ARM® Debug Interface Architecture Specification

ADIv5.0 to ADIv5.2

ARM®

ARM IHI 0031C (ID080813)

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

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
8 February 2006	A	Non-Confidential	First issue, for ADIv5.
14 May 2012	B	Confidential Beta	Second issue, for ADIv5.0 to ADIv5.2.
08 August 2013	C	Non-Confidential Final	Third issue, for ADIv5.0 to ADIv5.2.

This book includes information originally published in the *ARM® Debug Interface v5 Architecture Specification ADIv5.1 Supplement* (DSA09-PRDC-008772).

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

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

Preface

About this book x
Using this book xi
Conventions xii
Additional reading xiv
Feedback xv

Chapter 1 Introduction

1.1 About the ARM Debug Interface version 5 (ADIv5) 1-18
1.2 The function of the ARM Debug Interface 1-20
1.3 The subdivisions of an ARM Debug Interface v5 implementation 1-22
1.4 The external interface, the Debug Port (DP) 1-23
1.5 The resource interface, the Access Ports (APs) 1-24
1.6 Design choices and implementation examples 1-28

Chapter 2 The Debug Port (DP)

2.1 Common Debug Port features 2-34
2.2 DP architecture versions 2-40
2.3 DP register descriptions 2-45
2.4 System and debug power and debug reset control 2-62

Chapter 3 The JTAG Debug Port (JTAG-DP)

3.1 The Debug TAP State Machine introduction 3-70
3.2 The scan chain interface 3-71
3.3 IR scan chain and IR instructions 3-74
3.4 DR scan chain and DR registers 3-78

Chapter 4 **The Serial Wire Debug Port (SW-DP)**
4.1 Introduction to the Serial Wire Debug Port 4-88
4.2 Introduction to the ARM Serial Wire Debug (SWD) protocol 4-89
4.3 Serial Wire Debug protocol operation 4-93
4.4 Serial Wire Debug interface 4-104

Chapter 5 **The Serial Wire/JTAG Debug Port (SWJ-DP)**
5.1 Serial Wire/JTAG Debug Port (SWJ-DP) 5-108
5.2 SWD and JTAG select mechanism 5-110
5.3 Dormant operation 5-113
5.4 Restriction on switching 5-120

Chapter 6 **The Access Port (AP)**
6.1 Overview of Access Ports (APs) 6-122
6.2 Selecting and accessing an AP 6-123
6.3 The Programmers' Model for Access Port (AP) registers 6-124

Chapter 7 **The Memory Access Port (MEM-AP)**
7.1 About the function of a Memory Access Port (MEM-AP) 7-128
7.2 MEM-AP functions 7-132
7.3 Implementing a MEM-AP 7-142
7.4 MEM-AP examples of pushed-verify and pushed-compare 7-144
7.5 MEM-AP register summary 7-146
7.6 MEM-AP register descriptions 7-147

Chapter 8 **The JTAG Access Port (JTAG-AP)**
8.1 Overview of the JTAG Access Port (JTAG-AP) 8-162
8.2 Operation of the JTAG-AP 8-166
8.3 The JTAG Engine Byte Command Protocol 8-169
8.4 JTAG-AP register summary 8-176
8.5 JTAG-AP register descriptions 8-177

Chapter 9 **Component and Peripheral ID Registers**
9.1 Component and Peripheral ID registers 9-186
9.2 The Component ID Registers 9-187
9.3 The Peripheral ID Registers 9-191

Chapter 10 **ROM Tables**
10.1 ROM Table overview 10-200
10.2 ROM Table entries 10-202
10.3 The MEMTYPE register 10-205
10.4 Component and Peripheral ID Registers 10-206
10.5 ROM Table hierarchies 10-207

Appendix A **Standard Memory Access Port Definitions**
A.1 Introduction A-212
A.2 AMBA AXI3 and AXI4 .... A-213
A.3 AMBA AXI4 with ACE-Lite A-215
A.4 AMBA AHB A-217
A.5 AMBA APB2 and APB3 .... A-218

Appendix B **Cross-over with the ARM Architecture**
B.1 Introduction B-220
B.2 ARMv6-M architecture B-221
B.3 ARMv7-M architecture profile B-222
B.4 ARMv7-A without Large Physical Address Extension and ARMv7-R architecture profiles
B-223

B.5 ARMv7-A with Large Physical Address Extension and ARMv8-A architecture profiles . B-224
B.6 Summary of the required ADIv5 implementations B-225

Appendix C Pseudocode Definition
C.1 About ARM pseudocode C-228
C.2 Data types C-229
C.3 Expressions C-233
C.4 Operators and built-in functions C-235
C.5 Statements and program structure C-240

Glossary

Preface

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

- *About this book* on page x.
- *Using this book* on page xi.
- *Conventions* on page xii.
- *Additional reading* on page xiv.
- *Feedback* on page xv.

About this book

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

Intended audience

This specification is written for system designers and engineers who 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) that incorporates a debug interface that complies with the ADIv5 specification.

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.
- Specifying, designing or writing debug software.

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

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

Using this book

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 ***The Debug Port (DP)***

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 3 ***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 access the JTAG-DP.

Chapter 4 ***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 5 ***The Serial Wire/JTAG Debug Port (SWJ-DP)***

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

Chapter 6 ***The Access Port (AP)***

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

Chapter 7 ***The Memory Access Port (MEM-AP)***

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

Chapter 8 ***The JTAG Access Port (JTAG-AP)***

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

Chapter 9 ***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 specification.

Chapter 10 ***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.

Appendix A ***Standard Memory Access Port Definitions***

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

Appendix B ***Cross-over with the ARM Architecture***

Read this appendix for a description of the required or recommended options for the ARM Debug Interface for ARMv6-M and all ARMv7 architecture profiles.

Appendix C ***Pseudocode Definition***

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

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

Conventions

The following sections describe conventions that this specification can use:

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

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

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

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

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

SMALL CAPITALS

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

Colored text Indicates a link. This can be:

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

Signals

The signal conventions are:

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

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

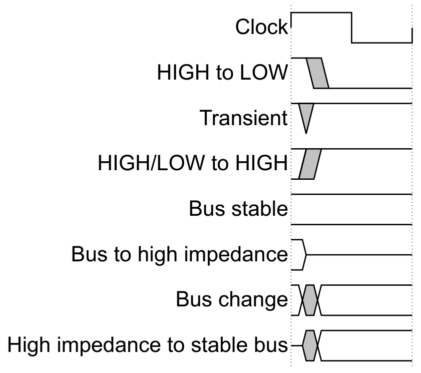
Lower-case n At the start or end of a signal name denotes an active-LOW signal.

Prefix DBG Denotes debug signals.

Timing diagrams

The figure named *Key to timing diagram conventions* on page xiii 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

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

Numbers

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

Pseudocode descriptions

This specification uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a `monospace` font, and is described in Appendix C *Pseudocode Definition*.

Additional reading

This section lists relevant publications from ARM and third parties.

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

ARM publications

See the following documents for other information that is relevant to this ADIv5 specification:

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014).
- *CoreSight Architecture Specification* (ARM IHI 0029).
- *CoreSight SoC Technical Reference Manual* (ARM DDI 0480).
- *AMBA® Specification (Rev 2.0)* (ARM IHI 0011).
- *AMBA® AXI™ and ACE™ Protocol Specification AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™* (ARM IHI 0022).
- *AMBA® 3 AHB-Lite™ Protocol Specification* (ARM IHI 0033).
- *AMBA® 4 APB™ Protocol Specification* (ARM IHI 0024).
- *ARM1136JF-S™ and ARM1136J-S™ Technical Reference Manual* (ARM DDI 0211).

Other publications

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

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

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, ARM IHI 0031C.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Chapter 1 Introduction

This chapter introduces the ARM Debug Interface architecture and summarizes the design decisions required for an ARM Debug Interface implementation. It also describes how the *Debug Access Port* (DAP) accesses 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. It contains the following sections:

- *About the ARM Debug Interface version 5 (ADIv5)* on page 1-18.
- *The function of the ARM Debug Interface* on page 1-20.
- *The subdivisions of an ARM Debug Interface v5 implementation* on page 1-22.
- *The external interface, the Debug Port (DP)* on page 1-23.
- *The resource interface, the Access Ports (APs)* on page 1-24.
- *Design choices and implementation examples* on page 1-28.

1.1 About the ARM Debug Interface version 5 (ADIv5)

The terms *ARM Debug Interface v5* and *ADIv5* refer to the major revision of ADI, that is, to v5.0, and also to v5.1, or any other revision of 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 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 Debug interface.

The main components of ADIv5 are split between two main architectures:
- The *Access Port* (AP) architecture.
- The *Debug Port* (DP) architecture.

1.1.1 About the versions of ADIv5

ADI versions 5.1 and 5.2 are backwards-compatible extensions of the original *ARM Debug Interface Architecture Specification* v5. From the introduction of ADIv5.1, the original ADIv5 specification is described as ADIv5.0.

Note

ADIv5.1 was originally defined only by a supplement to the original *ARM Debug Interface v5 Architecture Specification*, issue A of this document. Issue B of this document, ARM IHI 0031B, is the first publication of an integrated description of ADIv5.0, ADIv5.1, and ADIv5.2.

Features defined by ADIv5.0

ADIv5.0 defines:

- Two Debug Ports, JTAG-DP and SW-DP.
- Two Access Ports:
  - JTAG-AP, for accessing legacy JTAG components.

  - MEM-AP, for accessing memory and components with memory-mapped interfaces.
- The identification model for Access Ports.
- A discovery mechanism for components attached to a MEM-AP.

Features added in v5.1

ADIv5.1 formalizes version numbering of Debug Ports, and defines:

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

ADIv5.1 also adds:
- Extensions to the AP identification model.
- Standard definitions for MEM-AP implementations for AMBA bus protocols.
- JTAG support in Debug Port architecture version 1.
- The Minimal Debug Port extension.
- Debug Port architecture version 2 (DPv2).
- Multiple protocol interoperability.
- Serial Wire Debug protocol version 2, that provides a multi-drop capability.

The following subsections give more information about ADIv5 features added in ADIv5.1.

Multiple Protocol Interoperability

ADIv5.1 introduces multiple protocol interoperability extensions, that provide:

- Simple switching between Serial Wire Debug and JTAG protocols. For more information see Chapter 5 *The Serial Wire/JTAG Debug Port (SWJ-DP)*.
- A dormant state, for interoperability with other protocols. For more information see *Dormant operation* on page 5-113.

Serial Wire Debug protocol extensions

ADIv5.1 introduces Serial Wire Debug protocol version 2 that extends the Serial Wire Debug protocol, adding multi-drop capability. See Chapter 4 *The Serial Wire Debug Port (SW-DP)*.

Minimal Debug Port extension

The *Minimal Debug Port* (MINDP) is a simplified version of the Debug Port intended for low gate-count implementations. See *MINDP, Minimal Debug Port extension* on page 2-36.

Features added in ADIv5.2

ADIv5.2 includes:

- Extensions to the MEM-AP programmers' model to support:
  - Large physical address spaces of up to 64 bits.
  - Data sizes greater than 32 bits.
  - Barrier operations.
- Extended definitions of MEM-AP to include extensions for:
  - AMBA AXI3, AXI4, and AXI4-Lite.
  - AMBA ACE.
- Recommended implementations for:
  - ARMv7-A processors including the Large Physical Address Extension.
  - ARMv8-A processors.

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*.

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

See *Compatibility between CoreSight and ARM Debug interfaces* on page 1-21 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 can:

- 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, ARMv7-A and ARMv7-R edition.*
- The Embedded Trace Macrocell, see the *ETM Architecture Specification*.

In addition, the ARM Debug Interface v5 can 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.

As far as possible, functions operate with minimal change to the behavior of the system being debugged.

Examples of technologies that provide these facilities are:

- The ADIv5 *Debug Access Port* (DAP).
- The CoreSight debug architecture. For more information see the *CoreSight Architecture Specification*.
- The CoreSight components. For more information see the *CoreSight SoC 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 debug components to be CoreSight-compliant.

1.3 The subdivisions of an ARM Debug Interface v5 implementation

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

- A number of registers that are private to the interface. 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 DAP is connected.

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

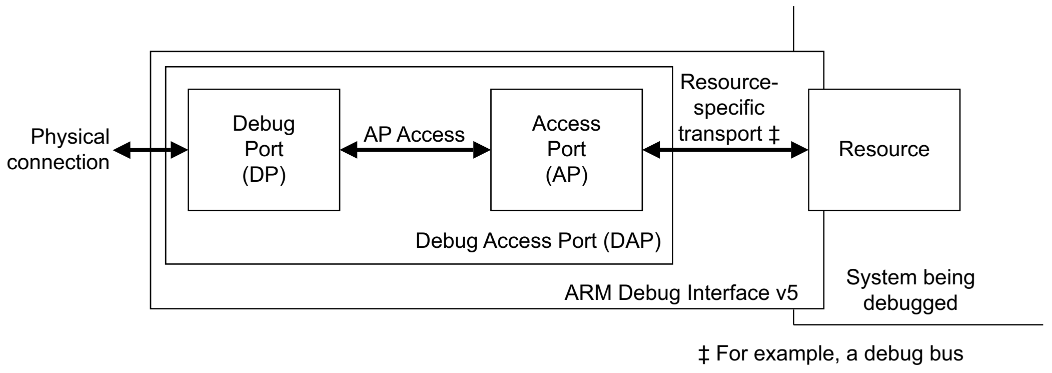


Figure 1-1 Subdivisions of an ARM Debug Interface v5 implementation

Because the DAP logically consists of two parts, the Debug Port and the 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 the Access Port (AP) registers. This is provided by Access Port accesses (APACC).

A DAP 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 1-23.

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 10 *ROM Tables*.
- A memory system.
- A legacy JTAG device.

Note

Although this specification logically divides the ADIv5 into the elements shown in Figure 1-1, this specification does not require implementations to be structured in this way. This specification describes the programmers' model for the ADI, and these subdivisions give a convenient representation of the programmers' model.

1.4 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 the following 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/JTAG Debug Port (SWJ-DP)*.
- The Serial Wire/JTAG Debug Port (SWJ-DP), see Chapter 5 *The Serial Wire/JTAG Debug Port (SWJ-DP)*.

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 failed.

1.4.1 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 1-1:

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

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

The signal naming conventions of IEEE 1149.1 are modified in a similar way, for example the IEEE 1149.1 **TDI** signal is named **DBGTDI** on a JTAG Debug Port. See *Physical connection to the JTAG-DP* on page 3-71 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 read or write register information. The DBGTAP scan chains are described in more detail in Chapter 3 *The JTAG Debug Port (JTAG-DP)*.

1.4.2 The Serial Wire Debug Port (SW-DP)

The SW-DP is a two-pin serial interface that uses a packet-based protocol to read or write registers.

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 4 *The Serial Wire Debug Port (SW-DP)*.

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

The SWJ-DP interface provides a mechanism to select between *Serial Wire Debug* (SWD) and JTAG Data Link protocols. This enables the JTAG-DP and SW-DP to share pins.

1.5 The resource interface, the Access Ports (APs)

An Access Port provides the interface between a DAP and one or more debug components. That is, the Access Port of the DAP 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 ADIv5 components defined by this specification include two alternatives for the resource-specific transport that connects the DAP to the system being debugged:

- A DAP can connect to a memory-mapped resource, such as a debug peripheral. For such connections, ADIv5 defines a *Memory Access Port* (MEM-AP) programmers' model.
- A DAP can connect to a legacy IEEE 1149.1 JTAG device. For such connections, ADIv5 defines a *JTAG Access Port* (JTAG-AP) and associated programmers' model.

The complete description of these is summarized in:

- *Guide to the detailed description of a MEM-AP* on page 1-26.
- *Guide to the detailed description of a JTAG-AP* on page 1-27.

This specification does not specify exact requirements about 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, ADIv5 might be directly integrated into the resource.

However, referring to Figure 1-1 on page 1-22, 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*.

Note

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

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 6 *The Access Port (AP)*.

As described:

- The simplest ADI has only one AP. This can be either a MEM-AP or a JTAG-AP.
- More 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. *Using the Access Port to access debug resources* on page 1-27 describes how to access any AP (MEM-AP or JTAG-AP), and summarizes how this gives access to resources in the system being debugged.

1.5.1 Using the Debug Port to access Access Ports

Figure 1-2 on page 1-26 shows the different levels between the physical connection to the debugger and the debug resources of the system being debugged. These levels are designed to enable efficient access to the system being debugged, and several levels provide registers within the DAP. This section describes how these register accesses are implemented.

The DAP supports two types of 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 4 *The Serial Wire Debug Port (SW-DP)*.

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

- For a DP register access, the address bits A[3:2] and SELECT.DPBANKSEL determine which register is accessed. SELECT is a DP register.
- For an AP register access, SELECT.APSEL selects an AP to access, and the address bits A[3:2] are combined with SELECT.APBANKSEL to determine which AP register is accessed, as summarized in Figure 1-2 on page 1-26. That is, the two address bits A[3:2] are decoded to select one of the four 32-bit words from the register bank indicated by SELECT.APBANKSEL in the AP indicated by SELECT.APSEL.

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

For example, to access the register at address `0x14` in AP number 0, the debugger must:

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

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

This access model is shown in Figure 1-2 on page 1-26. 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 7-1 on page 7-129, for a MEM-AP implementation.
- Figure 8-1 on page 8-162, for a JTAG-AP implementation.

