF4 in the MEM-AP register space. This means it is the second register in the last register bank of the MEM-AP register space, bank 0xF.
- A read-only register.

Figure 11-2 shows the CFG Register bit assignments.



### Figure 11-2 Configuration Register, CFG, bit assignments

Table 11-8 lists the bit assignments for the CFG Register.

### Table 11-8 Bit assignments for the Configuration Register, CFG,

Bits	Function	Access	Description
[31:1]	-	-	Reserved, RAZ/WI.
0	Big-endian	RO	<p>This field indicates whether memory accesses by the MEM-AP are big-endian or little-endian:</p> <p>0: Little-endian</p> <p>1: Big-endian.</p> <p>The value of this field is IMPLEMENTATION DEFINED.</p>

For more information about the endianness of memory accesses, see *Endianness and byte lanes* on page 8-14.



**Table 11-9 Bit assignments for the Debug Base Address Register, BASE,**

Bits	Function	Access	Description
[31:12]	BASEADDR	RO	Bits [31:12] of the address offset, in the memory-mapped resource, of the start of the debug register space or a ROM table address. See <i>The BASEADDR field, BASE[31:12]</i> . Bits [11:0] of the address offset are 0x000.
[11:2]	-	-	Reserved, RAZ.
1	Format	RO	Base address register format. This bit must read as 1, to indicate ARM Debug Interface v5 format. <sup>a</sup>
0	Entry present	RO	This field indicates whether a debug entry is present for this MEM-AP: 0: No debug entry present <sup>b</sup> 1: Debug entry present.

- a. This bit is Read As Zero in one of the legacy Debug Base Address Register formats, see *Legacy formats of the Debug Base Address Register* on page 11-14.
- b. *Legacy formats of the Debug Base Address Register* on page 11-14 includes a description of the legacy format of the BASE register when there is no debug entry present.

### The BASEADDR field, BASE[31:12]

The BASEADDR field, BASE[31:12], contains bits [31:12] of the base address, in the debug component memory map, of the description of the debug component or components to which the ARM Debug Interface is connected. The least significant bits of this address are 0, so the actual base address is (BASEADDR << 12).

The details of the memory area pointed to by this base address depend on the number of debug components connected to the ARM Debug Interface:

- If the ARM Debug Interface is connected to a single debug component, as in the system shown in Figure 1-3 on page 1-8, the base address is the base address of that component, and points to the start of the debug registers for that component.  
A debug component can occupy more than one 4KB page of memory. If it does, the base address points to the *final* 4KB page for the component.
- If the ARM Debug Interface is connected to more than one debug component, as in the system shown in Figure 1-5 on page 1-9, then the base address must point to a ROM table. This ROM table gives the addresses of the other debug components connected to the interface. ROM tables are described in Chapter 14 *ROM Tables*.  
A simple system, that contains only a single debug component, might be implemented with a separate ROM table, as shown in Figure 2-1 on page 2-7. In this case the base address points to the ROM table.

### ***Debugger issues***

A debugger can always examine the four words starting at offset 0xFF0 from the base address, to discover information about the debug components connected to the MEM-AP. That is, it can check the four words starting at  $((\text{BASEADDR} \ll 12) + 0xFF0)$  for this information. This address points to the four Component ID registers, either for the single debug component or for the ROM table, with Component ID Register 0 at offset 0xFF0. Reading the DRW register with  $((\text{BASEADDR} \ll 12) + 0xFF0)$  in the TAR returns the value of this register. For more information about these registers see *The Component ID Registers* on page 13-4.

The debugger can interpret the returned information to find the *type* of component connected. The component type is one of:

- ROM table
- debug component
- other.

The ARM Debug Interface v5 architecture specification does not mandate the type of component pointed to by the BASE register.

A debugger should handle the following situations as non-fatal errors:

- the base address,  $(\text{BASEADDR} \ll 12)$ , is a faulting location
- the four words starting at  $((\text{BASEADDR} \ll 12) + 0xFF0)$  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 starting at offset 0xFF0.

Typically, a debugger issues a warning if it encounters one of these situations. However it should continue operating. An example of an implementation that might cause errors of this type is a system with static  $(\text{BASEADDR} \ll 12)$  or ROM table entries that enable entire subsystems to be disabled, for example by a tie-off input, packaging choice, fuse or similar.

### ***Debug component address map segmentation***

If a 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. This enables the device to distinguish two types of access:

- debugger-initiated accesses address the logical segment with  $\text{TAR}[31]=1$
- system-initiated accesses, if permitted, address the logical segment with  $\text{TAR}[31]=0$ .

With such an implementation,  $\text{BASEADDR}[31]$  must be set to 1.

Even where system-initiated accesses are not permitted, ARM Limited recommends that the debug component address space is segmented in this way, and that debugger-initiated accesses have  $\text{TAR}[31]=1$ .

Other systems might include IMPLEMENTATION DEFINED methods for signaling debugger accesses to system components.

The section *Legacy formats of the Debug Base Address Register* is also important for debugger implementation.

### Legacy formats of the Debug Base Address Register

There are two legacy formats of the Debug Base Address Register that a debugger should recognize. These are described in the following subsections:

- *Legacy format when no debug entries are present*
- *Legacy format for specifying BASEADDR.*

#### ———— Note —————

These formats must not be used for new ARM Debug Interface designs.

#### ***Legacy format when no debug entries are present***

Table 11-10 shows the legacy format of the Debug Base Address Register when there are no debug entries present

**Table 11-10 Legacy Debug Base Address Register format when there are no debug entries**

Bits	Function	Access	Value	Description
[31:0]	NOTPRESENT	RO	0xFFFFFFFF	No debug entries present.

#### ***Legacy format for specifying BASEADDR***

If bit [1] of the Debug Base Address Register is 0 then the legacy format of the register is being used to hold the base address value. This format is shown in Table 11-11

**Table 11-11 Legacy Debug Base Address Register format when holding a base address value**

Bits	Function	Access	Value	Description
[31:12]	BASEADDR	RO	IMPLEMENTATION DEFINED	Debug base address bits [31:12]. See <i>The BASEADDR field, BASE[31:12]</i> on page 11-12
[11:0]	-	-	0x000	RAZ.



# Chapter 12

## JTAG Access Port (JTAG-AP) Registers

This chapter describes the register interface to the *JTAG Access Port (JTAG-AP)*. It contains the following sections:

- *JTAG Access Port (JTAG-AP) register summary* on page 12-2
- *JTAG-AP detailed register descriptions* on page 12-5.

A JTAG-AP must also implement the Identification Register, IDR, described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

For a full description of a JTAG-AP, see Chapter 9 *The JTAG Access Port (JTAG-AP)*. The title page of Chapter 9 lists the other chapters that contain relevant information.

## 12.1 JTAG Access Port (JTAG-AP) register summary

This section lists all the JTAG-AP registers, with links to the detailed description of each register.

The section *The JTAG Engine Byte Command Protocol* on page 9-10:

- describes the byte protocol used for encoding JTAG commands
- describes the command responses
- gives additional information about the use of the Byte FIFO Registers.

### 12.1.1 Summary of JTAG-AP registers

Table 12-1 lists the JTAG-AP registers, and indicates where they are described in detail. This table shows the memory map of the JTAG-AP registers, and includes the Identification Register that is described in Chapter 10 *Access Port (AP) Registers Overview and the Common AP Register*.

All of the registers listed in Table 12-1 are required in every JTAG-AP implementation.

**Table 12-1 Summary of JTAG Access Port (JTAG-AP) registers**

Address	Access	Reset value	Description, see
0x00	R/W	a	<i>The Control/Status Word Register, CSW</i> on page 12-5
0x04	R/W	UNPREDICTABLE	<i>The Port Select Register, PSEL</i> on page 12-7
0x08	R/W	0x0000 0000	<i>The Port Status Register, PSTA</i> on page 12-9
0x0C	RAZ/WI	-	Reserved
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

*The Byte FIFO registers, BRFIFO1 to BRFIFO4 and BWFIFO1 to BWFIFO4* on page 12-10

**Table 12-1 Summary of JTAG Access Port (JTAG-AP) registers (continued)**

Address	Access	Reset value	Description, see
0x20 - 0xF8	RAZ/WI	-	Reserved
0xFC	RO	a	<i>The Identification Register, IDR</i> on page 10-3

- a. See the register description.  
b. Accesses to Byte FIFO Read Registers stall until data is available in the FIFO. Therefore they do not have reset values.

Reserved addresses in the register memory map Read As Zero, and writes to these addresses are ignored.

As explained in *Accessing Access Ports* on page 2-8, AP registers are accessed by specifying a four-word register bank in the AP register space, and then selecting the required register within that bank:

- the register bank of the AP register space is selected by the APBANKSEL field in the Debug Port (DP) SELECT Register, see *The AP Select Register, SELECT* on page 6-15
- the register to access within the selected bank is selected by the A[3:2] field in the APACC or DPACC:
  - for accesses to a JTAG-DP, see *The JTAG-DP DP and AP Access Registers (DPACC and APACC)* on page 4-14
  - for accesses to a SW-DP, see *Serial Wire Debug protocol operation* on page 5-5.

Table 12-2 shows the register bank and register offset within the bank for each of the defined JTAG-AP registers.

**Table 12-2 Bank and Offset values for accessing the JTAG-AP registers**

JTAG-AP register	Access	Address	Register bank, APBANKSEL <sup>a</sup>	Offset, A[3:2] <sup>b</sup>
CSW	R/W	0x00	0x0	0, b00
PSEL	R/W	0x04	0x0	1, b01
PSTA	R/W	0x08	0x0	2, b10
BRFIFO1	RO	0x10	0x1	0, b00
BWFIFO1	WO			
BRFIFO2	RO	0x14	0x1	1, b01
BWFIFO2	WO			
BRFIFO3	RO	0x18	0x1	2, b10
BWFIFO3	WO			

**Table 12-2 Bank and Offset values for accessing the JTAG-AP registers (continued)**

JTAG-AP register	Access	Address	Register bank, APBANKSEL <sup>a</sup>	Offset, A[3:2] <sup>b</sup>
BRFIFO4	RO	0x1C	0x1	3, b11
BWFIFO4	WO			
IDR	RO	0xFC	0xF	3, b11

a. In the DP SELECT Register, see *The AP Select Register, SELECT* on page 6-15.

b. In the APACC or DPACC:  
for accesses to a JTAG-DP, see *The JTAG-DP DP and AP Access Registers (DPACC and APACC)* on page 4-14  
for accesses to a SW-DP, see *Serial Wire Debug protocol operation* on page 5-5.