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

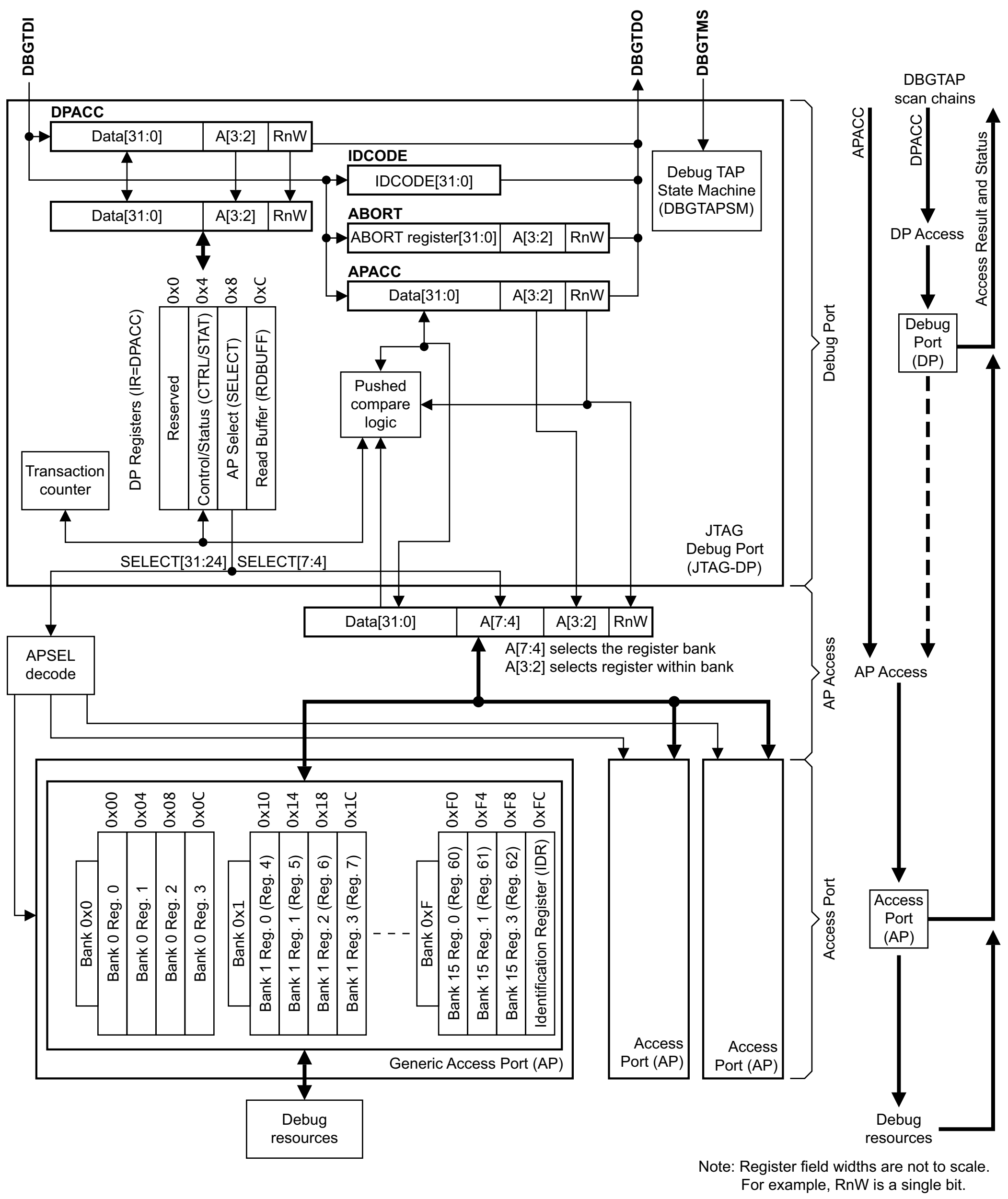


Figure 1-2 Structure of the Debug Access Port, showing DPv0 JTAG-DP accesses to a generic AP

1.5.2 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 6 *The Access Port (AP)*.
- Chapter 7 *The Memory Access Port (MEM-AP)*.

The MEM-AP provides access to zero, one, or more debug components. Any debug component that complies with the ARM Generic Identification Registers specification implements a set of Component Identification Registers. These are described in Chapter 9 *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 10 *ROM Tables*.

Note

As shown in *Design choices and implementation examples* on page 1-28, a system with only one functional debug component might also implement a ROM Table.

1.5.3 Guide to the detailed description of a JTAG-AP

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

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

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

- Chapter 6 *The Access Port (AP)*.
- Chapter 8 *The JTAG Access Port (JTAG-AP)*.

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.

1.5.4 Using the Access Port to access debug resources

Accessing the AP gives access to the system being debugged, shown as access to *Debug resources* in Figure 1-2 on page 1-26. 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 7 *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 7-130.

- 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 8 *The JTAG Access Port (JTAG-AP)*.

1.6 Design choices and implementation examples

The subdivisions of an ARM Debug Interface v5 implementation on page 1-22 introduces the logical subdivisions of an ARM Debug Interface, shown in Figure 1-1 on page 1-22. 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).

Note

This Architecture Specification is written for engineers implementing 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 implementer of the Debug Interface.

1.6.1 Choices for the Debug Port (DP)

An ADI Debug Access Port has a single Debug Port. The following subsections summarize the implementation options for the DP:

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

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

Note

- In illustrations of the ARM Debug Interface, the Debug Port can be any of the following three options: Serial Wire Debug Port, a JTAG Debug Port, or a Serial Wire/JTAG Debug Port.
- ARM might define other DPs in the future.

1.6.2 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. Typical 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.
- Using a JTAG-AP to connect to a single legacy IEEE 1149.1 device, as shown in Figure 1-4 on page 1-29.

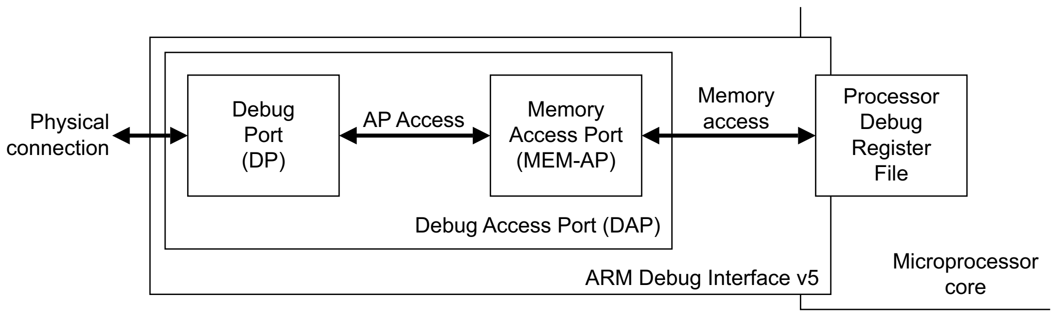


Figure 1-3 Simple ARM Debug Interface MEM-AP Implementation

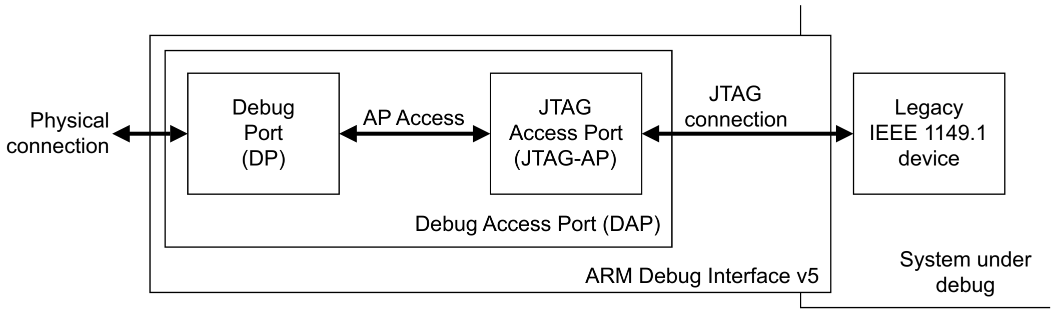


Figure 1-4 Simple ARM Debug Interface JTAG-AP Implementation

However, *ROM tables* on page 7-128 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 1-5.

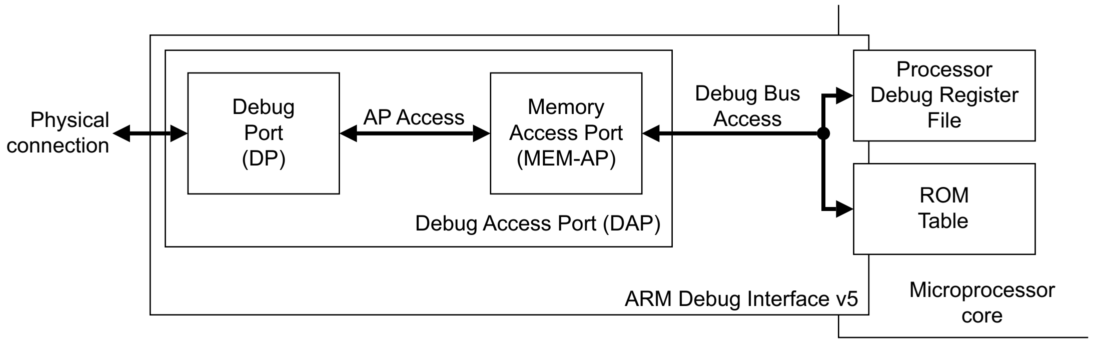


Figure 1-5 Simple example of an ARM Debug Interface implementation, with ROM Table

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 1-31.
  - *Choices for each MEM-AP* on page 1-30.

Top-level AP planning choices

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. 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 8 *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 1-6 shows a more complex ARM Debug Interface, and illustrates the different Access Port options.

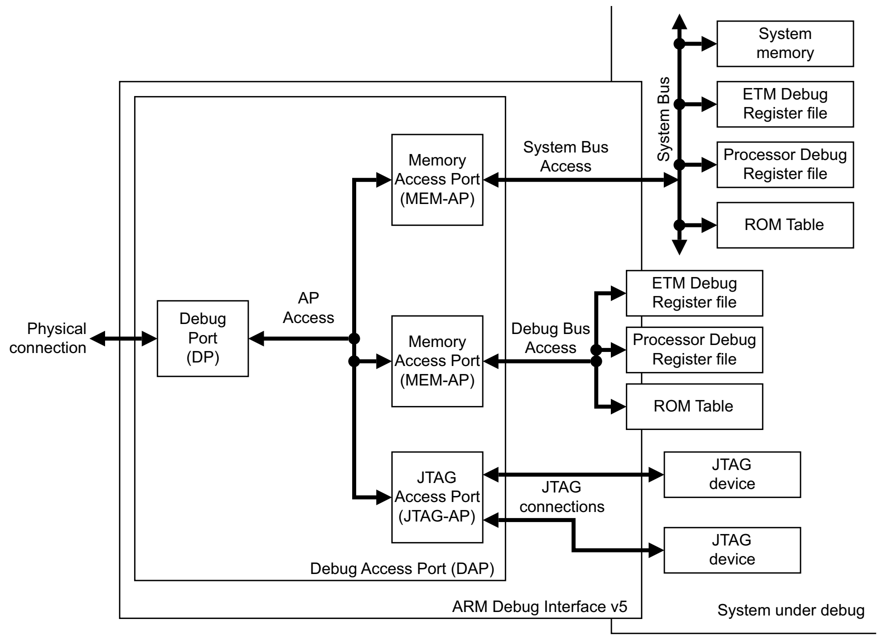


Figure 1-6 Complex ARM Debug Interface, showing the Access Port options

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 1-2 on page 1-26, the Debug Port can be any Debug Port defined by ADIv5.

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, and the width of the bus.
- The memory map of the MEM-AP address space.

If a 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 7-128 for more information.

It is IMPLEMENTATION DEFINED whether a MEM-AP includes certain features, for more information see:
- *MEM-AP functions* on page 7-132.
- *MEM-AP implementation requirements* on page 7-142.

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 7 *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 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:
- The number of JTAG scan chains connected to the JTAG-AP.
- Whether each scan chain contains one or more TAPs.

For more information, see Chapter 8 *The JTAG Access Port (JTAG-AP)*.

Chapter 2
The Debug Port (DP)

This chapter describes the features that are implemented by both Serial Wire Debug Ports (SW-DPs) and on JTAG Debug Ports (JTAG-DPs), and by the combined Serial Wire/JTAG Debug Port (SWJ-DP). It contains the following sections:

- *Common Debug Port features* on page 2-34.
- *DP architecture versions* on page 2-40.
- *DP register descriptions* on page 2-45.
- *System and debug power and debug reset control* on page 2-62.

Note

The following chapters give additional information about DPs:

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

2.1 Common Debug Port features

The following sections describe the features that are common to all Debug Port implementations:

- *Sticky flags and DP error responses*.
- *MINDP, Minimal Debug Port extension* on page 2-36.
- *Pushed-compare and pushed-verify operations* on page 2-36.
- *The transaction counter* on page 2-38.
- *Power and reset control* on page 2-38.

2.1.1 Sticky flags and DP error responses

Within the Debug Port, error conditions are recorded using sticky flags in the DP CTRL/STAT register. These error conditions are described in the following sections:

- *Read and write errors*.
- *Overrun detection*.
- *Protocol errors, SW-DP* on page 2-36.

In addition, if pushed transactions are supported, another sticky flag, CTRL/STAT.STICKYCMP, reports the result of pushed operations, see *Pushed-compare and pushed-verify operations* on page 2-36. CTRL/STAT.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. *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50 summarizes how these flags are cleared to 0.

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 after performing a series of APACC transactions, a debugger must check the CTRL/STAT register 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 determine the cause of the sticky flag condition. Because the flags are sticky, the debugger does not have to check the flags after every transaction. The debugger must only check the CTRL/STAT register periodically. This reduces the overhead of checking for errors. For details of how to clear sticky flags, see *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50.

Read and write errors

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

For example, a read or write error might be generated if the debugger makes an AP transaction request while the debug power domain is powered down. See *Power and reset control* on page 2-38 for information about power domains.

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 must 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 Sticky Overrun flag, CTRL/STAT.STICKYORUN to 1. In this overrun detection mode the debugger must check for overrun errors after each sequence of APACC transactions, by checking this flag.

Overrun detection mode is enabled by setting the Overrun Detect bit, CTRL/STAT.ORUNDETECT, to 1.

As the JTAG-DP and SW-DP protocols differ, the exact behavior in overrun detection mode is DATA LINK DEFINED:

JTAG-DP The Sticky Overrun flag, CTRL/STAT.STICKYORUN is set to 1 if the response to any transaction is other than OK/FAULT.

The response is WAIT until the previous Access Port (AP) transaction completes. Following responses are OK/FAULT. See *Sticky overrun behavior on DPACC and APACC accesses* on page 3-83.

SW-DP The Sticky Overrun flag, CTRL/STAT.STICKYORUN is set to 1 if the response to any transaction is other than OK.

The first response to a transaction when a previous AP transaction has not completed is WAIT. Following responses are FAULT, because the STICKYORUN bit is set to 1. See *Sticky overrun behavior* on page 4-98.

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

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

Note

The method of clearing the STICKYORUN flag to 0 is different for a JTAG-DP and a SW-DP. See *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50 for more information.

The behavior of the Debug Port when the Sticky Overrun flag is set to 1 is DATA LINK 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 UNKNOWN. To leave overrun detection mode, a debugger must:

- Check the value of the CTRL/STAT.STICKYORUN bit.
- Clear the STICKYORUN bit to 0, if it is set to 1.
- Clear the ORUNDETECT bit to 0, to disable overrun detection mode.

Protocol errors, SW-DP

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 SW-DP:

- If the SW-DP detects a protocol error in the packet request, the Debug Port does not respond to the message.
- If the SW-DP detects a parity error in the data phase of a write transaction, it sets the Sticky Write Data Error flag, CTRL/STAT.WDATAERR, to 1. This sticky flag is treated in the same way as the other sticky flags described in this section.

For more information see *Parity* on page 4-91 and *Protocol error response* on page 4-97.

2.1.2 MINDP, Minimal Debug Port extension

The *Minimal Debug Port* (MINDP) programmers' model is a is a simplified version of the Debug Port intended for low gate-count implementations. Minimal DP implementations must use DPv1 or later.

For MINDP implementations, the following Debug Port features are removed:

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

When MINDP is implemented:

- MIN, bit[16] of the Debug Port ID Register, DPIDR is RAO.
- The following fields of the CTRL/STAT register, are reserved, RES0:
  - TRNCNT, bits[21:12].
  - MASKLANE, bits[11:8].
  - STICKYCMP, bit[4].
  - TRNMODE, bits[3:2].

  See CTRL/STAT.
- STKCMPCLR, bit[1] of AP ABORT, is reserved, SBZ. Writing 1 to this bit is UNPREDICTABLE.

2.1.3 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, CTRL/STAT.STICKYCMP 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 canceled.

These operations are described as pushed operations. For more information see CTRL/STAT.

Each AP defines which registers support pushed transactions. If an AP register does not support pushed transactions, or SELECT.APSEL selects an AP that is not present, then a pushed transaction sets STICKYCMP to an UNKNOWN value. Reserved AP registers and the common AP IDR do not support pushed transactions.

Pushed operations are enabled using the Transfer Mode bits, CTRL/STAT.TRNMODE.

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 CTRL/STAT.MASKLANE register. For more information about this masking, see *MASKLANE and the bit masking of the pushed-compare and pushed-verify operations* on page 2-49.

Figure 2-1 gives an overview of the pushed operations.

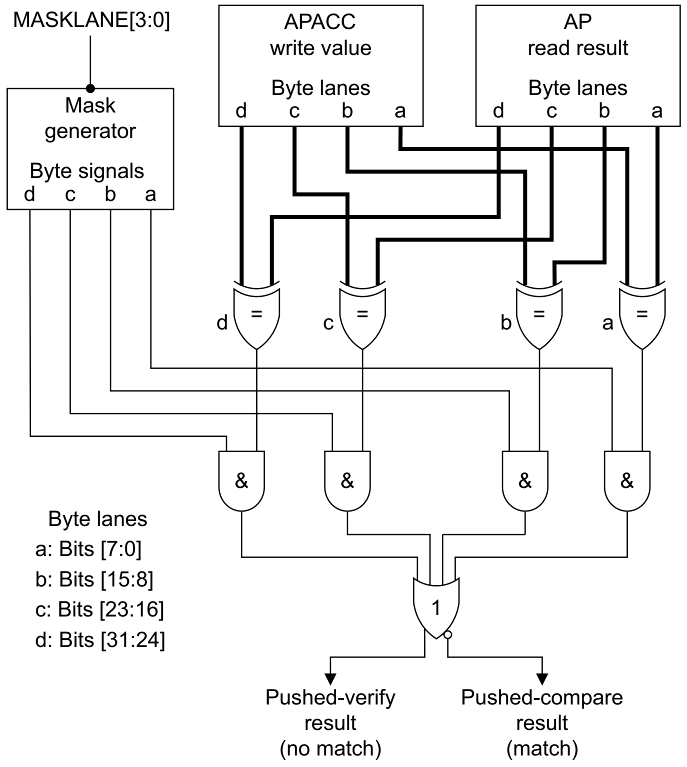


Figure 2-1 Pushed operations overview

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 1-2 on page 1-26.
- The AP reads from the address indicated by the MEM-AP *Transfer Address Register* (TAR).
- The value returned by this read is compared with the value held in the pushed-compare logic, and the CTRL/STAT.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 2-1. For more information about masking see *MASKLANE and the bit masking of the pushed-compare and pushed-verify operations* on page 2-49.

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 SW-DP, this means that performing an AP read transaction with pushed-compare or pushed-verify active returns an UNKNOWN value, and the read has UNPREDICTABLE side effects, although the wire-level protocol remains coherent.

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 CTRL/STAT.STICKYCMP to 1. This operation is described in more detail in *Example use of pushed-verify operation on a MEM-AP* on page 7-144.

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*. Again, this operation is described in more detail in Chapter 7 *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 7-145.

2.1.4 The transaction counter

Except in the implementation of the MINDP extension, a Debug Port must include an AP transaction counter, CTRL/STAT.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.

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

Examples of the use of the transaction counter are:

- For a memory fill operation, the transaction counter can 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 7-139.
- 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*, and is described in more detail in *Example using the transaction counter for a pushed-find operation on a MEM-AP* on page 7-145.

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

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 CTRL/STAT.STICKYERR flag is set to 1.
- The CTRL/STAT.STICKYCMP flag 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.

2.1.5 Power and reset control

The Debug Port provides:

- Control bits for system and debug power control.

- Control bits for debug reset control.

These control bits are programmable by the debugger, and drive signals into the target system. ADIv5 also recommends an external debug interface pin for system reset control. 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.

For more information, see *System and debug power and debug reset control* on page 2-62.

2.2 DP architecture versions

This section introduces the concept of Debug Port architecture versions and describes the *Debug Port* (DP) registers in compliant implementations. More information about Debug Ports is given in the following chapters:

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

2.2.1 DP architecture versions summary

Every ARM Debug Interface includes a single Debug Port (DP) that is compliant with one of the DP architecture versions. Table 2-1 shows the DP architecture versions.

Table 2-1 Debug Port architecture versions

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

Although the Debug Port architecture versions are different, their register sets are similar, and are described together in this chapter. In the section *DP architecture versions summary*, the detailed register descriptions always make clear if the register is implemented for a specific architecture, and if the implementation is DATA LINK DEFINED.

This section summarizes the Debug Port registers, and gives the register maps for the DP architecture versions DPv0, DPv1, and DP2. It contains the following sub-sections:

- *Debug Port (DP) registers summary*.
- *DP architecture version 0 (DPv0) address map* on page 2-41.
- *DP architecture version 1 (DPv1) address map* on page 2-42.
- *DP architecture version 2 (DPv2) address map* on page 2-43.

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

2.2.2 Debug Port (DP) registers summary

Table 2-2 summarizes the DP registers and shows if a register is implemented in a specific DP architecture version.

Table 2-2 Summary of Debug Port registers

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

Table 2-2 Summary of Debug Port registers (continued)

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

2.2.3 DP architecture version 0 (DPv0) address map

DPv0 provides support for JTAG-DP in ADIv5.

The JTAG-DP register accessed depends on both:

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

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

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

Table 2-3 DPv0 register map

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

a. Reserved, UNPREDICTABLE.

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

Accessing the JTAG-DP registers

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

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

These registers are shown in the illustration of the JTAG-DP in Figure 1-2 on page 1-26.

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

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

For more information about the JTAG-DP scan chains see Chapter 3 *The JTAG Debug Port (JTAG-DP)*.

2.2.4 DP architecture version 1 (DPv1) address map

DPv1 extends DPv0, adding support for Serial Wire Debug protocol version 1 and defining the following additional registers:

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

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

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

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

Table 2-4 shows the DPv1 register map.

Table 2-4 DPv1 register map

Address[a]	DPBANKSEL[b]	Name	Access	Notes
`0x0`	x	DPIDR	RO	-
		-	WO	DATA LINK DEFINED, as either: • ABORT, see *ABORT, AP Abort register* on page 2-45 • Reserved, UNPREDICTABLE
`0x4`	`0x0`	CTRL/STAT	RW	-
	`0x1`	DLCR	RW	-
	`0x2` to `0xF`	-	-	UNPREDICTABLE
`0x8`	x	-	RO	DATA LINK DEFINED
		SELECT	WO	-
`0xC`	x	RDBUFF	RO	-
		-	WO	DATA LINK DEFINED

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

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

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

SW-DP DATA LINK DEFINED registers, DPv1

Table 2-5 shows the DATA LINK DEFINED SW-DP registers for a DPv1 implementation. Bits[1:0] of the address are always `0b00`.

Table 2-5 SW-DP data link defined registers, DPv1

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

a. Reserved, SBZ.

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

2.2.5 DP architecture version 2 (DPv2) address map

DPv2 extends DPv1, adding support for Serial Wire Debug protocol version 2 and defining the following additional registers:

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

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

Table 2-6 shows the DPv2 register map.

Table 2-6 DPv2 address map

Address[a]	DPBANKSEL[b]	Name	Access	Notes
`0x0`	x	DPIDR	RO	-
		-	WO	DATA LINK DEFINED, as either: • ABORT, see *ABORT, AP Abort register* on page 2-45 • Reserved, RES0
`0x4`	`0x0`	CTRL/STAT	RW	-
	`0x1`	DLCR	RW	-
	`0x2`	TARGETID	RO	-
	`0x3`	DLPIDR	RO	-
	`0x4`	EVENTSTAT	RO	-
	All other values	-	-	Reserved, RES0
`0x8`	x	-	RO	DATA LINK DEFINED
		SELECT	WO	-

Table 2-6 DPv2 address map (continued)

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

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

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

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

SW-DP DATA LINK DEFINED registers, DPv2

Table 2-7 shows the DATA LINK DEFINED SW-DP registers for a DPv2 implementation:

Table 2-7 SW-DP data link defined registers, DPv2

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

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

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

2.2.6 Register maps, and accesses to reserved addresses

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

- Figure 1-2 on page 1-26, for accesses to JTAG-DP registers.
- Figure 7-1 on page 7-129, for accesses to MEM-AP registers.
- Figure 8-1 on page 8-162, for accesses to JTAG-AP registers.

There are a number of reserved addresses in these register maps. Reserved AP registers are RES0.

2.3 DP register descriptions

This section gives full descriptions of the DP registers.

The registers are listed in name order.

2.3.1 ABORT, AP Abort register

The AP Abort register attributes are:

Purpose The ABORT register forces an AP transaction abort.

In DPv1 and DPv2 only, the ABORT register has additional fields that clear error and sticky flag conditions. See *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50. In DPv0, these fields are reserved, SBZ.

Configurations

A DP architecture register. The ABORT register is defined and implemented in DPv0, DPv1, and DPv2.

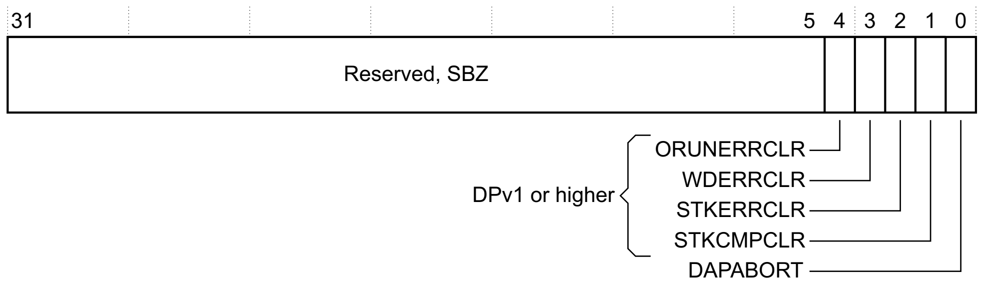
Attributes The ABORT register is:

- A write-only register.
- Accessed in a DATA LINK DEFINED manner:

  JTAG-DP Access is through its own scan-chain. See the JTAG-DP ABORT register.

  SW-DP Accessed by a write to offset `0x0` of the DP register map.
- Always accessible, completes all accesses on the first attempt, and returns an OK response if a valid transaction is received.

The ABORT register bit assignments are:



Bits[31:5] Reserved, SBZ.

ORUNERRCLR, bit[4], DPv1 or higher

Write 1 to this bit to clear the CTRL/STAT.STICKYORUN overrun error bit to 0.

WDERRCLR, bit[3], DPv1 or higher

Write 1 to this bit to clear the CTRL/STAT.WDATAERR write data error bit to 0.

STKERRCLR, bit[2], DPv1 or higher

Write 1 to this bit to clear the CTRL/STAT.STICKYERR sticky error bit to 0.

STKCMPCLR, bit[1], DPv1 or higher

Write 1 to this bit to clear the CTRL/STAT.STICKYCMP sticky compare bit to 0. It is IMPLEMENTATION DEFINED whether this bit is implemented. See *MINDP, Minimal Debug Port extension* on page 2-36.

Bits[4:1]. DPv0

Reserved, SBZ.

DAPABORT, bit[0]

Write 1 to this bit to generate a DAP abort. This aborts the current AP transaction.

In DPv0, this bit is SBO.

Do this only if the debugger has received WAIT responses over an extended period. See *DAP Aborts*.

DAP Aborts

Writing 1 to the ABORT.DAPABORT register bit generates a DAP abort, causing the current AP transaction to abort. This also terminates the transaction counter, if it was active. It is IMPLEMENTATION DEFINED whether the AP propagates the abort, for example by aborting a transaction in progress.

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

——— Caution ———

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

After a DAP abort:

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

2.3.2 CTRL/STAT, Control/Status register

The CTRL/STAT register attributes are:

Purpose The Control/Status register provides control of the DP and status information about the DP.

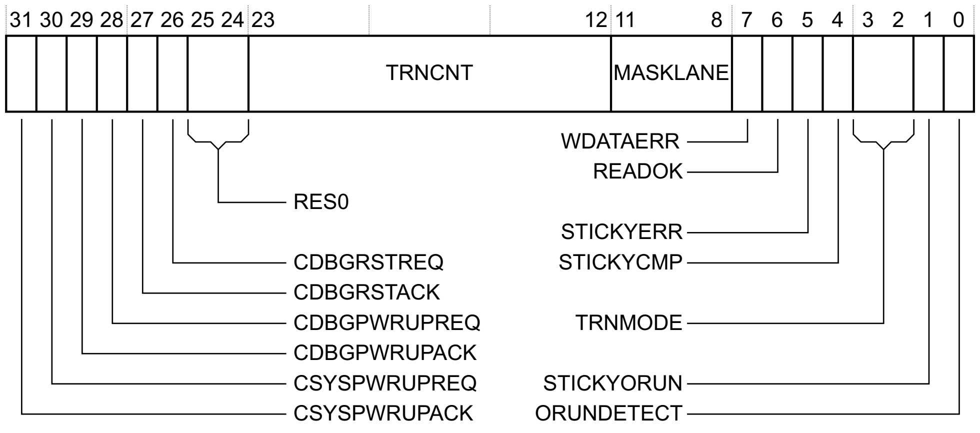
Configurations

A DP architecture register. The CTRL/STAT register is defined and implemented in DPv0, DPv1, and DPv2.

Attributes The CTRL/STAT register is:

- A read/write register, in which some fields are RO, meaning they ignore writes. See the field descriptions for more information.
- Accessed by a read or write to offset `0x4` of the DP register map, when SELECT.DPBANKSEL is `0x0`.

The CTRL/STAT register bit assignments are:



CSYSPWRUPACK, bit[31]

System powerup acknowledge. Indicates the status of the **CSYSPWRUPACK** signal. See *Power control requirements and operation* on page 2-64.

This bit is RO, meaning it ignores writes.

CSYSPWRUPREQ, bit[30]

System powerup request. This bit controls the **CSYSPWRUPREQ** signal. See *Power control requirements and operation* on page 2-64.

After a powerup reset, this bit is set 0.

CDBGPWRUPACK, bit[29]

Debug powerup acknowledge. Indicates the status of the **CDBGPWRUPACK** signal. See *Power control requirements and operation* on page 2-64.

This bit is RO, meaning it ignores writes.

CDBGPRWUPREQ, bit[28]

Debug powerup request. This bit controls the **CDBGPRWUPREQ** signal. See *Power control requirements and operation* on page 2-64.

After a powerup reset, this bit is set to 0.

CDBGRSTACK, bit[27]

Debug reset acknowledge. Indicates the status of the **CDBGRSTACK** signal. See *Debug reset control* on page 2-66.

This bit is RO, meaning it ignores writes.

CDBGRSTREQ, bit[26]

Debug reset request. This bit controls the **CDBGRSTREQ** signal. See *Debug reset control* on page 2-66.

It is IMPLEMENTATION DEFINED whether this bit is RW or RAZ/WI. See *Emulation of debug reset request* on page 2-67.

After a powerup reset, this bit is set to 0.

Bits[25:24] Reserved, RES0.

TRNCNT, bits[23:12]

Transaction counter.

See *The transaction counter* on page 2-38.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal Debug Port extension* on page 2-36.

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

MASKLANE, bits[11:8]

Indicates the bytes to be masked in *pushed-compare* and *pushed-verify* operations.

See *MASKLANE and the bit masking of the pushed-compare and pushed-verify operations* on page 2-49.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal Debug Port extension* on page 2-36.

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

WDATAERR, bit[7]

This bit is DATA LINK DEFINED:

- On a JTAG-DP this bit is reserved, RES0.
- On an SW-DP. this bit is RO/WI.

  This bit is set to 1 if a Write Data Error occurs. This happens if:

  — There is a a parity or framing error on the data phase of a write.

  — A write that has been accepted by the DP is then discarded without being submitted to the AP.

  For more information, see *Protocol errors, SW-DP* on page 2-36.

  On an SW-DP, this bit is cleared to 0 by writing 1 to the ABORT.WDERRCLR bit.

After a powerup reset, this bit is set to 0.

READOK, bit[6], DPv1 or higher

This bit is DATA LINK DEFINED

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

  The bit is set to 1 if the response to the previous AP read or RDBUFF read was OK. It is cleared to 0 if the response was not OK.

  This flag always indicates the response to the last AP read access. See *Protocol error response* on page 4-97.

After a powerup reset, this bit is set to 0.

STICKYERR, bit[5]

This bit is set to 1 if an error is returned by an AP transaction. See *Sticky flags and DP error responses* on page 2-34.

The behavior on writing is DATA LINK DEFINED:

- On a JTAG-DP, access is R/W1C.

- On a SW-DP, access is RO/WI.

The clearing of this bit to 0 is DATA LINK DEFINED, see *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50.

After a powerup reset, this bit is set to 0.

STICKYCMP, bit[4]

This bit is set to 1 when a match occurs on a *pushed-compare* or a *pushed-verify* operation. See *Pushed-compare and pushed-verify operations* on page 2-36.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal Debug Port extension* on page 2-36.

The behavior on writing is DATA LINK DEFINED:

- On a JTAG-DP access is R/W1C.
- On an SW-DP access is RO/WI.

The clearing of this bit to 0 is DATA LINK DEFINED, see *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50.

After a powerup reset, this bit is set to 0.

TRNMODE, bits[3:2]

This field sets the transfer mode for AP operations. See *TRNMODE, Transfer mode* on page 2-50.

It is IMPLEMENTATION DEFINED whether this field is implemented. See *MINDP, Minimal Debug Port extension* on page 2-36.