Accesses to other combinations of Register bank and Offset Read As Zero, and are ignored on write.

## 12.2 JTAG-AP detailed register descriptions

This section describes each of the JTAG-AP registers. These register are listed in Table 12-1 on page 12-2, and that table indexes the detailed register descriptions in this section. The registers are describe in the following subsections:

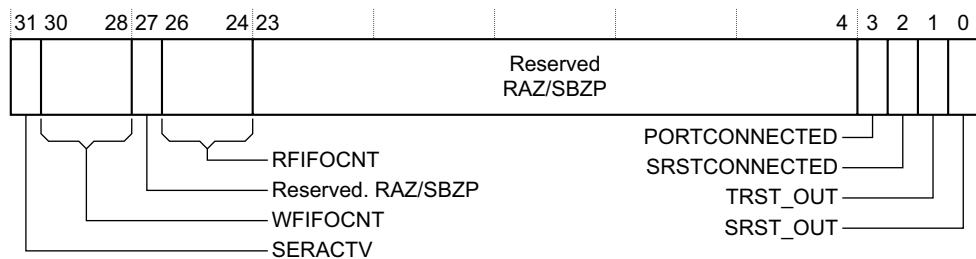
- *The Control/Status Word Register, CSW*
- *The Port Select Register, PSEL* on page 12-7
- *The Port Status Register, PSTA* on page 12-9
- *The Byte FIFO registers, BRFIFO1 to BRFIFO4 and BWFIFO1 to BWFIFO4* on page 12-10.

### 12.2.1 The Control/Status Word Register, CSW

The CSW configures and controls transfers through the JTAG interface. The CSW is:

- At offset 0x00 in the JTAG-AP register space. This means it is register 0 in bank 0.
- A read/write register, although some bits of the register are read-only.

Figure 12-1 shows the register bit assignments.



**Figure 12-1 JTAG-AP Control/Status Word Register, CSW, bit assignments**

Table 12-3 lists the bit assignments for the CSW.

**Table 12-3 Bit assignments for the JTAG-AP Control/Status Word Register, CSW**

Bits	Function	Access	Reset value	Description
[31]	SERACTV	RO	0	JTAG engine active. The possible values of this flag are: 0: JTAG engine is inactive 1: JTAG engine is processing commands from the Command FIFO.  The JTAG engine is only guaranteed to be inactive if both SERACTV and WFIFOCNT (bits [30:28]) are zero.

**Table 12-3 Bit assignments for the JTAG-AP Control/Status Word Register, CSW (continued)**

Bits	Function	Access	Reset value	Description
[30:28]	WFIFOCNT	RO	b000	Command FIFO outstanding byte count. Gives the number of command bytes held in the Command FIFO that have yet to be processed by the JTAG Engine.
[27]	-	RAZ/SZP	-	Reserved.
[26:24]	RFIFOCNT	RO	b000	Response FIFO outstanding byte count. Gives the number of bytes of response data available in the Response FIFO.
[23:4]	-	RAZ/SZP	-	Reserved.
[3]	PORTCONNECTED	RO	- <sup>a</sup>	Selected ports connected. The value of this bit is the logical AND of the <b>PORTCONNECTED</b> signals from all currently-selected ports.
[2]	SRSTCONNECTED	RO	- <sup>a</sup>	Selected ports reset connected. The value of this bit is the logical AND of the <b>SRSTCONNECTED</b> signals from all currently-selected ports.
[1]	TRST_OUT	R/W	0	This bit specifies the signal to drive out on the <b>TRST*</b> signal for the currently-selected port or ports. The <b>TRST*</b> signal is active LOW, when this bit is set to 1 the <b>TRST*</b> output is LOW. For more information about the use of this bit see <i>Resetting JTAG devices</i> on page 12-7. This bit does not self-reset, it must be cleared to 0 by a software write to this register.
[0]	SRST_OUT	R/W	0	This bit specifies the signal to drive out on the <b>nSRSTOUT</b> signal for the currently-selected port or ports. The <b>nSRSTOUT</b> signal is active LOW, when this bit is set to 1 the <b>nSRSTOUT</b> output is LOW. This bit does not self-reset, it must be cleared to 0 by a software write to this register.

a. The value of these fields always depend on the state of the connected signals when the register is read.

## Resetting JTAG devices

Although the TRST\_OUT bit, bit [1] of the CSW Register, specifies the value to be driven on the **TRST\*** signal, writing to this bit only causes the signal to change. It might be necessary to clock the devices connected to the selected JTAG port or ports, by **TCK**, to enable the devices to recognize the change on **TRST\***. This means that the normal process to perform a Test Reset of the selected JTAG ports is:

1. Write 1 to the TRST\_OUT bit, bit [1] of the CSW Register, to specify that **TRST\*** must be asserted LOW.
2. Drive a sequence of at least five **TMS** = 1 clocks from the JTAG engine. You can do this by issuing the command b00111111 to the JTAG engine. This sequence guarantees the TAP enters the Test-Logic/Reset state, even if has no **TRST\*** connection.
3. Write 0 to the TRST\_OUT bit of the CSW Register, so 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 then some or all TAPs might not reset.

### 12.2.2 The Port Select Register, PSEL

The PSEL Register selects one or more JTAG ports.

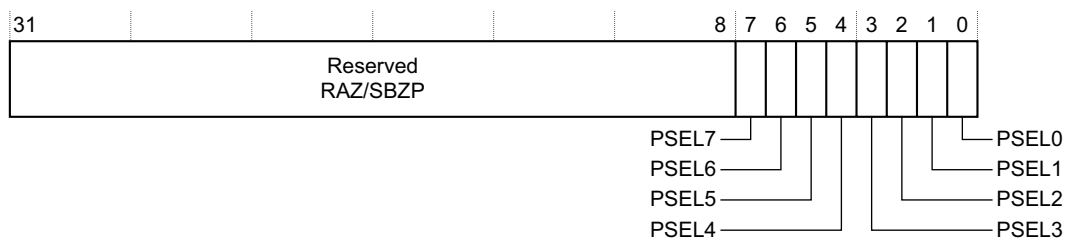
A JTAG port is enabled if all of the following apply:

- the port is connected to the JTAG-AP
- the PSEL<sub>n</sub> bit for the port, in the Port Select Register, is set to 1
- the **PORTENABLED** signal from the port to the JTAG-AP is asserted HIGH.

The PSEL Register is:

- at offset 0x04 in the JTAG-AP register space. This means it is register 1 in bank 0.
- a read/write register.

Figure 12-2 shows the register bit assignments.



**Figure 12-2 JTAG-AP Port Select Register, PSEL, bit assignments**

Table 12-4 lists the bit assignments for the PSEL Register.

**Table 12-4 Bit assignments for the JTAG-AP Port Select Register, PSEL**

Bits	Function	Access	Reset value	Description
[31:8]	-	RAZ/SBZP	-	Reserved.
[7]	PSEL7	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 7 is selected <sup>a</sup> .
[6]	PSEL6	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 6 is selected <sup>a</sup> .
[5]	PSEL5	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 5 is selected <sup>a</sup> .
[4]	PSEL4	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 4 is selected <sup>a</sup> .
[3]	PSEL3	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 3 is selected <sup>a</sup> .
[2]	PSEL2	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 2 is selected <sup>a</sup> .
[1]	PSEL1	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 1 is selected <sup>a</sup> .
[0]	PSEL0	R/W	UNPREDICTABLE	When this bit is set to 1, JTAG port 0 is selected <sup>a</sup> .

- a. You can select a port in the PSEL Register even if the port is not connected to the JTAG-AP. However a port is only enabled if it is connected to the JTAG-AP, and selected in the PSEL Register, and the **PORTENABLED** signal asserted.

### ———— Caution ————

You must only write to this register when the JTAG engine is inactive and the WFIFO is empty. Writing PSEL at any other time has UNPREDICTABLE results.

This means that before writing to PSEL you must read the JTAG-AP CSW and check that the SERACTV and WFIFOCNT fields are both zero. See *The Control/Status Word Register, CSW* on page 12-5 for more information.

When more than one JTAG port is connected to the JTAG-AP, the same values for **TDI**, **TMS**, **TRST\*** and **nSRSTOUT** are driven to all ports that are selected in the PSEL Register. If more than one port is selected in the PSEL register the return values from **TDO** is UNPREDICTABLE.

This selection model means that, with the normal serially-connected model for JTAG, it is possible to update multiple TAPs in parallel. This functionality can be very useful, for example to provide synchronized behavior.



Because each JTAG port might contain multiple TAPs connected in series, the process for updating TAPs in parallel is:

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 of the required ports.
3. Scan through the TAP Update state. All of the TAPs are updated synchronously.

**Note**

In the normal serially-connected JTAG model, Instruction Register updates are always made in parallel.

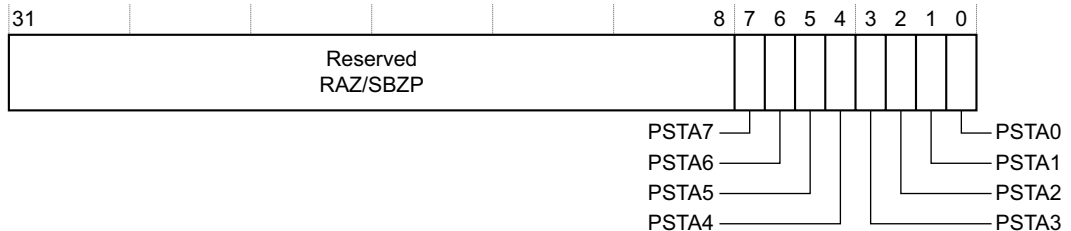
### 12.2.3 The Port Status Register, PSTA

The PSTA Register indicates whenever a JTAG port that is connected and selected has been disabled, even if the disable is only transient.

The PSTA Register is:

- at offset 0x08 in the JTAG-AP register space.
- a read/write register.

Figure 12-3 shows the register bit assignments.



**Figure 12-3 JTAG-AP Port Status Register, PSTA, bit assignments**

Table 12-5 on page 12-10 lists the bit assignments for the PSTA Register.