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

STICKYORUN, bit[1]

If overrun detection is enabled, this bit is set to 1 when an overrun occurs. See bit [0] of this register for details of enabling overrun detection.

The behavior on writing is DATA LINK DEFINED:

- On a JTAG-DP, access is R/W1C.
- On an SW-DP, access is RO/WI.

The clearing of this bit to 0 is DATA LINK DEFINED, see *Clearing the sticky error and compare flags in the CTRL/STAT register* on page 2-50.

After a powerup reset, this bit is set to 0.

ORUNDETECT, bit[0]

This bit is set to 1 to enable overrun detection. For more information see *Overrun detection* on page 2-34.

After a powerup reset, this bit is set to 0.

Note

- TRNCNT, MASKLANE, and TRNMODE are not supported in MINDP configuration. In MINDP configuration, the effect of writing a value other than zero to either TRNCNT or TRNMODE is UNPREDICTABLE.
- STICKYCMP is not supported in MINDP configuration. In MINDP configuration, the effect of writing a 1 to STICKYCMP is UNPREDICTABLE.

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 *TRNMODE, Transfer mode* on page 2-50.

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 2-36. 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 2-8 shows how the bits of MASKLANE control the comparison masking.

Table 2-8 Control of pushed operation comparisons by MASKLANE

MASKLANE[a]	Meaning	Mask used for comparisons[b]
`0b1xxx`	Include byte lane 3 in comparisons.	`0xFF------`
`0bx1xx`	Include byte lane 2 in comparisons.	`0x--FF----`
`0bxx1x`	Include byte lane 1 in comparisons.	`0x----FF--`
`0bxxx1`	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.

Note

A zero in any position excludes the corresponding byte lane from the comparison.

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

TRNMODE, Transfer mode

This field sets the transfer mode for AP operations. Table 2-9 lists the permitted values of this field, and their meanings.

Table 2-9 TRNMODE bit definitions

TRNMODE[a]	AP Transfer mode
`0b00`	Normal operation.
`0b01`	Pushed-verify operation.
`0b10`	Pushed-compare operation.
`0b11`	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 *Using the Access Port to access debug resources* on page 1-27.

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 2-36.

Clearing the sticky error and compare flags in the CTRL/STAT register

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

- STICKYERR, bit[5].
- STICKYCMP, bit[4].

- STICKYORUN, bit[1].
- WDATAERR, bit[7], SW-DP only.

The descriptions of these bits in CTRL/STAT 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. These bits can be cleared to 0 as follows:

JTAG-DP, all implementations

Write 1 to the appropriate bit of the CTRL/STAT 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.

You can use a single write of the CTRL/STAT register to clear multiple flags to 0 if necessary.

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

Write 1 to the appropriate CLR field of the ABORT register:

- To clear STICKYERR to 0, write 1 to the ABORT.STKERRCLR field.
- To clear STICKYCMP to 0, write 1 to the ABORT.STKCMPCLR field.
- To clear STICKYORUN to 0, write 1 to the ABORT.ORUNERRCLR field.
- To clear WDATAERR to 0, write 1 to the ABORT.WDERRCLR field, SW-DP only.

You can use a single write of the ABORT register to clear multiple flags to 0 if 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 STICKYERR or STICKYCMP flag, you must find which location was accessed to cause the flag to be set to 1.
- For the STICKYORUN flag, you must find which DP or AP transaction caused the overrun. You then have to repeat your transactions from that point.
- For the WDATAERR flag, you must resend the corrupted data.

2.3.3 DLCR, Data Link Control Register

The DLCR attributes are:

Purpose Controls the operating mode of the Data Link.

Configurations

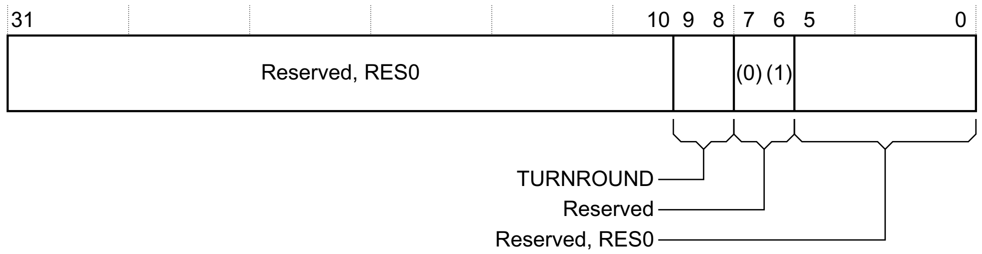
A DP architecture register. The DLCR is defined and implemented only in DPv1 and DPv2.

Attributes The DLCR is:

- DATA LINK DEFINED.
- A read/write register.
- Accessed by a read or write to offset `0x4` of the DP address map when SELECT.DPBANKSEL is set to `0x1`.

See the field descriptions for information about the register reset value.

For SW-DP, the DLCR bit assignments are:



Bits[31:10] Reserved, RES0.

TURNROUND, bits[9:8]

Turnaround tristate period. After a powerup or line reset, this field is set to `0b00`.

Note

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

Bit[7] Reserved, RES0.

Bit[6] Reserved, RES1.

Bits[5:0] Reserved, RES0.

TURNROUND, bits[9:8]

This field defines the turnaround tristate period, see *Line turn-round* on page 4-90. Table 2-10 shows the permitted values of this field, and their meanings.

Table 2-10 Turnaround tristate period field, TURNROUND, bit definitions

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

a. A *data period* is the period of a single data bit on the Serial Wire Debug interface.

2.3.4 DLPIDR, Data Link Protocol Identification Register

The DLPIDR attributes are:

Purpose The DLPIDR provides protocol version information.

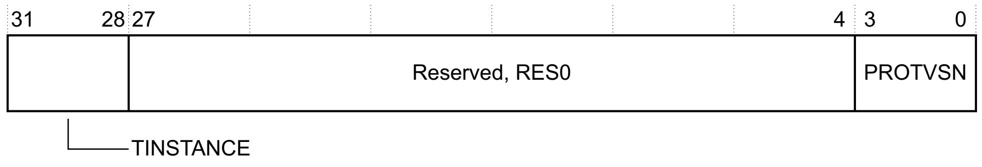
Configurations

A DP architecture register. The DLPIDR is defined and implemented only in DPv2.

Attributes The DLPIDR is:

- A read-only register.
- Accessed by a read to offset `0x4` of the DP register map when SELECT.DPBANKSEL is set to `0x3`.

The contents of the DLIPR are DATA LINK DEFINED. For SW-DP, the DLPIDR bit assignments are:



TINSTANCE, bits[31:28]

IMPLEMENTATION DEFINED. Defines an instance number for this device. This value must be unique for all devices with identical TARGETID[27:0] fields that are connected together in a multi-drop system.

Bits[27:4] Reserved, UNKNOWN.

PROTVSN, bits[3:0]

Defines the Serial Wire Debug protocol version that is implemented. Valid values for this field are:

`0x1`	Serial Wire Debug protocol version 2. Adds support for multidrop extensions. See Chapter 4 *The Serial Wire Debug Port (SW-DP)*.

All other values of this field are reserved.

Note

A Serial Wire Debug Port that implements DPv2, must implement at least Serial Wire Debug protocol version 2.

For JTAG-DP, the DLPIDR is reserved and accessing the register is UNPREDICTABLE.

2.3.5 DPIDR, Debug Port Identification Register

The DPIDR attributes are:

Purpose The DPIDR provides information about the Debug Port.

Configurations

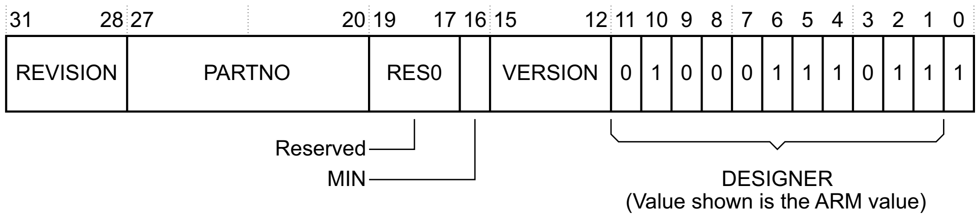
A DP architecture register. The DPIDR is defined and implemented only in DPv1 and DPv2.

Attributes The DPIDR is:

- A read-only register.
- Accessed by a read to offset `0x0` of the DP register map.

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

The DPIDR bit assignments are:



REVISION, bits[31:28]

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

PARTNO, bits[27:20]

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

Bits[19:17] Reserved, RES0.

MIN, bit[16] *Minimal Debug Port* (MINDP) functions implemented:

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

VERSION, bits[15:12]

Version of the Debug Port architecture implemented. Permitted values are:

`0x0`	Reserved. Implementations of DPv0 do not implement DPIDR.
`0x1`	DPv1 is implemented.
`0x2`	DPv2 is implemented.

All remaining values are reserved.

DESIGNER, bits[11:1]

Designer ID. An 11-bit JEDEC code formed from the JEDEC JEP106 continuation code and identity code. The ID identifies the designer of the Debug Port. The ARM value for this field is `0x23B`. Other designers must insert their own JEDEC-assigned code here.

Bit[0] RAO.

Note

- In DPv0, the DPIDR is reserved and accesses are UNPREDICTABLE.
- In all DP architecture versions, a JTAG-DP implementation must implement the IDCODE instruction and IDCODE scan-chain. The architecture does not require that the TAP IDCODE register value and the DPIDR value are the same.

2.3.6 EVENTSTAT, Event Status register

The EVENTSTAT register attributes are:

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

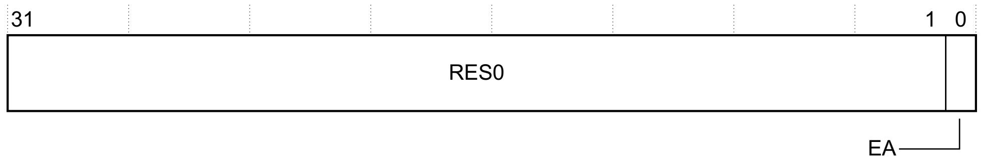
Configurations

A DP architecture register. The EVENTSTAT register is defined and implemented only in DPv2.

Attributes The EVENTSTAT register is:

- A read-only register.
- Accessed by a read to offset `0x4` of the DP register map when SELECT.DPBANKSEL is set to `0x4`.

The EVENTSTAT register bit assignments are:



Bits[31:1] Reserved, RES0.

EA, bit[0] Event status flag. Valid values for this bit are:

0	An event requires attention.
1	There is no event requiring attention.

If no event is implemented, this bit is RAZ.

ARM recommends that the event is either:

- Connected to an output trigger of a CoreSight Cross-Trigger Interface (CTI) with software acknowledge.
- For a uniprocessor system only, connected to an output from a processor that indicates whether the processor is halted:
  - For ARMv6-M and ARMv7-M processors, the recommended **HALTED** signal.
  - For all other ARM architecture processors, the recommended **DBGACK** signal.

Note

The status of the event is inverted in the register. This means that implementations that do not implement an event always indicate that an event requires attention. That is, that the debugger must poll registers elsewhere to detect the status of the system.

2.3.7 RDBUFF, Read Buffer register

The RDBUFF register attributes are:

Purpose

The RDBUFF register captures data from the AP, presented as the result of a previous read.

Configurations

A DP architecture register. The RDBUFF register is defined and implemented in DPv0, DPv1, and DPv2.

Attributes

The RDBUFF register is:

- A 32-bit read-only buffer.
- Accessed by a read of offset `0xC` in the DP register map.
- Has DATA LINK DEFINED behavior:

JTAG-DP	See *Read Buffer implementation and use on a JTAG-DP*.
SW-DP	See *Read Buffer implementation and use on a SW-DP*.

Read Buffer implementation and use on a JTAG-DP

On a JTAG-DP, the Read Buffer is always RAZ/WI.

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

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

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

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

2.3.8 RESEND, Read Resend register

The RESEND register attributes are:

Purpose

The RESEND register enables the read data to be recovered from a corrupted debugger transfer without repeating the original AP transfer.

Configurations

A DP architecture register. The RESEND register is defined and implemented only in DPv1 and DPv2.

Attributes

The RESEND register is:

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

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

Note

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

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-DP 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 returns the same value until a new access is made to an AP register or the DP RDBUFF register.

2.3.9 SELECT, AP Select register

The SELECT register attributes are:

Purpose

The SELECT register:
- Selects an *Access Port* (AP) and the active register banks within that AP.
- Selects the DP address bank.

Configurations

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

Attributes

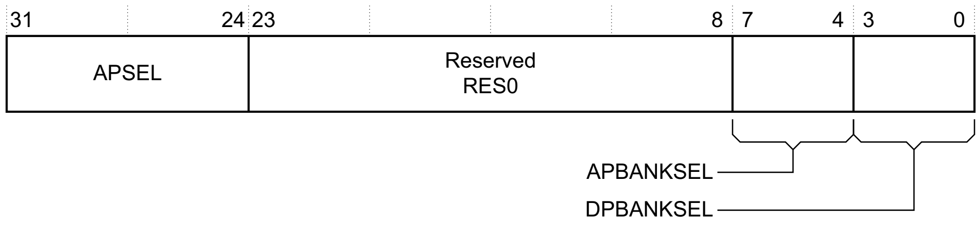
In DPv0, the SELECT register:
- Is a RW register.
- Is accessed by a read or write of the DP register at offset `0x8`.

Note

This specification deprecates reading the SELECT register in a DPv0 implementation.

In DPv1 and DPv2, the SELECT register:
- Is a WO register.
- Is accessed by a write to DP register `0x8`.

The SELECT register bit assignments are:



APSEL, bits[31:24]

Selects an AP. If APSEL is set to a non-existent AP then all AP transactions return zero on reads and are ignored on writes. See *Register maps, and accesses to reserved addresses* on page 2-44.

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

Note

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

Bits[23:8] Reserved, RES0.

APBANKSEL, bits[7:4]

Selects the active four-word register bank on the current AP. See *Using the Access Port to access debug resources* on page 1-27.

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

DPBANKSEL, bit[3:0]

Debug Port address bank select, see *DPBANKSEL* on page 2-59.

After a powerup reset, this field is set to 0.

DPBANKSEL

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

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

DPv1 In DPv1 the SELECT.DPBANKSEL field controls which SW-DP register is selected at address `0x4`, and Table 2-11 shows the permitted values of this field.

Table 2-11 DPBANKSEL SW-DP register allocation in DPv1

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

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

DPv2 In DPv2 the SELECT.DPBANKSEL field controls which SW-DP register is selected at address `0x4`, and Table 2-12 shows the permitted values of this field.

Table 2-12 DPBANKSEL SW-DP register allocation in DPv2

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

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

Note

Previous descriptions of ADIv5 have described DPBANKSEL as a single-bit field called CTRSEL, defined only for SW-DP. From issue B of this document, DPBANKSEL is redefined. The new definition is backwards-compatible.

2.3.10 TARGETID, Target Identification register

The TARGETID register attributes are:

Purpose

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

Configurations

A DP architecture register. The TARGETID register is defined and implemented only in DPv2.

Attributes

The TARGETID register is:

- A read-only register.
- Accessed by a read to offset `0x4` of the DP register map when SELECT.DPBANKSEL is set to `0x2`.

The TARGETID register bit assignments are:

31 28	27 12	11 1	0
TREVISION	TPARTNO	TDESIGNER	1

TREVISION, bits[31:28]

Target revision.

TPARTNO, bits[27:12]

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

TDESIGNER, bits[11:1]

IMPLEMENTATION DEFINED. An 11-bit code formed from the JEDEC JEP106 continuation code and identity code. The ID identifies the designer of the part. Designers must insert their JEDEC-assigned code here.

Note

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

Bit[0] RAO.

2.3.11 TARGETSEL, Target Selection register

The TARGETSEL register attributes are:

Purpose

The TARGETSEL register selects the target device in a Serial Wire Debug multi-drop system. See *Target selection*.

Configurations

A DP architecture register. The TARGETSEL register is defined and implemented only in DPv2.

Attributes

The TARGETSEL register is:

- A write-only register.
- DATA LINK DEFINED:

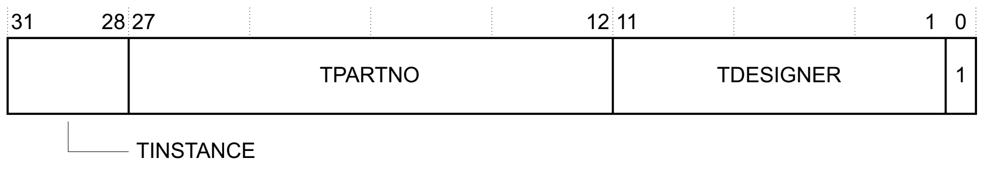
  JTAG-DP The register is reserved, any access is UNPREDICTABLE.

  SW-DP The register is implemented if SWD protocol version 2 is implemented.
- Accessed by a write to offset `0xC` of the DP register map.

Note

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

The TARGETSEL register bit assignments are:



TINSTANCE, bits[31:28]

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

TPARTNO, bits[27:12]

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

TDESIGNER, bits[11:1]

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

Bit[0] SBO.

Target selection

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

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

Writing any other value deselects the target. Debug tools must write `0xFFFFFFFF` to deselect all targets. This is an invalid TARGETID value. All other invalid TARGETID values are reserved.

During the response phase of a write to the TARGETSEL register, the target does not drive the line. See *Protocol errors, SW-DP* on page 2-36 for more information.

2.4 System and debug power and debug reset control

This section gives detailed information about system and debug power, and debug reset control.

2.4.1 The DAP power domains model

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

Three power domains are modeled:

Always-on power domain

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

System power domain

This includes system components.

Debug power domain

This includes all of the debug subsystem.

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

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

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

The DP registers reside in the always-on power domain, on the external interface side of the DP. Therefore, they can always be driven, enabling powerup requests to be made to a system power controller. The power and reset control bits are part of the DP CTRL/STAT register. See *Debug reset control* on page 2-66 for more information about the reset control bits in this register.

ADIv5 defines two pairs of power control signals:

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

Table 2-13 summarizes the programmers' model for the power control signal pairs.

Table 2-13 Debug Port programmers' model

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

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

CDBGPWRUPREQ and CDBGPWRUPACK

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

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

The power/clock controller must honor **CDBGPWRUPREQ** for as long as it is asserted. For example, if a component in a debug power domain requests to have its clocks stopped, the request must be emulated for non-debug logic within that power domain. This includes all components with a single shared domain.

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

- There is some means to save and restore the state of the debug resources.
- The remainder of the component does not require to be powered for the debugger to be able to communicate with the debug resources.

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

CDBGPWRUPACK is the acknowledge signal for the **CDBGPWRUPREQ** request signal. **CDBGPWRUPACK** must be asserted for as long as **CDBGPWRUPREQ** is asserted. See *Powerup request and acknowledgement timing* on page 2-65.

CSYSPWRUPREQ and CSYSPWRUPACK

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

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

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

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

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

CSYSPWRUPACK is the acknowledge signal for the **CSYSPWRUPREQ** request signal. **CSYSPWRUPACK** must be asserted for as long as **CSYSPWRUPREQ** is asserted. See *Powerup request and acknowledgement timing* on page 2-65.

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

2.4.2 Power control requirements and operation

Except where the context clearly indicates otherwise, the following description applies to both:

- System domain powerup, using the **CSYSPWRUPREQ** and **CSYSPWRUPACK** signals.
- Debug domain powerup, 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 powerup requests and acknowledgements are:

- The debugger must not set CTRL/STAT.{CSYSPWRUPREQ, CDBGPWRUPREQ} to {1, 0}. This combination of requests is UNPREDICTABLE.
- To initiate powerup, **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 power domain.
- Tools can only initiate an AP transfer when **CDBGPWRUPREQ** and **CDBGPWRUPACK** are asserted HIGH. If **CDBGPWRUPREQ** or **CDBGPWRUPACK** is LOW, any AP transfer generates an immediate fault response.
- 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.
- **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 powerdown has been requested, by the deassertion of **CxxxPWRUPREQ**, tools must wait until **CxxxPWRUPACK** is LOW before making a new request for powerup. **CxxxPWRUPACK** going LOW indicates that the power controller has recognized the original powerdown request.

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

Figure 2-2 on page 2-65 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 2-2 on page 2-65.

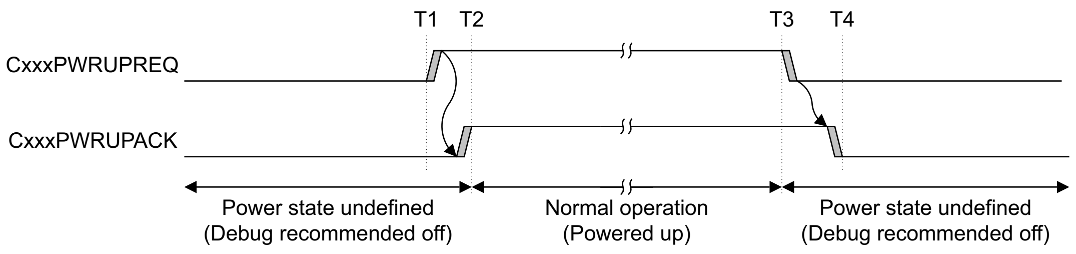

T1
T2
T3
T4
CxxxPWRUPREQ
CxxxPWRUPACK
Power state undefined (Debug recommended off)
Normal operation (Powered up)
Power state undefined (Debug recommended off)

Figure 2-2 Powerup request and acknowledgement timing

2.4.3 Emulation of powerdown

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

- A powerdown 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.

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

Emulation of powerdown 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.

2.4.4 Emulation of power control

Where the system to which a DAP is connected does not support the ADIv5 power control model, the required signals must be emulated or generated from other signals. The following subsections describe possible cases:

- *System without power controller support for the ADIv5 control scheme*.
- *System power controller does not support separate power domains* on page 2-66.

System without power controller support for the ADIv5 control scheme

If the connected system can not support the ADIv5 powerup and powerdown control scheme, then **CxxxPRWUPACK** must be connected to **CxxxPRWUPREQ**. With these connections, whenever the DAP requests powerup, or removes a powerup request, it receives the appropriate acknowledge immediately. This power control emulation is shown in Figure 2-3.

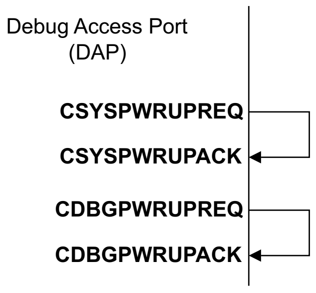


Figure 2-3 Emulation of powerup control

System power controller does not support separate power domains

If the debug power domain resides within the system power domain, **CSYSPWRUPREQ** and **CDBGPWRUPREQ** can independently request powerup of the debug functions in the debug and system power domains. Whenever **CSYSPWRUPREQ** is set to 1, **CDBGPWRUPREQ** must also be set to 1. Setting bits[30,28] in the CTRL/STAT register to `0b10` gives UNPREDICTABLE system behavior.

The **CxxxPRWUPACK** signals must be generated, so that the DAP sees the correct response to asserting a **CxxxPRWUPREQ** signal. This is shown in Figure 2-3 on page 2-65.

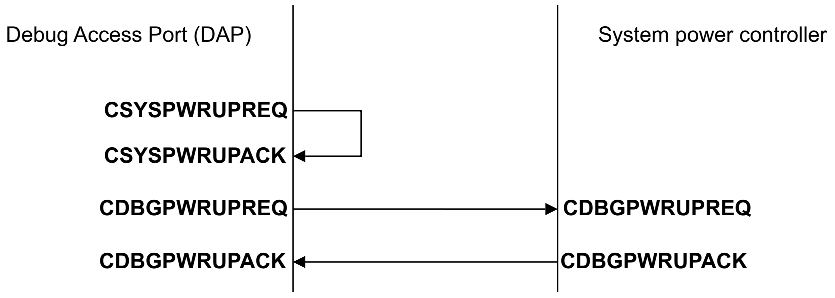


Figure 2-4 Signal generation for a single system and debug power domain

2.4.5 Debug reset control

The DP CTRL/STAT register provides two bits, bits[27:26], for reset control of the debug domain. 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, **CDBGRSTREQ**, and a reset acknowledge, **CDBGRSTACK**. 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 2-5 shows the reset request and acknowledge timing.

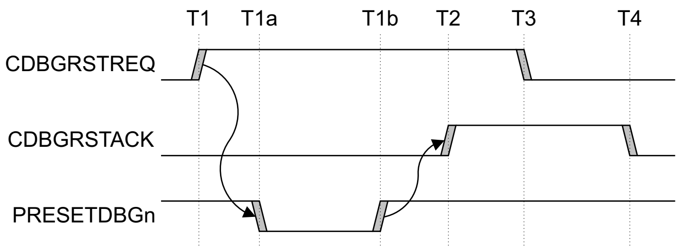


Figure 2-5 Reset request and acknowledge timing

Note

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

In Figure 2-5:

1. At time T1, the debugger writes 1 to CTRL/STAT.CDBGRSTREQ. 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 powerup reset.

2. At time T2, the system reset controller acknowledges that the reset of the debug domain has completed. The **CDBGRSTACK** signal sets the CTRL/STAT.CDBGRSTACK bit to 1.
3. At time T3, the debugger checks the DP CTRL/STAT register and finds that the reset has completed. Therefore, it writes 0 to CTRL/STAT.CDBGRSTREQ. This removes the reset request signal.
4. At time T4, the system reset controller recognizes that **CDBGRSTREQ** is no longer asserted, and deasserts **CDBGRSTACK**.

Caution

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

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

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

Emulation of debug reset request

If the debug reset control is not supported then:

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

Note

ARM recommends tying **CDBGRSTACK** LOW so that after a timeout the debugger can deduce that debug reset was not implemented.

Limitations of CDBGRSTREQ and CDBGRSTACK

Debug reset control on page 2-66 shows how these bits can drive the debug reset signal, **PRESETDBGn**. In an actual system, there might be other reset signals associated with other debug buses. For example, in an ARM CoreSight system, **ATRESETn** resets all registers in the AMBA Trace Bus domain.

Note

It is IMPLEMENTATION DEFINED which components are reset by **CDBGRSTREQ**. Figure 2-5 on page 2-66 is an example only. It is not the case that the only components reset are those which use **PRESETDBGn**.

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

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

ARM recommends that **CDBGRSTREQ** from an AP is ignored by the reset controller if the Device Enable signal to the AP, **DEVICEEN**, is LOW.

Caution

System-level use of debug components must be handled with caution. ARM recommends that such system-level usage is not combined with a reset system that permits those debug components to be reset without the knowledge of the system. ARM recommends that debuggers do not use debug reset request other than when absolutely necessary.

2.4.6 System reset control

The Debug Port does not provide any control bits for requesting a system reset. However, it is common for the physical interface to the debugger to include an system reset pin, **nSRST**. This section describes the recommended behavior for the **nSRST** pin.

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

During the time that **nSRST** is asserted:

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

Limitations of system reset control

The debugger must ensure that there are no system accesses in progress through the DAP before asserting **nSRST**. When **nSRST** is asserted, the debugger can access the debug domain.

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

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

ARM recommends that debuggers set CTRL/STAT.CDBGPWRUPREQ to zero during the time that **nSRST** is initially asserted.

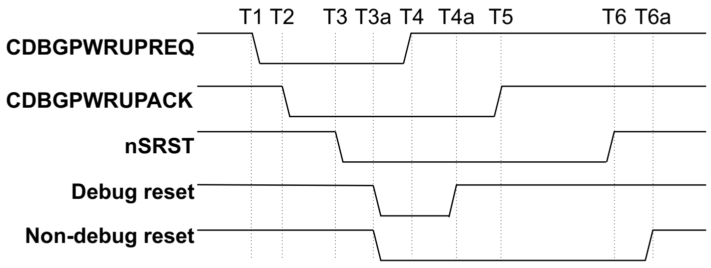


Figure 2-6 Example system reset timing

1. At time T1, the debugger writes 0 to CTRL/STAT.CDBGPWRUPREQ.
2. At time T2, the system power controller acknowledges this and CTRL/STAT.CDBGPWRUPACK is cleared to zero.
3. At time T3, the debugger asserts **nSRST**. The debug domain and non-debug domain are reset at time T3a. The debug domain reset is complete by time T4a.
4. At time T4, the debugger writes 1 to CTRL/STAT.CDBGPWRUPREQ.
5. At time T5, the system power controller acknowledges this request and signals the debug domain reset is complete. CTRL/STAT.CDBGPWRUPACK is set to 1. The debugger can now program the debug domain.
6. At time T6, the debugger releases **nSRST**. The non-debug domain reset completes at time T6a.

Chapter 3
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 DAP, and all interface accesses are made using the scan chains, driven by the DBGTAPSM.

This chapter contains the following sections:

- *The Debug TAP State Machine introduction* on page 3-70.
- *The scan chain interface* on page 3-71.
- *IR scan chain and IR instructions* on page 3-74.
- *DR scan chain and DR registers* on page 3-78.

3.1 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 DAP through the JTAG-DP. It is based closely on the JTAG TAP State Machine, see *IEEE 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture*. This chapter describes both the DBGTAPSM and its scan chain interface.

Figure 1-2 on page 1-26 shows a DAP with a JTAG-DP, including the basic operation of the scan chain interface.

3.2 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, 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 DAP.

Figure 1-2 on page 1-26 shows how the scan chains provide access to the different levels of the DAP architecture, and this is summarized in Figure 3-1.

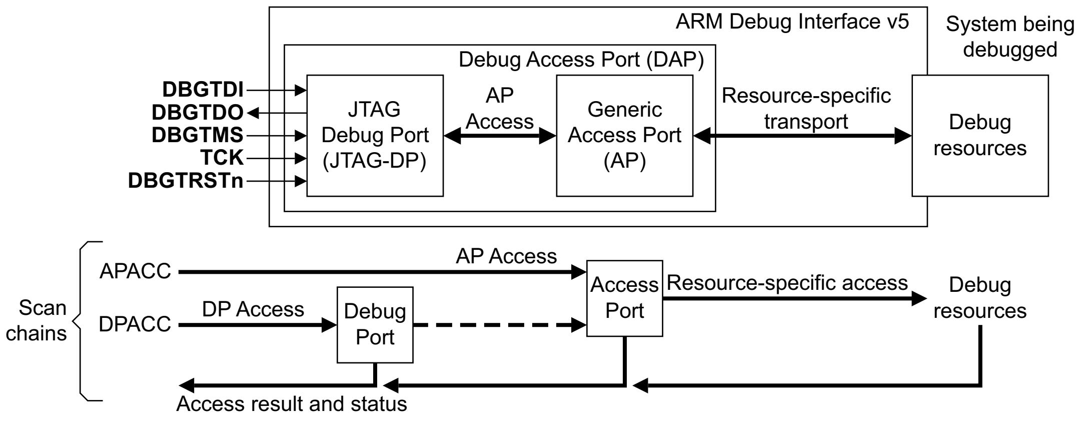


Figure 3-1 Mapping of the JTAG-DP scan chains onto the logical levels of the ARM Debug Interface

3.2.1 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 3-1. This table also shows the names of the equivalent signals in a JTAG implementation.

Table 3-1 JTAG-DP signal connections

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

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

Figure 3-2 shows the recommended physical connection to the JTAG-DP:

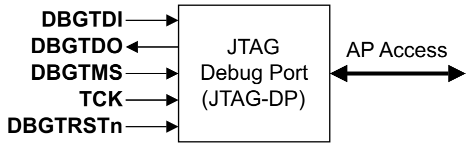
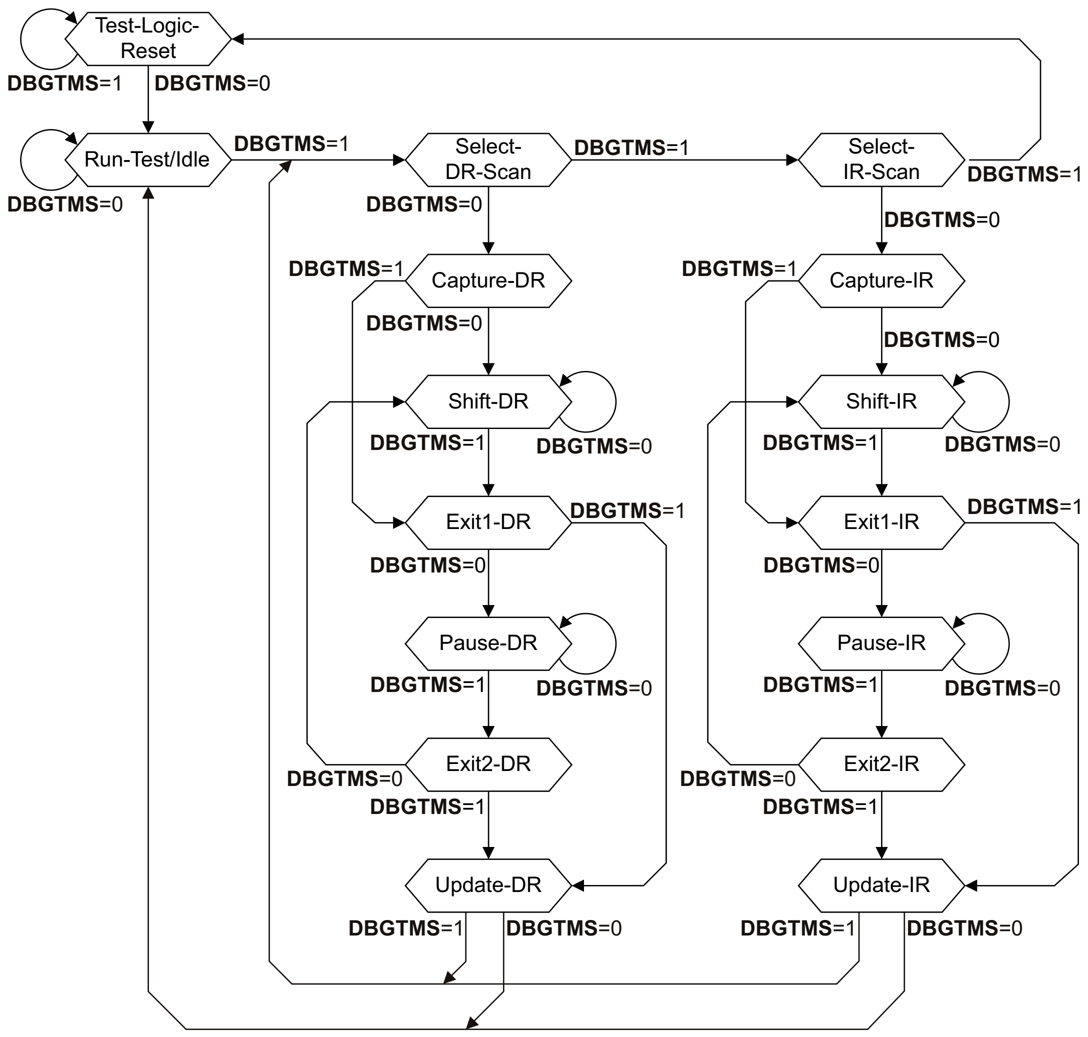


Figure 3-2 JTAG-DP physical connection

3.2.2 The Debug TAP State Machine (DBGTAPSM)

Figure 3-3 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 3-3 The Debug TAP State Machine (DBGTAPSM)

3.2.3 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 3-3 on page 3-72:

- When the DBGTAPSM goes through the Capture-IR state, `0b0001` 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, the IR scan chain advances one bit for each rising edge of **TCK**. This means that on the first tick:
  - — The LSB of the IR scan chain is output on **DBGTDO**.
  - — Bit[1] of the IR scan chain is transferred to bit[0].
  - — Bit[2] of the IR scan chain is transferred to bit[1].
  - — Similarly, for every other bit *n* of the IR scan chain, bit[*n*] of the scan chain is transferred to bit[*n*-1].
  - — The value on **DBGTDI** is transferred to the MSB of the IR scan chain.
- When the DBGTAPSM goes through the Update-IR state, the value scanned into the IR 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 DR scan chain is transferred into the selected Data Register.
- When the DBGTAPSM is in the Run-Test/Idle state, no special actions occur. Debuggers can use this 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 3-74 and *DR scan chain and DR registers* on page 3-78.