**Table 12-5 Bit assignments for the JTAG-AP Port Status Register, PSTA**

Bits	Function	Access	Reset value	Description
[31:8]	-	RAZ/SBZP	-	Reserved.
[7]	PSTA7	R/W <sup>a</sup>	0	Sticky status flags for JTAG ports 7 to 0. For a JTAG port that is connected to the JTAG-AP and selected in the PSEL register, the corresponding PSTAn bit is set to 1 if the port is disabled. These flags are sticky, meaning the PSTAn bit remains set to 1 even if the port is re-enabled. These flags capture and hold the fact that a connected and selected port has been disabled or powered down, even if this is only transient. When set to 1, a flag can only be cleared to 0 by writing 1 to the appropriate bit in this register.
[6]	PSTA6	R/W <sup>a</sup>	0	
[5]	PSTA5	R/W <sup>a</sup>	0	
[4]	PSTA4	R/W <sup>a</sup>	0	
[3]	PSTA3	R/W <sup>a</sup>	0	
[2]	PSTA2	R/W <sup>a</sup>	0	
[1]	PSTA1	R/W <sup>a</sup>	0	
[0]	PSTA0	R/W <sup>a</sup>	0	

a. These are sticky bits. When a bit is set to 1, it can only be cleared by writing 1 to the bit. Writing 0 to a bit has no effect.

If a port is not connected to the JTAG-AP, its PSTAn bit is Read As Zero.

## 12.2.4 The Byte FIFO registers, BRFFIFO1 to BRFFIFO4 and BWFIFO1 to BWFIFO4

The JTAG-AP engine JTAG protocol is byte encoded. The Byte FIFO registers:

- enable one, two, three or four bytes to be written in parallel to the Command FIFO, by writing to a BWFIFOn register
- enable one, two, three or four bytes to be read in parallel from the Response FIFO, by reading from a BRFFIFOn register.

The BRFFIFOn and BWFIFOn registers are mapped to the same JTAG-AP register addresses, with the BRFFIFOn registers being accessed on read operations and the BWFIFOn registers being accessed on write operations.

The registers are organized so that the low order bits of the register address indicate the number of bytes of data to be written to, or read from, the FIFO:

- the BRFFIFO1 and BWFIFO1 registers are at offset 0x10 in the JTAG-AP register space, and are used to transfer a single data byte from or to a FIFO
- the BRFFIFO2 and BWFIFO2 registers are at offset 0x14 in the JTAG-AP register space, and are used to transfer two data bytes from or to a FIFO

- the BRFIFO3 and BWFIFO3 registers are at offset 0x18 in the JTAG-AP register space, and are used to transfer three data bytes from or to a FIFO
- the BRFIFO4 and BWFIFO4 registers are at offset 0x1C in the JTAG-AP register space, and are used to transfer four data bytes from or to a FIFO.

The JTAG Engine Byte Command protocol used for the commands and responses is described in *The JTAG Engine Byte Command Protocol* on page 9-10.

### The Byte Read FIFO Registers, BRFIFO1 to BRFIFO4

Figure 12-4 shows the register bit assignments for the Byte Read FIFO Registers.

	31	24	23	16	15	8	7	0
BRFIFO1	RAZ		RAZ		RAZ		First response byte	
BRFIFO2	RAZ		RAZ		Second response byte		First response byte	
BRFIFO3	RAZ		Third response byte		Second response byte		First response byte	
BRFIFO4	Fourth response byte		Third response byte		Second response byte		First response byte	

**Figure 12-4 Bit assignments for the Byte Read FIFO Registers, BRFIFO1 to BRFIFO4**

An AP transaction that reads more responses than are available in the Response FIFO stalls until enough data is available to match the request. Before initiating an AP transaction to read from the Response FIFO you can read the RFIFOCNT field in the JTAG-AP CSW to check the number of response bytes available, see *The Control/Status Word Register, CSW* on page 12-5.

### The Byte Write FIFO Registers, BWFIFO1 to BWFIFO4

Figure 12-5 shows the register bit assignments for the Byte Write FIFO Registers.

	31	24	23	16	15	8	7	0
BWFIFO1	Ignored		Ignored		Ignored		First command byte	
BWFIFO2	Ignored		Ignored		Second command byte		First command byte	
BWFIFO3	Ignored		Third command byte		Second command byte		First command byte	
BWFIFO4	Fourth command byte		Third command byte		Second command byte		First command byte	

**Figure 12-5 Bit assignments for the Byte Write FIFO Registers, BWFIFO1 to BWFIFO4**

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. Before initiating an AP transaction to write to the Command FIFO you can read the WFIFOCNT field in the JTAG-AP CSW to check the number of commands already present in the Command FIFO, and from this value you know the number of additional commands you can write to the FIFO. For more information see *The Control/Status Word Register, CSW* on page 12-5.



# Chapter 13

## Component and Peripheral ID Registers

This chapter describes the Component and Peripheral ID Registers that form part of the register space of every debug component that complies with the ARM Peripheral Identification specification. This means that these registers form part of the Debug Register Files and ROM Tables shown in Figure 1-5 on page 1-9, and in other illustrations of debug systems.

This chapter contains the following sections:

- *Component and Peripheral ID registers* on page 13-2
- *The Component ID Registers* on page 13-4
- *The Peripheral ID Registers* on page 13-9.

---

**Note**

Contact ARM Limited if you require more details of the ARM Peripheral Identification specification.

---

## 13.1 Component and Peripheral ID registers

The Component and Peripheral ID registers provide a generic model for component identification.

A generic component occupies a continuous register space that covers one or more 4KB blocks. The Peripheral and Component ID registers are always located at the end of this register space, with the Component ID Registers occupying the last four words of this block. This means that, if a component occupies more than one 4KB block, the Component and Peripheral ID Registers are located at the end of the last block.

This section gives a summary of the registers, including the memory map of the registers in the final 4KB block of register address space. The registers are described in detail in:

- *The Component ID Registers* on page 13-4
- *The Peripheral ID Registers* on page 13-9.

### 13.1.1 Summary of Component and Peripheral ID Registers

Table 13-1 lists the Component and Peripheral ID Registers, and indicates where each register is described in detail. This table shows the memory map of these registers, at the end of the final 4KB block of register space.

All of the registers listed in Table 13-1 are required in every debug component implementation.

**Table 13-1 Summary of Component and Peripheral ID Registers**

Address	Name	Access	Value	For description see
0xFD0	Peripheral ID4	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Peripheral ID4 Register</i> on page 13-15
0xFD4	Peripheral ID5	RO	0x00000000	<i>Peripheral ID5 to Peripheral ID7 Registers</i> on page 13-16
0xFD8	Peripheral ID6	RO	0x00000000	
0xFDC	Peripheral ID7	RO	0x00000000	
0xFE0	Peripheral ID0	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Component ID0 Register</i> on page 13-6
0xFE4	Peripheral ID1	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Peripheral ID1 Register</i> on page 13-12
0xFE8	Peripheral ID2	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Peripheral ID2 Register</i> on page 13-13
0xFEC	Peripheral ID3	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Peripheral ID3 Register</i> on page 13-14

**Table 13-1 Summary of Component and Peripheral ID Registers (continued)**

Address	Name	Access	Value	For description see
0xFF0	Component ID0	RO	0x00000000	<i>Component ID0 Register</i> on page 13-6
0xFF4	Component ID1	RO	IMPLEMENTATION DEFINED <sup>a</sup>	<i>Component ID1 Register</i> on page 13-6
0xFF8	Component ID2	RO	0x00000005	<i>Component ID2 Register</i> on page 13-7
0xFFC	Component ID3	RO	0x000000B1	<i>Component ID3 Register</i> on page 13-8

a. See the register description for additional information.

## 13.2 The Component ID Registers

There are four read-only Component Identification Registers, Component ID3 to Component ID0. Table 13-2 lists these registers:

**Table 13-2 Summary of the Component Identification Registers**

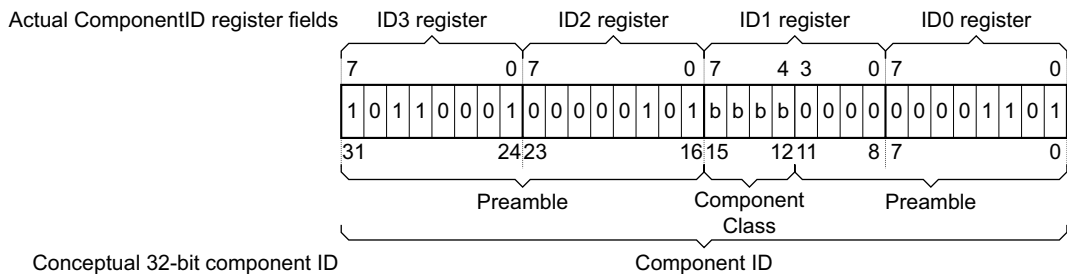
Register	Address
Component ID0	0xFF0
Component ID1	0xFF4
Component ID2	0xFF8
Component ID3	0xFFC

These registers occupy the last four words of a 4KB register space of a component. If the register space of a component occupies more than one 4KB block they are the last four words of the last 4KB block.

The Component ID Registers:

- Identify the 4KB block of memory space as a component, by the use of a signature.
- Identify the component type, by the contents of the signature. This means that the contents of the Component ID Registers act as a preamble to a component type-specific set of identification registers.

Only bits [7:0] of each register are used. Figure 13-1 shows the concept of a single 32-bit component ID, obtained from the four Component Identification Registers.



**Figure 13-1 Mapping between the Component ID Registers and the Component ID value**



The Component Class field identifies the type of the component. Table 13-3 lists the permitted values for this field. It also shows which components have a standardized layout for the remainder of the 4KB register space.

**Table 13-3 Component Class values in the Component ID**

Component Class value	Class description	Standardized layout
0x0	Generic verification component	None.
0x1	ROM Table	See Chapter 14 <i>ROM Tables</i> .
0x2 - 0x8	Reserved	-
0x9	Debug component	See the <i>CoreSight Architecture Specification</i> .
0xA	Reserved	-
0xB	Peripheral Test Block (PTB)	None.
0xC	Reserved	-
0xD	OptimoDE Data Engine SubSystem (DESS) component	None.
0xE	Generic IP component	None.
0xF	PrimeCell peripheral	None.

As shown in Table 13-3, ROM Table components have a standard register space layout. This is defined in Chapter 14 *ROM Tables*.

Debug components have a standard layout of Peripheral Identification Registers, as defined in the *CoreSight Architecture Specification*. These registers are summarized in *The Peripheral ID Registers* on page 13-9.