The **DBGTRSTn** signal only resets the DBGTAP state machine and Instruction Register. **DBGTRSTn** asynchronously takes the DBGTAPSM to the Test-Logic-Reset state. As shown in Figure 3-3 on page 3-72, 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 powerup reset.
- The AP registers are reset on a powerup reset, and also by the Debug Reset Control described in *Debug reset control* on page 2-66.

3.3 IR scan chain and IR instructions

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

3.3.1 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 `0b0001` 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.

In the Test-Logic-Reset state, IDCODE becomes the current instruction.

Order Figure 3-4 shows the operation of the Instruction Register.

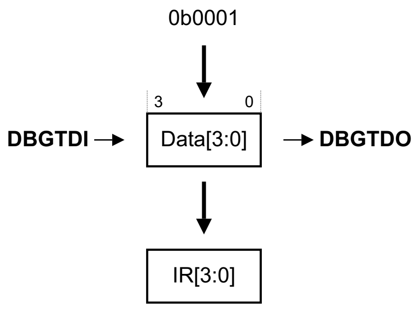


Figure 3-4 JTAG-DP Instruction Register operation

This register is mandatory in the IEEE 1149.1 standard.

3.3.2 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 3-78. The standard IR instructions are listed in Table 3-2, and recommended IMPLEMENTATION DEFINED extensions to this instruction set are described in *IMPLEMENTATION DEFINED extensions to the IR instruction set* on page 3-76.

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

Table 3-2 Standard IR instructions

IR instruction value	Data register	DR scan length	Notes
`0b0xxx`	-	-	*IMPLEMENTATION DEFINED extensions to the IR instruction set* on page 3-76
`0b1000`	ABORT	35	-
`0b1001`	-	-	Reserved
`0b1010`	DPACC	35	See *DPACC and APACC, the JTAG-DP DP Access register and AP Access register* on page 3-81
`0b1011`	APACC	35	
`0b110x`	-	-	Reserved
`0b1110`	IDCODE	32	-
`0b1111`	BYPASS	1	-

3.3.3 IMPLEMENTATION DEFINED extensions to the IR instruction set

The eight IR instructions `0b0000` to `0b0111` 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 3-3. All these instructions select the boundary scan data register.

Note

This extension describes only the boundary scan instructions described by IEEE 1149.1-1990 and IEEE 1149.1-2001. Other editions of IEEE 1149.1 might define additional instructions. The extension only supports up to eight additional instructions.

ARM recommends that:

- Separate JTAG TAPs are used for boundary scan and debug.
- The instructions listed in Table 3-3 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 3-3 Recommended IMPLEMENTATION DEFINED IR instructions for IEEE 1149.1-compliance

IR instruction value	Instruction	Required by IEEE 1149.1?
`0b0000`	EXTEST	See note in main text.
`0b0001`	SAMPLE	Yes
`0b0010`	PRELOAD	Yes
`0b0011`	Reserved	-
`0b0100`	INTEST[a]	No
`0b0101`	CLAMP[a]	No
`0b0110`	HIGHZ[a]	No
`0b0111`	CLAMPZ[a]	No. See *The CLAMPZ instruction*.

a. Reserved, if the instruction is not implemented.

If you require a boundary scan implementation, you must implement the instructions that are required by IEEE 1149.1. The other IR instruction values listed in Table 3-3 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, requires that instruction `0b0000` is EXTEST. However, in more recent editions this requirement is removed and the specification recommends that instruction `0b0000` is reserved. See the IEEE specification for more details.

The IEEE 1149.1 specification also defines the IDCODE and BYPASS instructions. These are included in Table 3-2 on page 3-75.

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.

3.4 DR scan chain and DR registers

The physical DR registers are:

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

There is a scan chain associated with each of these registers. As described in *IR scan chain and IR instructions* on page 3-74, the value in the IR register determines which of these scan chains is connected to the **DBGTDI** and **DBGTDO** signals.

3.4.1 BYPASS, the JTAG-DP Bypass register

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 3-5 shows the operation of the Bypass register.

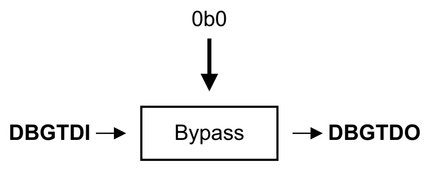


Figure 3-5 JTAG-DP Bypass register operation

This register is mandatory in the IEEE 1149.1 standard.

3.4.2 IDCODE, the JTAG TAP ID register

Purpose JTAG-DP TAP identification. The IDCODE value enables a debugger to identify the Debug Port to which it is connected. JTAG-DP implementations have different IDCODE values, so that a debugger can distinguish between them.

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 3-6 shows the operation of the JTAG TAP ID Register.

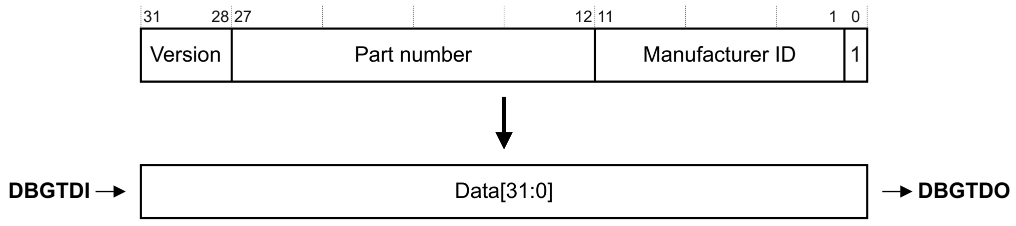
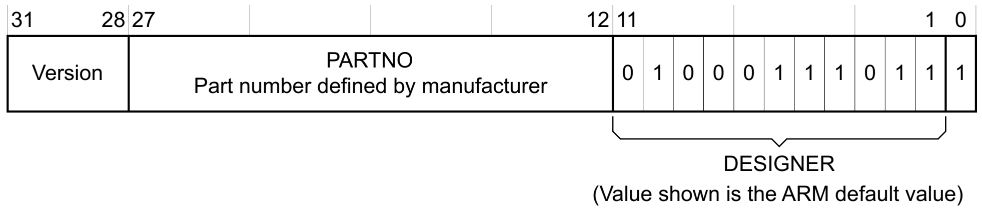


Figure 3-6 JTAG TAP ID Register operation

In more detail, the IDCODE bit assignments are:


31
28 27
12 11
1 0
Version
PARTNO
Part number defined by manufacturer
0 1 0 0 0 1 1 1 0 1 1
1
DESIGNER
(Value shown is the ARM default value)

VERSION, bits[31:28]

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

PARTNO. bits[27:12]

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

DESIGNER, bits[11:1]

Designer ID. An 11-bit JEDEC code formed from the JEDEC JEP106 continuation code and identity code. The ID identifies the designer of the JTAG-DP TAP. See *The JEDEC Designer ID* on page 3-80.

The ARM default value for this field is `0x23B`. Other designers must insert their own JEDEC assigned code here.

Bit[0] RAO.

The JEDEC Designer ID

This field is bits [11:1] of the Identification Code Register. The JEDEC Designer ID is also described as the JEP106 manufacturer identification code, and can be subdivided into two fields, as shown in Table 3-4.

Table 3-4 JEDEC JEP106 manufacturer ID code, with ARM values

JEP106 field	Bits[a]	ARM registered value
Continuation code	4-bits, [11:8]	`0b0100` (`0x4`)
Identity code	7-bits, [7:1]	`0b0111011` (`0x3B`)

a. Field width in bits, and the corresponding bits in the IDCODE register.

JEDEC codes are assigned by the JEDEC Solid State Technology Association, see *JEP106, Standard Manufacturer's Identification Code*.

Normally, this field identifies the designer of the ADIv5 implementation, rather than the system architect or the device manufacturer. 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.

3.4.3 DPACC and APACC, the JTAG-DP DP Access register and AP Access register

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 accesses the DP registers, other than ABORT. For addressing information, see:

- *DP architecture version 0 (DPv0) address map* on page 2-41.
- *DP architecture version 1 (DPv1) address map* on page 2-42.
- *DP architecture version 2 (DPv2) address map* on page 2-43.

The APACC scan chain accesses registers in the currently-selected AP register bank, see:

- *MEM-AP register summary* on page 7-146 for details of accessing MEM-AP registers.
- *JTAG-AP register summary* on page 8-176 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 3-5.

Table 3-5 DPACC and APACC ACK responses

Response	ACK[2:0] encoding	See:
OK/FAULT	`0b010`	*The OK/FAULT response to a DPACC or APACC access* on page 3-82
WAIT	`0b001`	*The WAIT response to a DPACC or APACC access* on page 3-83

All other ACK encodings are reserved.

- In the Shift-DR state, this data is shifted out, least significant bit first. As shown in Figure 3-7, 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 3-82.
- 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 3-82.
  - — *Update-DR operation following a WAIT response* on page 3-83.

Order Figure 3-7 shows the operation of the DP and AP Access Registers.

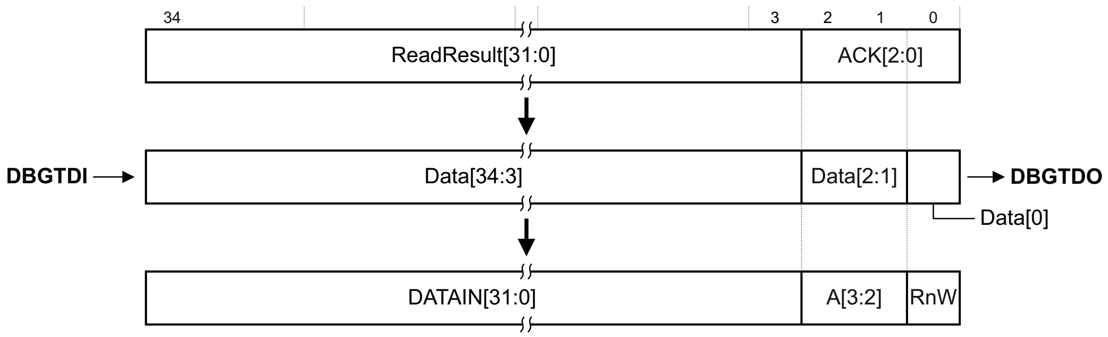


Figure 3-7 Operation of JTAG-DP DP Access register and AP Access register

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:

- If the previous instruction in the IR was not one of DPACC, APACC or BYPASS, then the captured ReadResult[31:0] is UNKNOWN, and if Data[34:3] is shifted out it must be discarded.
- If the previous transaction was a read that completed successfully, 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 UNKNOWN, 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]. For addressing information, see:
  - *DP architecture version 0 (DPv0) address map* on page 2-41.
  - *DP architecture version 1 (DPv1) address map* on page 2-42.
  - *DP architecture version 2 (DPv2) address map* on page 2-43.
- 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.

  For addressing information, see:
  - *MEM-AP register summary* on page 7-146, for accesses to MEM-AP registers.
  - *JTAG-AP register summary* on page 8-176, for accesses to JTAG-AP registers.

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. The section *Target response summary* on page 3-84 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 2-34.

- 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 CTRL/STAT.STICKYCMP flag is updated based on this comparison. For more information see *Pushed-compare and pushed-verify operations* on page 2-36.

  Pushed operations are enabled using the CTRL/STAT.TRNMODE field.

- The AP access does not complete until the AP signals it as completed. For example, if you access a 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 must 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 SoC 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 UNKNOWN.

Sticky overrun behavior on DPACC and APACC accesses

At the Capture-DR state, if the previous transaction has not completed a WAIT response is generated. When this happens, if the Overrun Detect flag, CTRL/STAT.ORUNDETECT is set to 1, the Sticky Overrun flag, CTRL/STAT.STICKYORUN, is set to 1.

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 2-34 for more information.

Minimum response times

As explained in *The OK/FAULT response to a DPACC or APACC access* on page 3-82, 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 DBGTAPSM 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 3-3 on page 3-72, 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 recommends that debuggers must 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 3-82, 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 3-6, for cases where the previous scan was a DPACC access.
- Table 3-7 on page 3-85, for cases where the previous scan was an APACC access.

Note

The target responses shown in Table 3-6 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 3-6 JTAG-DP target response summary, when previous scan[a] was a DPACC access

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

a. The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates made following a WAIT response are discarded.

b. A[3:2] in the DPACC access.

c. The *Sticky?* column indicates whether any Sticky flag was set to 1 in the DP CTRL/STAT register.

d. The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.

e. If the Overrun Detect flag is set to 1 then this access and response sequence causes the CTRL/STAT.Sticky Overrun bit to be set to 1.

Table 3-7 JTAG-DP target response summary, when previous scan[a] was an APACC access

Previous scan[a], at Update-DR state			Current scan, at Capture-DR state			Notes
Access	**A[3:2][b]**	**Sticky?[c]**	**AP state[d]**	**Read result**	**ACK**	
X	`0bXX`	X	Busy	UNKNOWN	WAIT	Might cause Sticky Overrun flag to be set to 1.[e]
R	`0bXX`	No	Ready	See Notes	OK/ FAULT	See footnote[f].
			Error	UNKNOWN		Sticky Error flag is set to 1.
W	`0bXX`	No	Ready	UNKNOWN	OK/ FAULT	See footnote[g].
			Error	UNKNOWN		Sticky Error flag is set to 1.
X	`0bXX`	Yes	X	UNKNOWN	OK/ FAULT	Previous transaction was discarded.

a. The previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates made following a WAIT response are discarded.

b. A[3:2] in the APACC access.

c. The *Sticky?* column indicates whether any Sticky flag was set to 1 in the DP CTRL/STAT register.

d. The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.

e. 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 *CTRL/STAT, Control/Status register* on page 2-47.

f. If pushed-verify or pushed-compare is implemented and enabled, the behavior is UNPREDICTABLE. Otherwise, returns the value of the AP register addressed on the previous scan.

g. If pushed-verify or pushed-compare is implemented and 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 2-36. 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 3-6 on page 3-84 and Table 3-7, shows the responses the host might receive after initiating a DPACC or APACC access. Table 3-8 indicates the normal action of a host in response to each of these ACKs.

Table 3-8 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.

3.4.4 ABORT, the JTAG-DP Abort register

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.

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 accesses the AP Abort register.

In DPv0, the effect of writing a value other than `0x00000001` to the ABORT scan chain is UNPREDICTABLE. For more information, see the AP ABORT register. This means that, in DPv0, the debugger must scan the value `0x000000008` into this scan chain.

Order Figure 3-8 shows the operation of the ABORT scan chain.

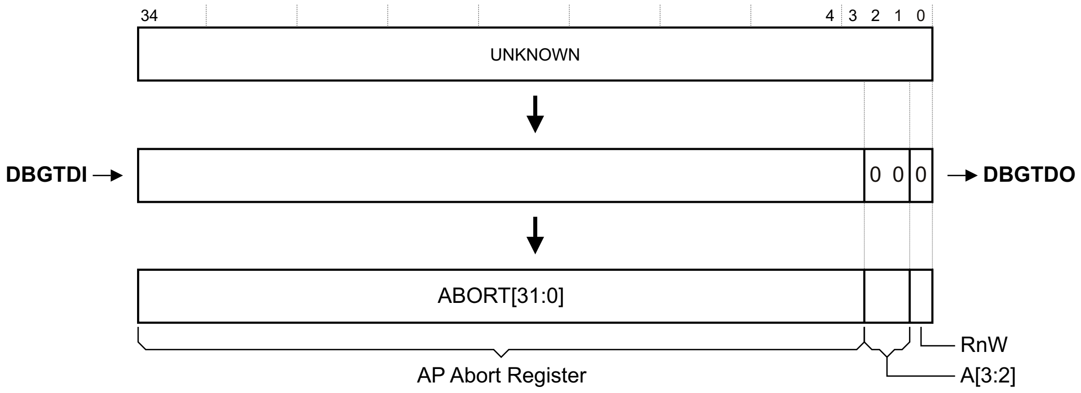


Figure 3-8 JTAG-DP ABORT scan chain operation

Chapter 4
The Serial Wire Debug Port (SW-DP)

This chapter describes the implementation of the *Serial Wire Debug Port* (SW-DP), including the 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 summarized in this chapter.