———— **Note** ————

The CoreSight Architecture Specification requires CoreSight components to implement other registers in the address space 0xF00 to 0xFFF, in addition to the Component ID Registers and Peripheral ID Registers.

Processors that comply with the ARMv7 Debug Architecture, and trace macrocells that comply with the ETM Architecture Specification version 3.2 or later, identify themselves as Debug components. For more information see the *ARM Architecture Reference Manual Debug Supplement* and the *ETM Architecture Specification*.

The following sections describe each of the Component ID Registers.

13.2.1 Component ID0 Register

The Component ID0 Register holds byte 0 of the preamble information. It is a read-only register at address offset 0xFF0.

Figure 13-2 shows the Component ID0 Register bit assignments:

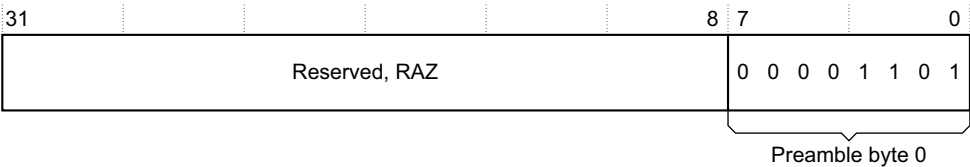


Figure 13-2 Component ID0 Register bit assignments

Table 13-4 lists the bit assignments for the Component ID0 Register:

Table 13-4 Component ID0 Register bit assignments

Bits	Value	Description
[31:8]	-	Reserved. RAZ.
[7:0]	0x0D	Preamble byte 0

13.2.2 Component ID1 Register

The Component ID1 Register holds byte 1 of the preamble information. This includes the Component Class field. It is a read-only register at address offset 0xFF4.

Figure 13-3 shows the Component ID1 Register bit assignments:

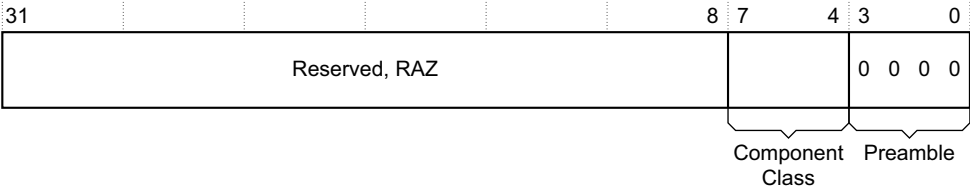


Figure 13-3 Component ID1 Register bit assignments

Table 13-5 lists the bit assignments for the Component ID1 Register:

Table 13-5 Component ID1 Register bit assignments		
Bits	Value	Description
[31:8]	-	Reserved. RAZ.
[7:4]	See Table 13-3 on page 13-5	Component Class
[3:0]	0x0	Preamble.

The possible values of the Component Class field are shown in Table 13-3 on page 13-5.

13.2.3 Component ID2 Register

The Component ID2 Register holds byte 2 of the preamble information. It is a read-only register at address offset 0xFF8.

Figure 13-4 shows the Component ID2 Register bit assignments:

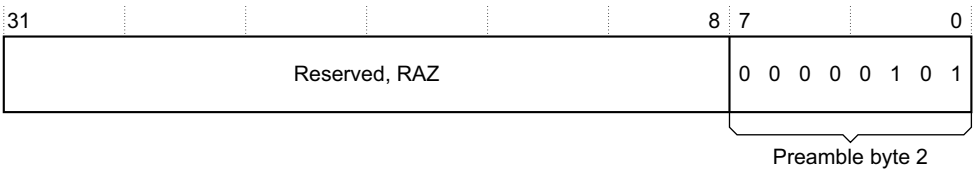


Figure 13-4 Component ID2 Register bit assignments

Table 13-6 lists the bit assignments for the Component ID2 Register:

Table 13-6 Component ID2 Register bit assignments		
Bits	Value	Description
[31:8]	-	Reserved. RAZ.
[7:0]	0x05	Preamble byte 2.

13.2.4 Component ID3 Register

The Component ID3 Register holds byte 3 of the preamble information. It is a read-only register at address offset 0xFFC.

Figure 13-5 shows the Component ID3 Register bit assignments:

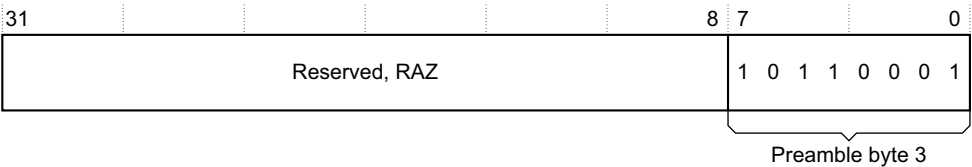


Figure 13-5 Component ID3 Register bit assignments

Table 13-7 lists the bit assignments for the Component ID3 Register:

Table 13-7 Component ID3 Register bit assignments

Bits	Value	Description
[31:8]	-	Reserved. RAZ.
[7:0]	0xB1	Preamble byte 3.

### 13.3 The Peripheral ID Registers

The peripheral identification registers provide standard information required by all components that conform to the generic ID registers specification. They identify a peripheral within a particular namespace.

A set of Peripheral ID Registers comprises eight registers. Table 13-8 lists the Peripheral ID registers in order of their address offsets, and gives their offsets within a 4KB register space.

Table 13-8 Summary of the peripheral identification registers

Description	Address
Peripheral ID4	0xFD0
Peripheral ID5	0xFD4
Peripheral ID6	0xFD8
Peripheral ID7	0xFDC
Peripheral ID0	0xFE0
Peripheral ID1	0xFE4
Peripheral ID2	0xFE8
Peripheral ID3	0xFEC

**Note**

Table 13-8 lists the Peripheral ID registers in address order. You can see that this does not match the numerical order of the registers, ID0 to ID7.

Only bits [7:0] of each Peripheral ID Register are used, with bits [31:8] Reserved. Together, the eight Peripheral ID Registers can be considered to define a single 64-bit Peripheral ID, as shown in Figure 13-6.

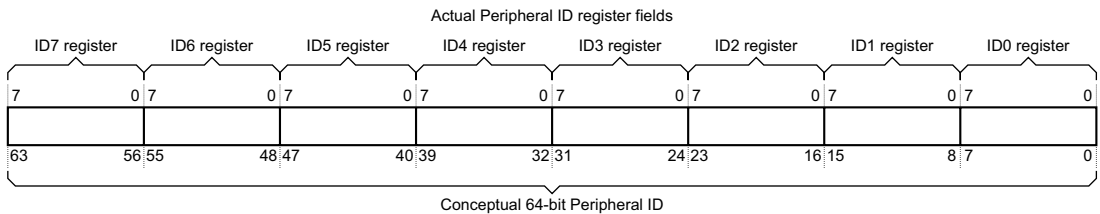
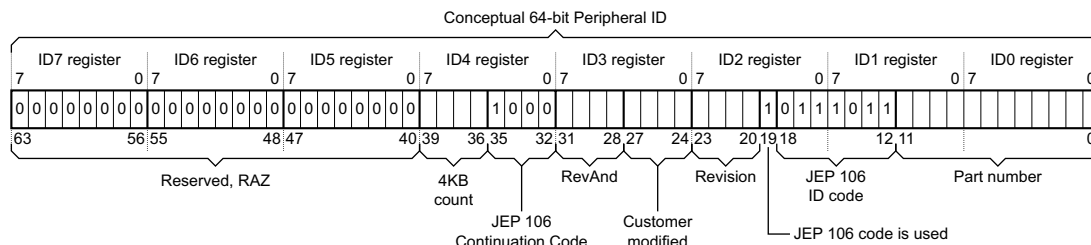


Figure 13-6 Mapping between the Peripheral ID Registers and the Peripheral ID value

Figure 13-7 on page 13-10 shows the standard Peripheral ID fields in the single conceptual Peripheral ID, for a peripheral designed by ARM Limited.



Bits with no value shown are IMPLEMENTATION DEFINED.

Other bits not shown as Reserved are for an implementation designed by ARM Limited.

**Figure 13-7 Peripheral ID fields**

Table 13-9 lists the standard Peripheral ID fields, and shows where this information is held in the Peripheral ID Registers.

**Table 13-9 Register fields for the peripheral identification registers**

Name	Size	Description	See Register
4KB Count	4 bits	Log <sub>2</sub> of the number of 4KB blocks occupied by the device. The number of blocks occupied by a device must be a power of two, and this field gives that power. The first four possible values of this field are: b0000: 1 block b0001: 2 blocks b0010: 4 blocks b0011: 8 blocks.	Peripheral ID4
JEP 106 code	4+7 bits	Identifies the designer of the device. This field consists of a 4-bit continuation code and a 7-bit identity code. For an implementation designed by ARM Limited, the continuation code is 0x4 and the identity code is 0x3B.	Peripheral ID1, Peripheral ID2, Peripheral ID4
RevAnd	4 bits	Manufacturer Revision Number. Indicates a late modification to the device, usually as a result of an Engineering Change Order. This field starts at 0x0 and is incremented by the integrated circuit manufacturer on metal fixes.	Peripheral ID3
Customer modified	4 bits	Indicates an endorsed modification to the device. If the system designer cannot modify the RTL supplied by the processor designer then this field is Read As Zero.	Peripheral ID3
Revision	4 bits	Revision of the peripheral. Starts at 0x0 and increments by 1 at both major and minor revisions.	Peripheral ID2

**Table 13-9 Register fields for the peripheral identification registers (continued)**

Name	Size	Description	See Register
JEP-106 ID Code is used	1 bit	This bit is set to 1 when a JEP 106 Identity Code is used. This bit must be 1 on new implementations.	Peripheral ID2
Part Number	12 bits	Part number for the device. Each organization designing components to the ARM Limited Peripheral Identification specifications has its own part number list.	Peripheral ID0, Peripheral ID1

For more information about these fields, see the *CoreSight Architecture Specification*.

———— **Note** ————

In legacy components, an ASCII Identity Code, allocated by ARM Limited, is used rather than the JEP 106 Identity Code. In such a component:

- The 7-bit ASCII Identity Code replaced the 7-bit JEP 106 Identity Code shown in Figure 13-7 on page 13-10. This field is split between the Peripheral ID1 and Peripheral ID2 Registers.  
On legacy components designed by ARM Limited this field has the value 0x41.
- The JEP 106 Identity Code is used flag, bit [3] of the Peripheral ID2 Register, is zero.
- The JEP 106 Continuation Code bits, bits [3:0] of the Peripheral ID4 Register, are not used and Read As Zero.