This chapter contains the following sections:

- *Introduction to the Serial Wire Debug Port* on page 4-88.
- *Introduction to the ARM Serial Wire Debug (SWD) protocol* on page 4-89.
- *Serial Wire Debug protocol operation* on page 4-93.
- *Serial Wire Debug interface* on page 4-104.

4.1 Introduction to the Serial Wire Debug Port

This chapter 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).

The ARM Serial Wire Debug interface is a synchronous serial interface. This chapter does not describe the physical characteristics of the SWD interface, such as signal timings.

Note

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

4.2 Introduction to the ARM Serial Wire Debug (SWD) protocol

The ARM Serial Wire Debug interface uses a single bidirectional data connection and a separate clock to transfer data synchronously.

An operation on the wire consists of two or three phases:

Packet request

The external *host* debugger issues a request to the DP. The DP is the *target* of the request.

Acknowledge response

The target sends an acknowledge response to the host.

Data transfer phase

This phase is only present when either:

- A data read or data write request is followed by a valid (OK) acknowledge response.
- The CTRL/STAT.ORUNDETECT flag is set to 1.

The data transfer is one of:

- Target to host, following a read request (RDATA).
- Host to target, following a write request (WDATA).

Note

If the CTRL/STAT.ORUNDETECT bit is 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 4-98.

When the SW-DP receives a packet request from the debug host, it must respond immediately with the acknowledge phase. There is a turnaround period between these phases, as they are in different directions. If a data phase is required, this follows immediately after the acknowledge phase.

For a write request, there is a turnaround period between the acknowledge phase and the WDATA data transfer phase. Following the WDATA data transfer phase the host continues to drive the wire. There is no additional turnaround period.

For a read request, there is no turnaround period between the acknowledge phase and the data transfer phase. There is a turnaround period following the RDATA data transfer phase, following which the host drives the wire.

To ensure that the transfer can be clocked through the SW-DP, after the data transfer phase the host must do one of the following:

- Immediately start a new SWD operation with the start bit of a new packet request.
- Continue to drive the SWD interface with idle cycles until the host starts a new SWD operation.
- If the host is driving the Serial Wire Debug clock, continue to clock the SWD interface with at least eight idle cycles. After this it can stop the clock.

4.2.1 SWD protocol versions

Serial Wire Debug protocol version 1 is a point-to-point architecture, supporting connection between a single host and a single device. It permits connection to multiple devices by providing additional connections from the host. This has a number of disadvantages:

- It complicates the physical connection standard, by having variants with different numbers of connections.
- It increases the number of pins required for the connector on the device PCB. This is unacceptable where size is a limiting factor.
- It increases the number of pins required on a package with multiple dies inside.
- It makes it difficult to integrate multiple platforms accessed by the Serial Wire Debug protocol into the same chip.

Techniques to solve this require connections that are shared between multiple Serial Wire devices. This is detrimental to the maximum speed of operation, but in many situations this is an acceptable trade-off.

Serial Wire Debug protocol version 2 is a multi-drop architecture that:

- Enables a two-wire host connection to communicate simultaneously with multiple devices.
- Permits an effectively unlimited number of devices to be connected simultaneously, subject to electrical constraints.
- Is largely backwards-compatible, because provision for multi-drop support in a device does not break point-to-point compatibility with existing host equipment that does not support the multi-drop extensions. For more information, see *Limits on backwards compatibility of SWD protocol version 2*.
- Permits a device to power down completely, during the time that the device is not selected.
- Prevents multiple devices from driving the wire simultaneously, and continues to support the wire being actively driven both HIGH and LOW, maintaining a high maximum clock speed.
- Permits multi-drop connections incorporating devices that do not implement the Serial Wire Debug protocol.

Limits on backwards compatibility of SWD protocol version 2

SWD protocol version 2 requires implementation of dormant mode, which can limit its compatibility with SWD version 1:

- For an SWJ-DP implementation, JTAG is selected on a powerup reset. Selecting SWD bypasses dormant state, and therefore operation is compatible with SWD protocol version 1.
- For an SW-DP implementation of SWD protocol version 2, dormant state is selected on a powerup reset, meaning the start-up state differs from that with SWD protocol version 1. If SWD operation is then selected, operation becomes compatible with SWD protocol version 1.

4.2.2 Line turn-round

To prevent contention, a turnaround period is required when the device driving the wire changes. For the turnaround period, neither the host nor the target drives the wire, and the state of the wire is undefined. See also *Line pull-up on page 4-104*.

Note

The line turn-round period can provide for pad delays when using a high sample clock frequency.

The length of the turnaround period is controlled by DLCR.TURNROUND. The default setting is a turnaround period of one clock cycle.

4.2.3 Idle cycles

After completing a transaction, the host must either insert idle cycles or continue immediately with the start bit of a new transaction.

The host clocks the Serial Wire Debug interface with the line LOW to insert idle cycles.

4.2.4 Bit order

All data values in SWD operations are transferred LSB first.

For example, the OK response of `0b001` appears on the wire as 1, followed by 0, followed by 0, as shown in Figure 4-1 on page 4-94 and Figure 4-2 on page 4-95.

4.2.5 Parity

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 4-1 on page 4-94 to Figure 4-7 on page 4-99. It appears on the wire immediately after the A[2:3] bits. A parity error in the packet request is detected by the SW-DP, which responds with a protocol error. See *Protocol error response* on page 4-97.

The WDATA parity bit is shown in Figure 4-1 on page 4-94 and in Figure 4-7 on page 4-99. It appears on the wire immediately after the WDATA[31] bit. A parity error in the WDATA data transfer phase is detected by the SW-DP and, other than writes to TARGETSEL, recorded in CTRL/STAT.WDATAERR. If overrun detection is enabled, CTRL/STAT.STICKYORUN is set to 1. A parity error in a write to TARGETSEL deselects the target.

The RDATA parity bit is shown in Figure 4-2 on page 4-95. It appears on the wire immediately after the RDATA[31] bit. The debugger must check for parity errors in the RDATA data transfer phase and retry the read if required.

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 4-2 on page 4-95:

- 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.

4.2.6 Limitations of multi-drop

This section describes the configuration and auto-detection limitations of a multi-drop Serial Wire Debug system.

System configuration

Each device must be configured with a unique target ID, that includes a 4-bit instance ID, to differentiate between otherwise identical targets. This places a limit of 16 such targets in any system, and means that identical devices must be configured before they are connected together to ensure that their instance IDs do not conflict.

Auto-detection of the target

It is not possible to interrogate a multi-drop Serial Wire Debug system that includes multiple devices to establish which devices are connected. Because all devices are selected on coming out of a line reset, no communication with a device is possible without prior selection of that target using its target ID. Therefore, connection to a multi-drop Serial Wire Debug system that includes multiple devices requires that either:

- The host has prior knowledge of the devices in the system and is configured before target connection.
- The host attempts auto-detection by issuing a target select command for each of the devices it has been configured to support. While this is likely to involve a large number of target select commands, it must be possible to iterate through all the supported devices in a reasonable time from the viewpoint of a user of the debug tools.

Note

This means that debug tools cannot connect seamlessly to targets in a multi-drop Serial Wire Debug system that they have never seen before. However, if the debug tools can be provided with the target ID of such targets by the user then the contents of the target can be auto-detected as normal.

To protect against multiple selected devices all driving the line simultaneously SWD protocol version 2 requires:

- For multi-drop SWJ-DP, the JTAG connection is selected out of powerup reset. JTAG does not drive the line. See Chapter 5 *The Serial Wire/JTAG Debug Port (SWJ-DP)*.
- For multi-drop SW-DP, the DP is in the dormant state out of powerup reset. See *Dormant operation* on page 5-113.

4.3 Serial Wire Debug protocol operation

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

The sequences of operations illustrated here are:

- *Successful write operation (OK response)* on page 4-94.
- *Successful read operation (OK response)* on page 4-95.
- *WAIT response to read or write operation request* on page 4-96.
- *FAULT response to read or write operation request* on page 4-96.
- *Protocol error response* on page 4-97.
- *Sticky overrun behavior* on page 4-98.
- *SW-DP write buffering* on page 4-99.

Key to illustrations of operations describes the terms used in the illustrations.

4.3.1 Key to illustrations of operations

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

Start A single start bit, with value 1.

APnDP 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.

RnW 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.

A[2:3] Two bits, giving the A[3:2] address field for the DP or AP register Address:

- For a DPACC access, the register being addressed depends on the A[3:2] value and, if A[3:2]==`0b01`, the value held in SELECT. DPBANKSEL. For details see:
  - — *DP architecture version 1 (DPv1) address map* on page 2-42
  - — *DP architecture version 2 (DPv2) address map* on page 2-43.
- For an APACC access, the register being addressed depends on the A[3:2] value and the value held in SELECT.{APSEL,APBANKSEL}. For details of the addressing see:
  - — *MEM-AP register summary* on page 7-146 for accesses to a MEM-AP register
  - — *JTAG-AP register summary* on page 8-176 for accesses to a JTAG-AP register.

Note

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.

Parity A single parity bit for the preceding packet. See *Parity* on page 4-91.

Stop A single stop bit. In the synchronous SWD protocol this is always 0.

Park A single bit. The host must drive the Park bit HIGH to park the line before tristating it for the turnaround period. This ensures the line is read as HIGH by the target. This is required as the pull-up on the Serial Wire Debug interface is weak. The target reads this bit as 1.

Trn Turnaround. See *Line turn-round* on page 4-90.

Note

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

ACK[0:2] 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 if you require more information about timings of the serial connection to the SW-DP.

4.3.2 Successful write operation (OK response)

On receiving a write package request, if the SW-DP is ready for the WDATA data transfer phase, and there is no error condition, it issues an OK response. This is indicated by an acknowledge response of `0b001`.

This does not apply to writes to TARGETSEL. See *Connection and line reset sequence* on page 4-104.

Therefore, a successful write operation consists of three phases:

1. An eight-bit write packet request, from the host to the target.
2. A three-bit OK acknowledge response, from the target to the host.
3. A 33-bit WDATA data transfer phase, from the host to the target.

By default, there are single-cycle turnaround periods between each of these phases. See *Line turn-round* on page 4-90 for more information.

A successful write operation is shown in Figure 4-1.

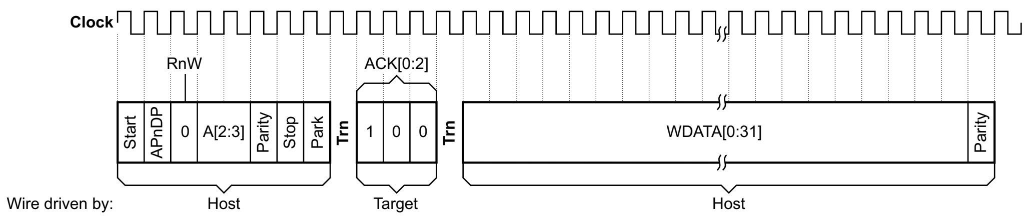


Figure 4-1 Serial Wire Debug successful write operation

The host must start the write transfer immediately after receiving the OK response from the target. This behavior is the same whether the write is to the DP or to an AP. The OK response shown in Figure 4-1 on page 4-94 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. However, the SW-DP can buffer AP writes, as described in *SW-DP write buffering* on page 4-99.

There is no turnaround phase after the data phase. The host is driving the line, and can start the next operation immediately.

4.3.3 Successful read operation (OK response)

On receiving a read packet request, if the SW-DP is ready for the RDATA data transfer phase, and there is no error condition, it issues an OK response. This is indicated by an acknowledge response of `0b001`.

Therefore, a successful read operation consists of three phases:

1. An eight-bit read packet request, from the host to the target.
2. A three-bit OK acknowledge response, from the target to the host.
3. A 33-bit RDATA data transfer phase, where data is transferred from the target to the host.

By default, there are single-cycle turnaround periods between the first and second of these phases, and after the third phase. See *Line turn-round* on page 4-90 for more information. However, there is no turnaround period between the second and third phases, because the line is driven by the target in both of these phases.

Figure 4-2 shows a successful read operation.

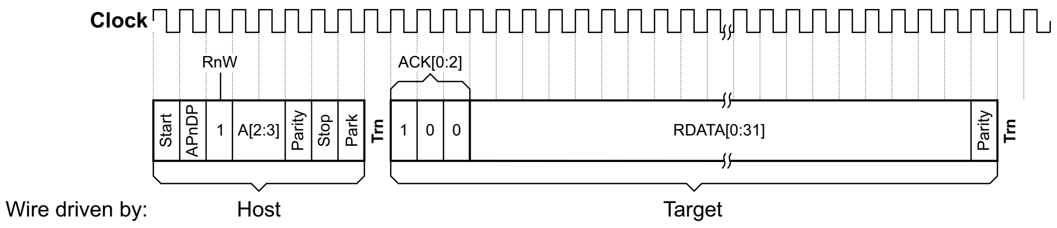


Figure 4-2 Serial Wire Debug successful read operation

If the host requested a read access to the DP then the SW-DP sends the read data immediately after the acknowledgement response.

Read accesses to the AP are posted. This means that the result of the access is returned on the next transfer. This can be another AP register read, or a DP register read of RDBUFF.

To make a series of AP reads a debugger only has to insert one read of the RDBUFF register:

- On the first AP read access, the read data returned is UNKNOWN. The debugger must discard this result.
- The next AP read access, if successful, returns the result of the previous AP read.
- This can repeated for any number of AP reads. Issuing the last AP read packet request returns the last-but-one AP read result.
- The debugger can then read the DP RDBUFF register to obtain the last AP read result.

So that a debugger can recover from line errors, the next transaction after an AP register read can be any DP register read. If the next transaction is a DP register read other than a read of RDBUFF then a following AP register read or read of RDBUFF returns the result of the first AP register read.

If the next transaction following an AP register read is an AP register write or a DP register write then the result of the first AP register read is lost because any following AP register read or read of RDBUFF returns an UNKNOWN value.

4.3.4 WAIT response to read or write operation request

If the SW-DP is not able to process the request from the debugger immediately it must issue a WAIT response. A WAIT response to a read or write packet request consists of two phases:

1. An eight-bit read or write packet request, from the host to the target.
2. 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 *Line turn-round* on page 4-90 for more information.

A WAIT response to a read or write packet request is shown in Figure 4-3.

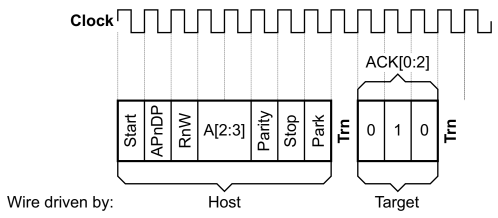


Figure 4-3 Serial Wire Debug WAIT response to a packet request

If overrun detection is enabled then CTRL/STAT.STICKYORUN is set to 1 and a data phase is required on a WAIT response. For more information see *Sticky overrun behavior* on page 4-98.

A WAIT response must not be issued to the following requests. The SW-DP must always process these requests immediately:

- Reads of the DPIDR register.
- Reads of the CTRL/STAT register.
- Writes to the ABORT register.

With any other request, the DP issues a WAIT response if it cannot process the request. This happens if:

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

Normally, when a debugger receives a WAIT response it retries the same operation. 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 ABORT.DAPABORT. 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 SoC interface. However, on receiving a DAP abort request the AP must free up the interface to the Debug Port.

Writing to the ABORT register after receiving a WAIT response enables the debugger to access other parts of the debug system.

4.3.5 FAULT response to read or write operation request

A SW-DP must not issue a FAULT response for:

- Reads of the DPIDR register, which is a read-only register.
- Reads of the CTRL/STAT register, which is a read/write register.
- Writes to the ABORT register, which is a write-only register.

For any other access, the SW-DP issues a FAULT response if any sticky flag is set to 1 in the CTRL/STAT register.

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

1. An eight-bit read or write packet request, from the host to the target.

2. 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 *Line turn-round* on page 4-90 for more information.

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

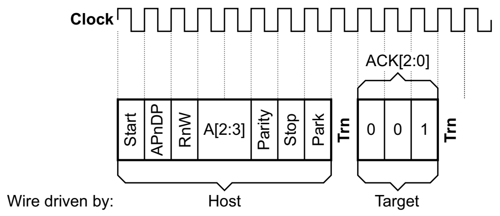


Figure 4-4 Serial Wire Debug FAULT response to a packet request

If overrun detection is enabled then CTRL/STAT.STICKYORUN is set to 1 and a data phase is required on a FAULT response. For more information see *Sticky overrun behavior* on page 4-98.

Use of the FAULT response enables the protocol to remain synchronized. A debugger might stream a block of data and then check the CTRL/STAT register at the end of the block.

In a SW-DP, the sticky error flags are cleared to 0 by writing bits in the ABORT register.