*Legacy Peripheral ID layout* on page 13-17 is an example of the use of the ASCII Identity Code.

ASCII Identity Codes *must not* be used for new designs.

The fields present in each Peripheral ID Register are indicated in the following sections, where the registers are described in register name order (ID0 to ID7). Table 13-8 on page 13-9 lists the register numbers and addresses of these registers, in register address order, which does not match the register name order.

13.3.1 Peripheral ID0 Register

The Peripheral ID0 Register holds peripheral identification information. It is a read-only register at address offset 0xFE0.

Figure 13-8 shows the Peripheral ID0 Register bit assignments:

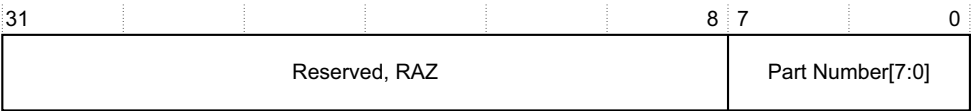


Figure 13-8 Peripheral ID0 Register bit assignments

Table 13-10 lists the bit assignments for the Peripheral ID0 Register:

Table 13-10 Peripheral ID0 Register bit assignments

Bits	Value	Description <sup>a</sup>
[31:8]	-	Reserved. RAZ.
[7:0]	IMPLEMENTATION DEFINED	Part Number[7:0]

a. See Table 13-9 on page 13-10 for more information about the register fields.

13.3.2 Peripheral ID1 Register

The Peripheral ID1 Register holds peripheral identification information. It is a read-only register at address offset 0xFE4.

Figure 13-9 shows the Peripheral ID1 Register bit assignments:

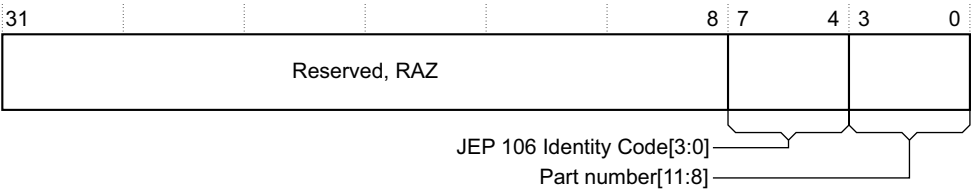


Figure 13-9 Peripheral ID1 Register bit assignments

Table on page 13 shows the bit assignments for the Peripheral ID1 Register:



Table 13-11 Peripheral ID1 Register bit assignments

Bits	Value	Description <sup>a</sup>
[31:8]	-	Reserved. RAZ.
[7:4]	IMPLEMENTATION DEFINED	JEP-106 Identity Code[3:0] <sup>b</sup>
[3:0]	IMPLEMENTATION DEFINED	Part Number[11:8]

- a. See Table 13-9 on page 13-10 for more information about the register fields.
- b. On legacy components, this might be the ASCII Identity Code[3:0]. See the footnotes to Table 13-12 on page 13-14 for more information.

13.3.3 Peripheral ID2 Register

The Peripheral ID2 Register holds peripheral identification information. It is a read-only register at address offset 0xFE8.

Figure 13-10 shows the Peripheral ID2 Register bit assignments:

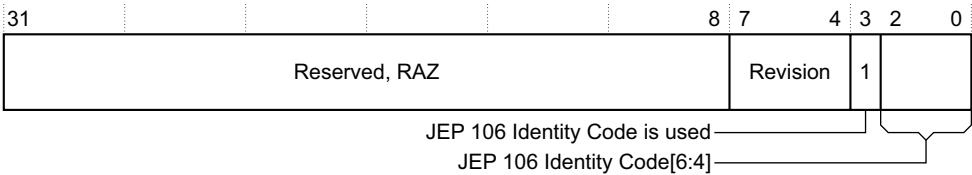


Figure 13-10 Peripheral ID2 Register bit assignments

Table 13-12 on page 13-14 shows the bit assignments for the Peripheral ID2 Register:

**Table 13-12 Peripheral ID2 Register bit assignments**

Bits	Value	Description <sup>a</sup>
[31:8]	-	Reserved. RAZ.
[7:4]	IMPLEMENTATION DEFINED	Revision.
[3]	1	JEP 106 Identity Code is used. Always 1 in new implementations. <sup>b</sup>
[2:0]	IMPLEMENTATION DEFINED	JEP 106 Identity Code[6:4]. <sup>b</sup>

a. See Table 13-9 on page 13-10 for more information about the register fields.

b. On legacy components, bit [3] might be 0. This indicates that an ASCII Identity Code is used, and that bits [2:0] are ASCII Identity Code[6:4].

### 13.3.4 Peripheral ID3 Register

The Peripheral ID3 Register holds peripheral identification information. It is a read-only register at address offset 0xFEC.

Figure 13-11 shows the Peripheral ID3 Register bit assignments:

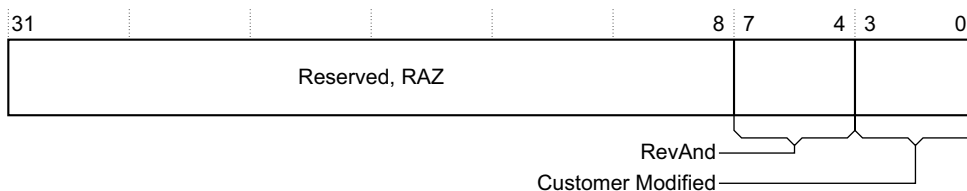
**Figure 13-11 Peripheral ID3 Register bit assignments**

Table 13-13 shows the bit assignments for the Peripheral ID3 Register:

**Table 13-13 Peripheral ID3 Register bit assignments**

Bits	Value	Description <sup>a</sup>
[31:8]	-	Reserved. RAZ.
[7:4]	IMPLEMENTATION DEFINED	RevAnd
[3:0]	IMPLEMENTATION DEFINED	Customer Modified

a. See Table 13-9 on page 13-10 for more information about the register fields.

13.3.5 Peripheral ID4 Register

The Peripheral ID4 Register holds peripheral identification information. It is a read-only register at address offset 0xFD0.

Figure 13-12 shows the Peripheral ID4 Register bit assignments:

**Note**

On legacy components, when bit [3] of the Peripheral ID2 Register is 0 the Peripheral ID4 Register is not implemented. However, the register at 0xFD0 might be used by the component.

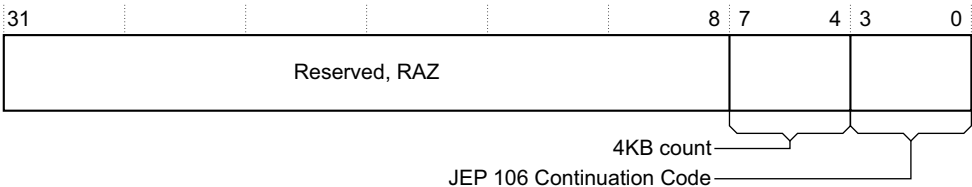


Figure 13-12 Peripheral ID4 Register bit assignments

Table 13-14 shows the bit assignments for the Peripheral ID4 Register:

Table 13-14 Peripheral ID4 Register bit assignments

Bits	Value	Description <sup>a</sup>
[31:8]	-	Reserved. RAZ.
[7:4]	IMPLEMENTATION DEFINED	4KB count
[3:0]	IMPLEMENTATION DEFINED	JEP-106 Continuation Code

a. See Table 13-9 on page 13-10 for more information about the register fields.

### 13.3.6 Peripheral ID5 to Peripheral ID7 Registers

No information is held in the Peripheral ID5, Peripheral ID6 and Peripheral ID7 Registers. Table 13-8 on page 13-9 shows the address offsets of these registers.

---

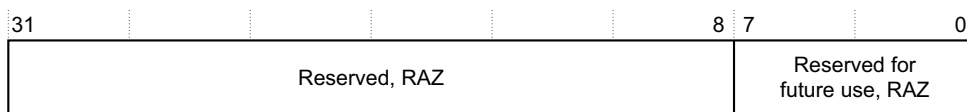
#### Note

---

On legacy components, where bit [3] of the Peripheral ID2 Register is 0, the Peripheral ID5 to Peripheral ID7 Registers are not implemented. However, the registers at 0xFD4, 0xFD8 and 0xFDC might be used by the component.

---

Figure 13-13 shows the bit assignments for these registers:



**Figure 13-13 Peripheral ID5 to Peripheral ID7 Registers, bit assignments**

Table 13-15 shows the bit assignments for the Peripheral ID5, Peripheral ID6 and Peripheral ID7 Registers:

**Table 13-15 Peripheral ID5 to Peripheral ID7 Registers, bit assignments**

Bits	Value	Description
[31:8]	-	Reserved. RAZ.
[7:0]	-	Reserved for future use. RAZ.

### 13.3.7 Legacy Peripheral ID layout

Table 13-16 shows the format of the peripheral identification registers in a legacy component

**Table 13-16 Identification Registers for a legacy component**

Address	Access	Register	Bits	Value	Description
0xFD0 - 0xFDC	-	-	[31:0]	-	Reserved. Might be used by component. If so, value is IMPLEMENTATION DEFINED.
0xFE0	RO	Peripheral ID0	[31:8]	-	Reserved. RAZ.
			[7:0]	IMPLEMENTATION DEFINED	Part number[7:0]
0xFE4	RO	Peripheral ID1	[31:8]	-	Reserved. RAZ.
			[7:4]	IMPLEMENTATION DEFINED	ASCII Identity code[3:0]
			[3:0]	IMPLEMENTATION DEFINED	Part number[11:8]
0xFE8	RO	Peripheral ID2	[31:8]	-	Reserved. RAZ.
			[7:4]	IMPLEMENTATION DEFINED	Revision number of peripheral.
			[3]	0	ASCII Identity code is used.
			[2:0]	IMPLEMENTATION DEFINED	ASCII Identity code[6:4].
0xFE8	RO	Peripheral ID3	[31:8]	-	Reserved. RAZ.
			[7:0]	IMPLEMENTATION DEFINED	Configuration Register.

A legacy peripheral component does not use JEP 106 Identity Codes, and only implements four Peripheral ID Registers.

The Configuration Register, that corresponds to the Peripheral ID3 Register, contains information specific to the peripheral about build options. For example, it might indicate the width of a bus in the implementation.



# Chapter 14

## ROM Tables

The chapter describes ROM Tables. It includes the following sections:

- *ROM Table overview* on page 14-2
- *ROM Table entries* on page 14-5
- *The MEMTYPE Register* on page 14-9
- *Component and Peripheral ID Registers* on page 14-10
- *ROM Table hierarchies* on page 14-12.

## 14.1 ROM Table overview

ROM Tables are used to hold information about debug components.

- When a Debug Access Port connects to a single debug component, a ROM Table is not required. However, a designer might choose to implement such a system to include a ROM Table, as shown in Figure 2-1 on page 2-7.
- If a Debug Access Port connects to more than one debug component then the system must include at least one ROM Table.

A ROM Table connects to a bus controlled by a Memory Access Port (MEM-AP). In other words, the ROM Table is part of the address space of the memory system that is connected to a MEM-AP. There can be more than one ROM Table connected to a single bus.

A ROM Table:

- Always occupies 4KB of memory.
- Is a read-only device. Writes to ROM Table addresses are ignored.

Two ROM Table formats are supported, one for 8-bit ROMs and one for 32-bit ROMs. In the 8-bit format, entries are stored LSB-first.

A ROM Table is divided into a number of regions, as shown in Table 14-1. The format of the ROM Table, for both 8-bit and 32-bit ROMs, is illustrated in Figure 14-1 on page 14-3.

**Table 14-1 ROM Table regions**

Region	Start address <sup>a</sup>	End address <sup>a</sup>	Notes
ROM Table entries	0x000	IMPLEMENTATION DEFINED	See <i>ROM Table entries</i> on page 14-5.
Reserved region	IMPLEMENTATION DEFINED	0xFCB	Unused area of ROM Table. See <i>The unused area of the ROM Table</i> on page 14-8
MEMTYPE register	0xFCC	0xFCF	See <i>The MEMTYPE Register</i> on page 14-9.
Peripheral ID registers	0xFD0	0xFEf	See <i>Component and Peripheral ID Registers</i> on page 14-10.
Component ID registers	0xFF0	0xFFF	

- a. For a given number of entries, the size of the ROM Table entries region of the ROM Table is larger for an 8-bit ROM Table implementation than for a 32-bit ROM Table implementation. As a result, the Reserved region that ends at 0xFCB is smaller in the 8-bit implementation than it is in the 32-bit implementation. This is shown in Figure 14-1 on page 14-3. The other regions are always identical in the 8-bit and 32-bit implementations.



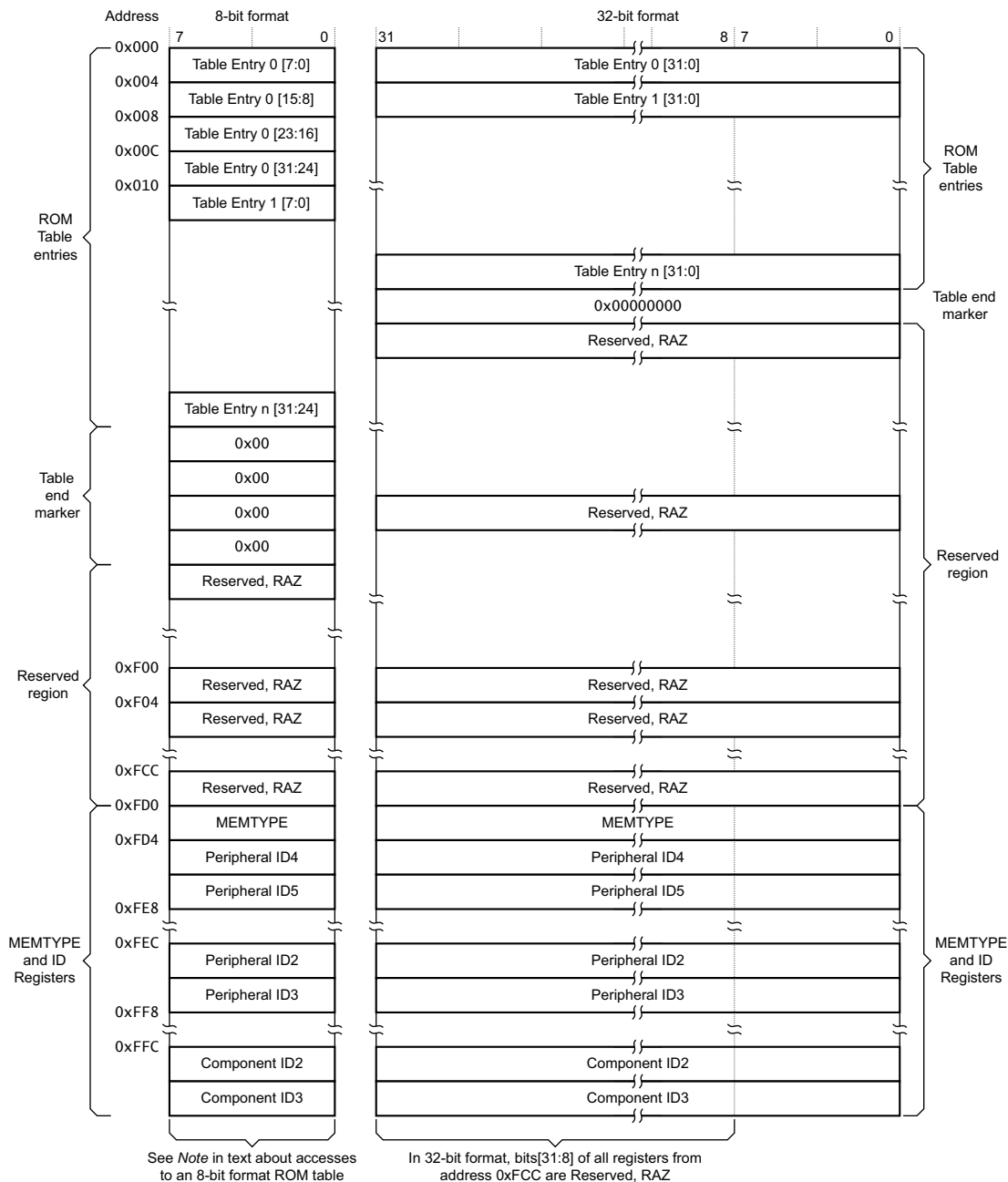


Figure 14-1 ROM Table formats, for 8-bit and 32-bit ROM

---

**Note**

---

- For a given number of entries, the *ROM Table entries* region in an 8-bit format ROM Table is four times as long as it would be in a 32-bit format ROM Table.
  - The ROM Table area from 0xF00 to 0xFCB is Reserved, RAZ. The last possible ROM Table entry is at 0xEFC.
  - AP accesses are 32 bits wide. When reading a location in an 8-bit format ROM Table, bits [31:8] of the access Read As Zero.
-

## 14.2 ROM Table entries

The series of ROM Table entries start at the base address of the ROM Table. The values of these entries depend on the subsystem that is implemented.

Each ROM Table entry:

- describes a single debug component within the system
- consists of four bytes of data:
  - in a 32-bit ROM Table an entry is a single 32-bit word
  - in an 8-bit ROM Table an entry is four 8-bit words, stored LSB-first.

Figure 14-1 on page 14-3 shows how an entry is formatted, in 8-bit and 32-bit ROM Tables.

Table 14-2 shows the format of a single 32-bit entry in the ROM Table.

**Table 14-2 Format of a ROM Table entry**

Bits	Field name	Description
[31:12]	Address offset	<p>The base address of the component, relative to the base address of this ROM Table. Negative values are permitted, using two's complement.</p> <hr/> <p><b>Note</b></p> <p>The Address offset field of a ROM Table entry must not be zero, even if bit [0] of the entry is 0. This is because a zero address offset points back to this ROM Table.</p> <hr/> <p>See <i>Component descriptions and the component base address</i> on page 14-6 for more information.</p>
[11:2]	-	Reserved. Should Be Zero.
[1]	Format	<p>This bit indicates the format of the ROM Table. Its possible values are:</p> <p>0: 8-bit ROM Table.</p> <p>1: 32-bit ROM Table</p> <p>The value of the Format bit must be the same for all entries in the ROM Table.</p>
[0]	Entry present	<p>This bit indicates whether an entry is present at this location in the ROM table. Its possible values are:</p> <p>0: Entry not present</p> <p>1: Entry present.</p> <p>See <i>Empty entries and the end of the ROM Table</i> on page 14-7 for more information.</p>

### 14.2.1 Component descriptions and the component base address

Each debug component occupies one or more 4KB blocks of address space. This block of address space is referred to as the *Debug Register File* for the component. See Figure 1-5 on page 1-9, and other figures in Chapter 1 and Chapter 2, for examples.

The Address offset field of a ROM Table entry points to the start of the *last* 4KB block of the address space of the component. This block always contains the Component and Peripheral ID Registers for the component, starting at offset 0xFD0 from the start of the block. The 4KB count field, bits [7:4], of the Peripheral ID4 Register specifies the number of 4KB blocks for the component. Therefore, the process for finding the start of the address space for a component is:

1. Read the ROM Table entry for the component, and extract the Address offset for the component. The Address offset is bits [31:12] of the ROM Table entry.
2. Use the Address offset, together with the base address of the ROM Table, to calculate the base address of the component. The component base address is:

$$\text{Component\_Base\_Address} = \text{ROM\_Base\_Address} + (\text{Address\_Offset} \ll 12)$$

When performing this calculation, remember that the Address\_Offset value might be a two's complement negative value.

Component\_Base\_Address is the start address of the final 4KB block of the address space for the component.

3. Read the Peripheral ID4 Register for the component. The address of this register is:  

$$\text{Peripheral\_ID4\_address} = \text{Component\_Base\_Address} + 0xFD0$$
4. Extract the 4KB count field, bits [7:4], from the value of the Peripheral ID4 Register.
5. Use the 4KB count value to calculate the start address of the address space for the component.  
 If the 4KB count field is b0000, indicating a count value of 1, the address space for the component starts at the Component\_Base\_Address obtained at stage 2.

In general, the ROM Table indicates all the valid addresses in the memory map of the connection from the ADI to the system being debugged. For more information about accesses to addresses that are not pointed to by the ROM Table see *The MEMTYPE Register* on page 14-9

#### ————— **Note** —————

As explained in this section, the ROM Table only indicates the base address of each component, and you must examine the Peripheral ID Registers for the component to check whether the component occupies more than one 4KB memory block.

For more information about the Component and Peripheral ID Registers see Chapter 13 *Component and Peripheral ID Registers*.

## 14.2.2 Empty entries and the end of the ROM Table

The descriptions of the debug components are stored in sequential locations in the ROM Table, starting at the ROM Table base address. However, a ROM Table entry can be marked as *not present* by setting bit [0] of the entry to 0.

When scanning the ROM Table, an entry marked as not present must be skipped. However you must not assume that an entry that is marked as not present represents the end of the ROM Table. For example, a ROM Table might be generated using static configuration tie-offs that indicate the presence or absence of particular devices, giving *not present* entries in the ROM Table.

### ———— **Note** ————

If the top-level ROM Table is generated using static configuration tie-offs, then the Peripheral ID Register values must also depend on these tie-offs. This is because each possible topology must have a unique Peripheral ID.

*ROM Table hierarchies* on page 14-12 describes how there can be a hierarchy of ROM Tables, and that such a hierarchy must have a top-level ROM Table.

## The end of the ROM Table

Immediately after the last component entry in the ROM Table there must be a blank ROM Table entry. This means a 32-bit entry with the value 0x00000000:

- in a 32-bit ROM table, the blank entry is a 32-bit word with the value 0x00000000
- in an 8-bit ROM table, the blank entry comprises four consecutive 8-bit words:
  - starting on a 4-word boundary, that is at address 0xnn0
  - each 8-bit word has the value 0x00.

### ———— **Note** ————

Because it is possible for a ROM Table to include entries that are marked as not present, a single byte of 0x00 must not be taken to indicate the end of the ROM Table.

This blank entry marks the end of the ROM Table.

## The unused area of the ROM Table

A ROM Table always:

- occupies a single 4KB block of memory
- has its ROM Table entries starting at offset 0x000 in the ROM Table
- has a blank entry to indicate the end of the ROM Table entries
- has a Reserved area, starting at offset 0xF00 in the ROM Table
- has its MEMTYPE and ID Registers starting at offset 0xFCC in the ROM Table.

This means that there is almost always an unused area in the ROM Table, between the blank entry that marks the end of the ROM Table entries and the start of the Reserved area at offset 0xF00. This unused area is Reserved, and must Read As Zero. Similarly, the Reserved area starting at offset 0xF00 and ending at 0xFCB, immediately before the MEMTYPE Register, must Read As Zero.

---

### Note

- An 8-bit ROM Table can hold up to 240 entries.
- A 32-bit ROM Table can hold up to 960 entries.

If a ROM Table holds its maximum number of entries, there is no blank entry indicating the end of the ROM Table. The table entries end at offset 0xF00 in the ROM Table.

Unless a ROM Table is holding its maximum number of entries there is an unused area between the blank entry that marks the end of the ROM Table entries and the start of the Reserved area at offset 0xF00.

An implementation is unlikely to require the maximum number of entries in a ROM Table. However, *ROM Table hierarchies* on page 14-12 describes how larger ROM Tables can be constructed.

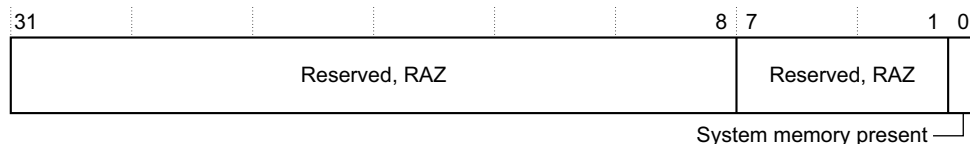
---

## 14.3 The MEMTYPE Register

Every ROM Table must implement a MEMTYPE register.

The MEMTYPE register identifies the type of memory present on the bus that connects the DAP to the ROM Table. In particular, it identifies whether system memory is connected to the bus. The MEMTYPE register is a read-only register, at offset 0xFCC from the base address of the ROM Table.

Figure 14-2 shows the register bit assignments:



**Figure 14-2 MEMTYPE Register bit assignments**

Table 14-3 lists the bit assignments for the MEMTYPE Register:

**Table 14-3 MEMTYPE Register bit assignments**

Bits	Name	Description
[31:1]	-	Reserved. RAZ.
[0]	System memory present	This bit indicates whether system memory is present on the bus that connects to the ROM Table. Its possible values are: 0: System memory not present on bus. This is a dedicated debug bus. 1: System memory is also present on this bus.

### Note

Because bits [31:8] of this register are always Reserved, Read As Zero, the register can be implemented as a single word on both 8-bit and 32-bit ROM Table implementations.

The value of bit [0] of the MEMTYPE register tells you about the addresses that the ADI can access:

- When MEMTYPE[0] = 0 the ROM Table indicates all the valid addresses in the memory system that the ADI is connected to, and the result of accessing any other address is UNPREDICTABLE, see *Component descriptions and the component base address* on page 14-6 for more information.
- When MEMTYPE[0] = 1 there might be other valid addresses in the memory system that the ADI is connected to. The result of accessing these addresses is IMPLEMENTATION DEFINED:
  - the ADI specification does not include any mechanism that the ADI can use to discover what addresses it can access, other than those indicated by the ROM Table
  - if the ADI accesses addresses other than those indicated by the ROM Table this might have side effects on the system that the ADI is connected to.

## 14.4 Component and Peripheral ID Registers

Any ROM Table must implement a set of Component and Peripheral ID Registers, that start at offset 0xFD0 in the ROM Table. These registers are described in full in Chapter 13 *Component and Peripheral ID Registers*. This section only describes particular features of the registers when they relate to a ROM Table.

---

### Note

Because bits [31:8] of these registers are always Reserved, Read As Zero, each register can be implemented as a single word on either an 8-bit or a 32-bit ROM Table implementation. This means that the register map is identical on 8-bit and 32-bit implementations.

---

In a ROM Table implementation:

- The Component class field, bits [7:4], of the Component ID1 register is 0x1, identifying the component as a ROM Table.
- The 4KB count field, bits [7:4], of the Peripheral ID4 register must be 0. This is because a ROM Table must occupy a single 4KB block of memory.

When a ROM Table is implemented as part of a standard component, the following fields of the Peripheral ID must be available for customer modification:

- JEP-106 continuation code (4 bits)
- JEP-106 identity code (7 bits)
- Part number (12 bits)
- Revision (4 bits)
- Customer modified (4 bits).

For more information about these fields, see *The Peripheral ID Registers* on page 13-9.

### 14.4.1 Identifying the debug SoC or platform

The ROM Table Peripheral ID Registers uniquely identify the SoC or platform. If a system has more than one Memory Access Port then the information from the Peripheral ID Registers of all of the MEM-APs, considered collectively, is required to uniquely identify the SoC or platform. An example of a system with multiple MEM-APs is shown in Figure 1-5 on page 1-9.

---

### Note

Where a MEM-AP bus connects to a ROM Table hierarchy, it is the Peripheral ID Registers of the top-level ROM Table that are used to identify the SoC or platform.

Hierarchies of ROM Tables are described in *ROM Table hierarchies* on page 14-12.

---

If there is any change in the set of components identified by the ROM Table, or any change in the connections to those components, then the Peripheral ID Registers must be updated to reflect the change.

Each possible configuration must be uniquely identifiable from the Peripheral ID Register values. This is because the Peripheral ID can be used by the debugger to name the description of the system.



When a debugger performs topology detection on the system that it connects to through a Debug Interface, it can save its description of the system with the Peripheral ID. If that system is connected to the debugger again, the debugger can retrieve that saved description, avoiding any requirement for topology detection.

If two different systems have the same Peripheral ID a debugger might retrieve an incorrect description. If this situation occurs, the user must force the debugger to perform topology detection again.

## 14.5 ROM Table hierarchies

Normally, each ROM Table entry points to the memory space of a debug component. The Component and Peripheral ID Registers for that component start at offset 0xFD0 from the base address of the component, as describe in *Component descriptions and the component base address* on page 14-6. The Component class field, bits [7:4], of the Component ID1 Register identify the type of the component. This field is described in *The Component ID Registers* on page 13-4.

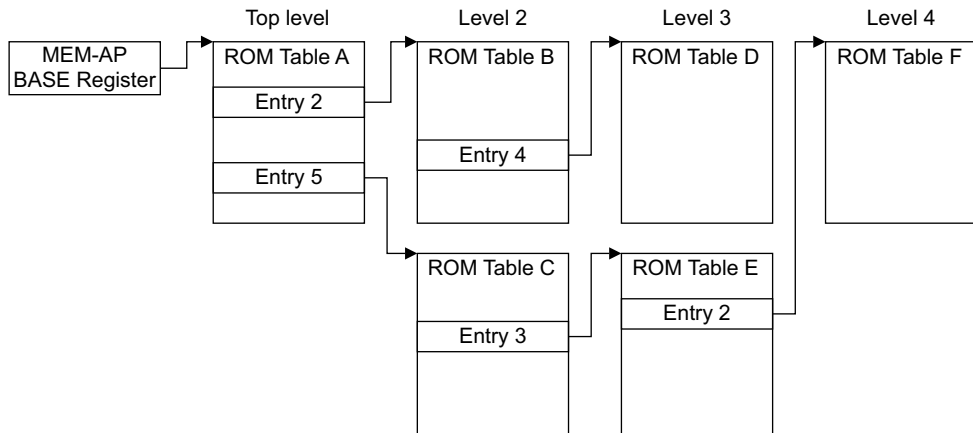
A ROM Table entry can point to another ROM Table. In this case:

- The Component class field of the Component ID1 Register in the memory area indicated by the ROM Table Entry is 0x1, indicating that the component is another ROM Table.
- The top level ROM Table and the second-level ROM Table must both be examined to discover all the debug components in the system.

A ROM table at any level can include entries that point to lower-level ROM Tables. Also, any ROM Table can include more than one entry that points to lower-level ROM Tables. This means that a hierarchy of ROM Tables can exist. All ROM Tables within that hierarchy must be scanned to discover all of the debug components in the system.

The MEM-AP BASE Register must point to the top level ROM Table in the hierarchy, see *Debug Base Address Register (BASE)* on page 11-11.

Figure 14-3 shows an example of a ROM Table hierarchy.



**Figure 14-3 ROM Table hierarchy example**

A hierarchy of ROM Tables might be used to increase the total number of ROM Table entries in the system. These limits are:

- 960 entries in a 32-bit ROM Table
- 240 entries in an 8-bit ROM Table.

However, a hierarchy might be implemented for some other reason, for example to reflect the logical organization of the debug components of the system. There might be only a few entries in each ROM Table within a hierarchy.

### 14.5.1 Peripheral ID Registers in lower-level ROM Tables

The Peripheral ID value obtained from the Peripheral ID Registers of any ROM Table that is not a top-level ROM Table is not used by the debugger, and does not have to be unique. This applies to the Peripheral ID of all ROM Tables that are only accessed through other ROM Tables.

The contents of the Peripheral ID Registers in lower-level ROM Tables might be:

- set to a Peripheral ID value that represents the subsystem described by the ROM Table, enabling that subsystem to be implemented as the only component of another debug system
- set to the Peripheral ID value of the top-level ROM Table
- set to a value reserved by the implementor to indicate a lower-level ROM Table.

These are only examples of the values that might be used for the Peripheral ID Registers of lower-level ROM Tables. This specification does place any requirement on the values used for these registers.

---

#### Note

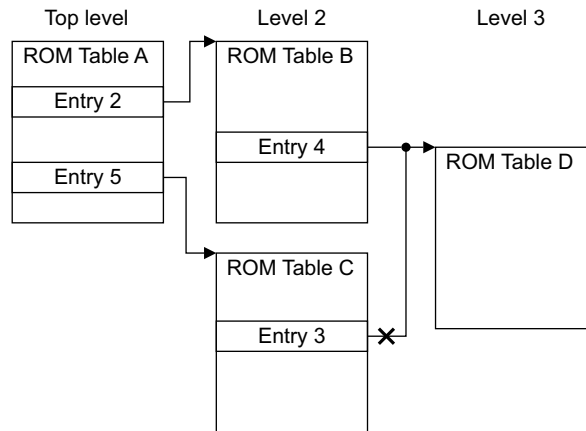
---

The Component ID Registers of lower-level ROM Tables must be implemented as described in *The Component ID Registers* on page 13-4. It is particularly important that the Component class field of the Component ID1 Register correctly identifies the component as a ROM Table.

---

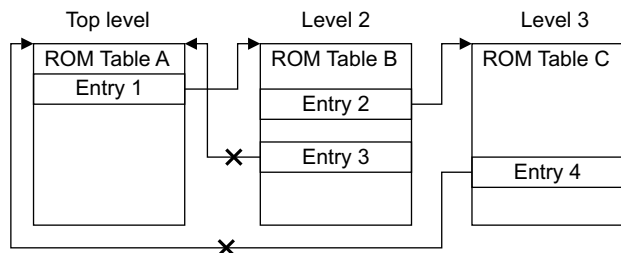
### 14.5.2 Prohibited ROM Table references

Every debug component within a system must appear only once in the ROM Table, or ROM Table hierarchy, that is visible to an external debugger. Figure 14-4 shows a prohibited case, where entries in ROM Tables B and C both point to ROM Table D.



**Figure 14-4 Prohibited duplicate ROM Table reference**

In addition, circular ROM Table references are prohibited. This means that a ROM Table entry must not point to a ROM Table that directly or indirectly points to itself. In particular, ROM Table entries must not point back to the top-level ROM Table. This is shown in Figure 14-5, where both ROM Table B and ROM Table C have prohibited links back to ROM Table A.



**Figure 14-5 Prohibited circular ROM Table references**

# Glossary

This glossary describes some of the terms used in technical documents from ARM Limited.

**Abort** A mechanism that indicates to a core that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch or Data Abort, and an internal or External Abort.

*See also* Data Abort, External Abort and Prefetch Abort.

**ADI** *See* ARM Debug Interface

**Advanced eXtensible Interface (AXI)**

A bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.

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

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

**Advanced High-performance Bus (AHB)**

A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave IP developments and ARM Limited recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

*See also* Advanced Microcontroller Bus Architecture.

**Advanced Microcontroller Bus Architecture (AMBA)**

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM Limited open standard for on-chip buses. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

**Advanced Peripheral Bus (APB)**

A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

**Advanced Trace Bus (ATB)**

A bus used by trace devices to share CoreSight capture resources.

**AHB** *See* Advanced High-performance Bus.

**Aligned** A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

**AMBA** *See* Advanced Microcontroller Bus Architecture.

**APB** *See* Advanced Peripheral Bus.

**Application Specific Integrated Circuit (ASIC)**

An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

**Architecture**

The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

**ARM Debug Interface (ADI)**

Version 5 of the ARM Debug Interface is defined by this specification. *See About the ARM Debug Interface version 5 (ADIv5)* on page 1-2 for a summary of earlier versions of the ARM Debug Interface.

**ASIC** *See* Application Specific Integrated Circuit.

**ATB** *See* Advanced Trace Bus.

**AXI**      *See* Advanced eXtensible Interface.

### **Big-endian**

Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

*See also* Little-endian and Endianness.

### **Big-endian memory**

Memory in which:

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

*See also* Little-endian memory.

### **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. Processors can contain several shift registers to enable you to access selected parts of the device.

**Burst**      A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AMBA are controlled using signals to indicate the length of the burst and how the addresses are incremented.

**Byte**      An 8-bit data item.

### **Cold reset**

Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required.

*See also* Warm reset.

**Core**      A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

**DAP**      *See* Debug Access Port.

### **Data Abort**

An indication from a memory system to the core of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.

*See also* Abort, External Abort, and Prefetch Abort.

### **DBGTAP**

*See* Debug Test Access Port.

### **Debug Access Port (DAP)**

A TAP block that acts as an AMBA, AHB or AHB-Lite, master for access to a system bus. The DAP is the term used to encompass a set of modular blocks that support system wide debug. The DAP is a modular component, intended to be extendable to support optional access to multiple systems such as memory mapped AHB and CoreSight APB through a single debug interface.

### **Debugger**

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

### **Debug Test Access Port (DBGTAP)**

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 **DBGTDI**, **DBGTDO**, **DBGTMS**, and **TCK**. The optional terminal is **DBGTRSTn**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

### **Direct Memory Access (DMA)**

An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

**DMA**     *See* Direct Memory Access.

**ECT**     *See* Embedded Cross Trigger.

### **Embedded Cross Trigger (ECT)**

The ECT is a modular component to support the interaction and synchronization of multiple triggering events with an SoC.

### **Embedded Trace Macrocell (ETM)**

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

### **Endianness**

Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

*See also* Little-endian and Big-endian

**ETM**     *See* Embedded Trace Macrocell.

**Event**     An observable condition that can be used by an ETM to control aspects of a trace.

### **Exception**

A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.



**External Abort**

An indication from an external memory system to a core that the value associated with a memory access is invalid. An external abort is caused by the external memory system as a result of attempting to access invalid memory.

*See also* Abort, Data Abort and Prefetch Abort.

**Halfword**

A 16-bit data item.

**Host**

A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

**IEM**

*See* Intelligent Energy Management.

**IMPLEMENTATION DEFINED**

The behavior is not architecturally defined, but is defined and documented by individual implementations.

**Instrumentation trace**

A component for debugging real-time systems through a simple memory-mapped trace interface, providing printf style debugging.

**Intelligent Energy Management (IEM)**

A technology that enables dynamic voltage scaling and clock frequency variation to be used to reduce power consumption in a device.

**Interrupt handler**

A program that control of the processor is passed to when an interrupt occurs.

**Joint Test Action Group (JTAG)**

The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

**JTAG**

*See* Joint Test Action Group.

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

An optional component of the DAP that provides JTAG access to on-chip components, operating as a JTAG master port to drive JTAG chains throughout a SoC.

**JTAG-AP**

*See* JTAG Access Port.

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

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

**JTAG-DP**

*See* JTAG Debug Port.

### **Little-endian**

Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

*See also* Big-endian and Endianness.

### **Little-endian memory**

Memory in which:

- a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

*See also* Big-endian memory.

### **Macrocell**

A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.

### **Microprocessor**

*See* Processor.

### **Power-on reset**

*See* Cold reset.

### **Prefetch Abort**

An indication from a memory system to the core that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

*See also* Data Abort, External Abort and Abort.

### **RAZ**

*See* Read As Zero.

### **Read As Zero (RAZ)**

On a read operation, always returns a value of 0, or of all zeros for a bit field. The RAZ description can be applied to:

- A bit field in a register. The field always reads as 0, or as all zeros, when the register is read.
- A register, or a word address in a register map. The 32-bit word at the RAZ address always reads as 0x0000 0000.

**Region** A partition of instruction or data memory space.

### **Reserved**

A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

**SBZ**      *See* Should Be Zero.

**SBZP**      *See* Should Be Zero or Preserved.

### 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.

### Should Be Zero (SBZ)

Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces UNPREDICTABLE results.

### Should Be Zero or Preserved (SBZP)

Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

### Serial-Wire Debug Port (SW-DP)

An optional external interface for the DAP that provides a low pin count bidirectional serial debug interface.

**SW-DP**      *See* Serial-Wire Debug Port

**TAP**      *See* Test Access Port.

### 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**. The optional terminal is **TRST\***. This signal is mandatory in ARM cores because it is used to reset the debug logic.

### Trace port

A port on a device, such as a processor or ASIC, used to output trace information.

### Unaligned

A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

### Undefined

Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more details on ARM exceptions.

**UNP**      *See* UNPREDICTABLE.

### UNPREDICTABLE

For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. UNPREDICTABLE behavior must not present security holes, and must not halt or hang the processor, or any part of the system.

### Warm reset

Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

**WI** Writes ignored.

**Word** A 32-bit data item.

**Word-invariant**

In a word-invariant system, the address of each byte of memory changes when switching between little-endian and big-endian operation, in such a way that the byte with address  $A$  in one endianness has address  $A \oplus 3$  in the other endianness. As a result, each aligned word of memory always consists of the same four bytes of memory in the same order, regardless of endianness. The change of endianness occurs because of the change to the byte addresses, not because the bytes are rearranged.

The ARM architecture supports word-invariant systems in ARMv3 and later versions. When word-invariant support is selected, the behavior of load or store instructions that are given unaligned addresses is instruction-specific, and is in general not the expected behavior for an unaligned access. It is recommended that word-invariant systems use the endianness that produces the desired byte addresses at all times, apart possibly from very early in their reset handlers before they have set up the endianness, and that this early part of the reset handler must use only aligned word memory accesses.

*See also* Byte-invariant.