4.3.6 Protocol error response

A protocol error occurs if any of the following occurs:

- The Parity bit does not match the parity of the packet request.

  For more information about the parity checks in the SWD protocol see *Parity* on page 4-91.
- The Stop bit is not 0.
- The Park bit is not 1.
- DLCR.TURNROUND indicates an unsupported turnaround period.

Note

If the Parity bit in the WDATA transfer phase does not match the parity of the data, this does not cause a protocol error response, as the SW-DP has already given its response to the header. For more information, see *Sticky flags and DP error responses* on page 2-34.

If overrun detection is enabled then CTRL/STAT.STICKYORUN is set to 1 and the target must wait until the data phase of the transaction has completed before entering the protocol error state. Otherwise, it enters the protocol error state immediately.

When a protocol error is detected by the SW-DP, the SW-DP does not reply to the packet request and does not drive the line. This is illustrated in Figure 4-5 on page 4-98.

Note

If SWD protocol version 2 is implemented, the SW-DP also does not reply to a TARGETSEL register write packet request.

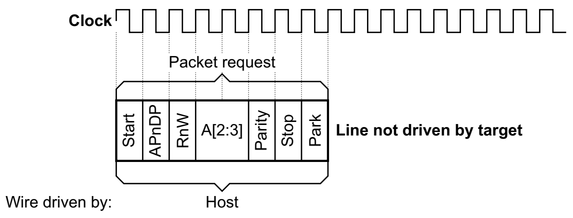


Figure 4-5 Serial Wire Debug protocol error after a packet request

When in protocol error state:

- If the target detects a valid read of the DP DPIDR register, it is IMPLEMENTATION DEFINED whether the target leaves the protocol error state, and gives an OK response.
- If the target detects a valid packet header, other than the read of the DP DPIDR register, or the target detects an IMPLEMENTATION DEFINED number of additional protocol errors, it enters the lockout state.

  ARM recommends that the target enters the lockout state after one more protocol error is detected while in the protocol error state.

If the target cannot leave the protocol error state on a read of the DPIDR register, then the protocol error and lockout states are equivalent.

The target must leave the protocol error state on an line reset.

The target only leaves the lockout state on a line reset.

If the SW-DP implements SWD protocol version 2, it must enter the lockout state after a single protocol error immediately after a line reset. However, if the first packet request detected by the target following line reset is valid it can then revert to entering the lockout state after an IMPLEMENTATION DEFINED number of protocol errors.

Host response to protocol error

If the host does not receive an expected response from the target, it must leave the line not driven for at least the length of any potential data phase and then attempt a line reset. For more information, see *Connection and line reset sequence* on page 4-104.

The host can attempt reads of the DP DPIDR register before attempting a line reset, as the target might respond and leave the protocol error state, but ARM does not recommend this.

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.

4.3.7 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 CTRL/STAT.STICKYORUN flag is set to 1. Subsequent transactions generate FAULT responses, because a sticky flag is set to 1. If overrun detection is enabled, CTRL/STAT.STICKYORUN is also set if there is a FAULT response, protocol error, or line reset.

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

- If the transaction is a read the data in the data phase is UNKNOWN. The target does not drive the line, and the host must not check the parity bit.
- If the transaction is a write the data phase is ignored.

Figure 4-6 on page 4-99 shows the WAIT or FAULT response to a read operation when overrun detection is enabled, and Figure 4-7 on page 4-99 shows the response to a write operation when overrun detection is enabled.

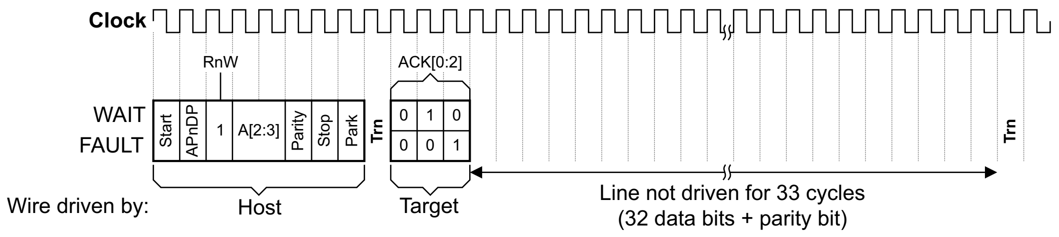


Figure 4-6 SW-DP WAIT or FAULT response to a read operation when overrun detection is enabled

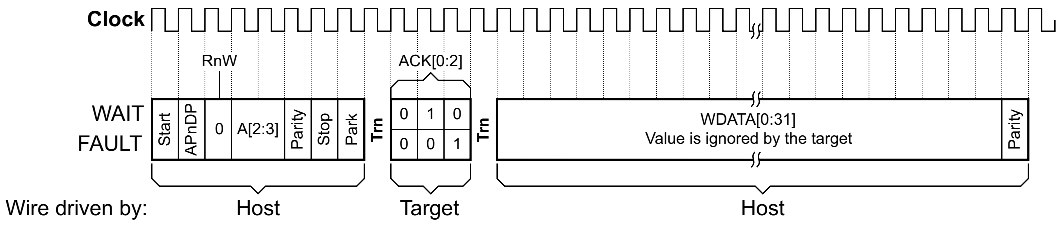


Figure 4-7 SW-DP WAIT or FAULT response to a write operation when overrun detection is enabled

4.3.8 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 CTRL/STAT.WDATAERR flag is set to 1. A buffered write is be 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 must not 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. The DP must not stall an ABORT register access.

This means that if software makes a series of AP write transactions, it might not be possible to determine which transaction failed from examining the ACK responses. However it might be possible to use other enquiries to find which write failed. For example, if when using the *auto-address increment* (AddrInc) feature of a Memory Access Port, software can read the TAR to find the address of the last successful write transaction.

The write buffer must be emptied before the following operations can be performed:

- Any AP read operation.
- Any DP operation other than a read of the 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 2-36 for details of these operations.

If a DP read of the IDCODE or CTRL/STAT register, or a DP write to the ABORT register, is required immediately after a sequence of AP writes, the software must first perform an access that the DP is able to stall. This ensures that the write buffer is emptied before the DP register access is performed. If this 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.

4.3.9 Summary of target responses

The following subsections show the target SW-DP responses for different transaction requests:

- *Target SW-DP responses for DP transaction requests*.
- *Target SW-DP responses for AP transaction requests* on page 4-102.

Target SW-DP responses for DP transaction requests

For DP transaction requests, the register accessed is determined by:

- The value of A[3:2].
- In DPv1 and DPv2, when A[3:2] is 0b01, the value of SELECT.DPBANKSEL.

The behavior of some read transaction requests depends on the register accessed, as Table 4-1 shows.

Table 4-1 shows the target SW-DP response to all possible debugger DP read operation requests.

Table 4-2 on page 4-101 shows the target SW-DP response to all possible debugger DP write operation requests, assuming the WDATA parity check is good.

Table 4-1 Target response summary for DP read transaction requests

A[3:2]	SELECT. DPBANKSEL	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
				ACK	Action
`0b00`	x	x[a]	x[a]	OK	Respond with register value.
`0b01`	`0x0`	x[a]	x[a]	OK	Respond with register value.
	Not `0x0`	No	Yes	OK	Respond with register value.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].

Table 4-1 Target response summary for DP read transaction requests (continued)

A[3:2]	SELECT.DPBANKSEL	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
				ACK	Action
`0b10`	x	No	Yes	OK	Respond by resending the last read value sent to the host. This value is the result of one of: • the most recent AP read • the most recent DP RDBUFF read.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].
`0b11`	x	No	Yes	OK	Respond with the value from the previous AP read, and set CTRL/STAT.READOK bit to 1.
		No	No	WAIT	No data phase, unless overrun detection is enabled[b]. Set CTRL/STAT.READOK bit to 0.
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b]. Set CTRL/STAT.READOK bit to 0.

a. The SW-DP must always give an OK response to a read of the IDCODE or CTRL/STAT register.

b. See *Sticky overrun behavior* on page 4-98 for details of data phase when overrun detection is enabled.

Table 4-2 Target response summary for DP write transaction requests

A[3:2]	Protocol version	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
				ACK	Action
`0b00`	x	x	x	OK	Write WDATA value to ABORT register.
`0b01` or `0b10`	x	No	Yes[a]	OK	Write WDATA value to the selected DP register.
			No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].
`0b11`	v1	No	Yes[a]	OK	Register is reserved, SBZ. Write is ignored.
			No	WAIT	No data phase, unless overrun detection is enabled[b].
		Yes	x	FAULT	No data phase, unless overrun detection is enabled[b].
	v2	x	x	None	Write WDATA to TARGETSEL register[c].

a. Writes might be accepted when other transactions are still outstanding, These writes might be abandoned subsequently. See *SW-DP write buffering* on page 4-99 for more information.

b. See *Sticky overrun behavior* on page 4-98 for details of data phase when overrun detection is enabled.

c. Target gives no response. See *Connection and line reset sequence* on page 4-104.

Fault conditions that are not shown in these tables are described in *Fault conditions not included in the target response tables* on page 4-103.

Target SW-DP responses for AP transaction requests

For AP transaction requests, the register accessed is determined by the value of A[3:2] combined with the values of SELECT.{APSEL,APBANKSEL}. For more information, see *Using the Access Port to access debug resources* on page 1-27.

Table 4-3 summarizes the target SW-DP response to all possible debugger AP read operation requests.

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

Table 4-3 Target response summary for AP read transaction requests

A[3:2]	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
			ACK	Action
`0bxx`	No	Yes	OK	Normally[a], return value from previous AP read[b] and set CTRL/STAT.READOK bit to 1. Initiate AP read of addressed register.
		No	WAIT	No data phase, unless overrun detection is enabled[c]. Set CTRL/STAT.READOK bit to 0.
	Yes	x	FAULT	No data phase, unless overrun detection is enabled[c]. Set CTRL/STAT.READOK bit 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 UNKNOWN.
c. See *Sticky overrun behavior* on page 4-98 for details of data phase when overrun detection is enabled.

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

A[3:2]	Sticky flag set to 1?	AP Ready?	SW-DP (target) response	
			ACK	Action
`0bxx`	No	Yes[a]	OK	Normally[b], write WDATA value to the indicated AP register.
		No	WAIT	No data phase, unless overrun detection is enabled[c].
	Yes	x	FAULT	No data phase, unless overrun detection is enabled[c].

a. Writes might be accepted when other transactions are still outstanding, These writes might be abandoned subsequently. See *SW-DP write buffering* on page 4-99 for more information.
b. If pushed-verify or pushed-compare is enabled, the write is converted to a read of the addressed AP register, and the value returned by this read is compared with the supplied WDATA value, see *Pushed-compare and pushed-verify operations* on page 2-36 for more information.
c. See *Sticky overrun behavior* on page 4-98 for details of data phase when overrun detection is enabled.

Fault conditions that are not shown in these tables are described in *Fault conditions not included in the target response tables* on page 4-103.

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 4-1 on page 4-100 to Table 4-4 on page 4-102:

Protocol error

If there is a protocol error 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. See *Protocol error response* on page 4-97.

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 4-2 on page 4-101. When the parity check is performed and fails, the CTRL/STAT.WDATAERR flag is set to 1.

4.3.10 Summary of host responses

Every access by a debugger to a SW-DP starts with an operation request. *Summary of target responses* on page 4-100 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 4-1 on page 4-100 to Table 4-4 on page 4-102. Table 4-5 summarizes how the debugger must respond to this acknowledgement, for all possible cases.

Note

For SWD protocol version 2, this table does not apply to writes to TARGETSEL. See *Connection and line reset sequence*.

Table 4-5 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[a] and protocol.	Might have to repeat original read request or use the RESEND register if a parity or protocol fault occurs and are unable to flag data as invalid[b].
	Invalid ACK	Back off because of possible data phase.	Can check CTRL/STAT register to see if the response sent was OK.
W	OK	Send WDATA.	Validity of this transfer is confirmed on next access.
	Invalid ACK	Back off to ensure that target does not capture next header as WDATA.	Repeat the write access. A FAULT response is possible if the first response was sent as OK but not recognized as valid by the debugger. The subsequent write is not affected by the first, misread, response.
x	WAIT	No data phase, unless overrun detection is enabled[c].	Normally, repeat the original operation request. See *WAIT response to read or write operation request* on page 4-96 for more information.
	FAULT	No data phase, unless overrun detection is enabled[c].	Can send new headers, but only an access to DP register addresses `0b0X` gives a valid response.
	No ACK	Back off because of possible data phase.	Can attempt IDCODE register read. Otherwise reset connection and retrain. See *Protocol error response* on page 4-97.

a. See *Parity* on page 4-91 for details of the parity checking.

b. The host debugger might support corrupted reads, or it might have to re-try the transfer.

c. If overrun detection is enabled, a data phase is required. See *Sticky overrun behavior* on page 4-98 for a description of the behavior on read and write operations.

4.4 Serial Wire Debug interface

The Serial Wire Debug protocol operates with a synchronous serial interface. This uses a single bidirectional data signal, and a clock signal.

This section gives an overview of the physical Serial Wire Debug interface.

4.4.1 Line interface

The Serial Wire Debug interface uses a single bidirectional data pin, **SWDIO**. That is, the same signal is used for both host and target sourced signals.

The Serial Wire Debug interface is synchronous, and requires a clock pin, **SWCLK**.

The clock can be sourced from the target and exported, or provided by the host. This clock is then used by the host as a reference for generation and sampling of data so that the target is not required to perform any over-sampling.

Both the target and host are capable of driving the bus HIGH and LOW or tristating it. The ports must be able to tolerate short periods of contention that might occur because of a loss of synchronization.

The clock can be asynchronous to any system clock, including the debug logic clock. The Serial Wire Debug interface clock can be stopped when the debug port is idle, see *Introduction to the ARM Serial Wire Debug (SWD) protocol* on page 4-89.

4.4.2 Line pull-up

So that the line is in a known state when neither host nor target is driving the line, a 100KΩ pull-up is required at the target, but this can only be relied on to maintain the state of the wire. If the wire is driven LOW and released, the pull-up resistor eventually returns the line to the HIGH state, but this takes many clock periods.

The pull-up is intended to prevent false detection of signals when no host is connected. This means it must be of a suitably high value to reduce current consumption from the target when the host actively pulls down the line.

Note

A small current drains from the target whenever the line is driven LOW. If the interface is left connected for extended periods when the target has to use a low-power mode, the line must be held HIGH, or reset, by the host until the interface is activated.

4.4.3 Connection and line reset sequence

A debugger must use a line reset sequence to ensure that hot-plugging the serial connection does not result in unintentional transfers. The line reset sequence ensures that the SW-DP is synchronized correctly to the header that signals a connection.

The SWD interface does not include a reset signal. A line reset is achieved by holding the data signal HIGH for at least 50 clock cycles, followed by at least two idle cycles. Figure 4-8 shows the interface timing for a line reset followed by a DP DPIDR register read.

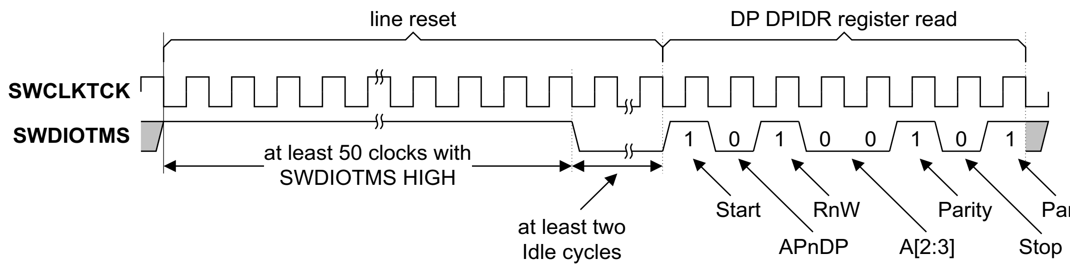


Figure 4-8 Line reset sequence followed by a DP DPIDR register read

A line reset is required when first connecting to the target. A line reset might be required following a protocol error. See *Protocol error response* on page 4-97.

A line reset resets DLCR.

Note

Other SW-DP registers are reset only by a powerup reset.

When waiting for a packet header, if the target detects a sequence of 50 clock cycles with the data signal held HIGH, followed by at least two idle cycles, it must enter the reset state. It is IMPLEMENTATION DEFINED whether a sequence of 50 clock cycles with the data signal held HIGH, detected at any other time, causes the interface to enter the reset state.

The only valid transactions in reset state are:

- A read of the DPIDR register. This takes the connection out of reset state.
- One of the switching sequences defined by *SWD and JTAG select mechanism* on page 5-110, if implemented.
- A write to the TARGETSEL register, if SWD protocol version 2 is implemented. If this selects the target, the interface remains in reset state.

  Note

  Only writes to TARGETSEL immediately after entry to the reset state can select or deselect the target. See *Target selection protocol, SWD protocol version 2*.

Any of these sequences can be aborted by a second line reset. The behavior of the target is UNPREDICTABLE if any other transaction is made in reset state.

If the host does not see an expected response when reading the DPIDR register, it must retry the reset sequence. This is because the target might have been in a state where, for example, it treated the initial line reset as a data phase of a transaction and so did not detect it as a valid line reset. If this is the case, the target detects the line reset as a protocol error and requires a second line reset to respond correctly.

If overrun detection is enabled then the line reset sets CTRL/STAT.STICKYORUN to 1.

4.4.4 Target selection protocol, SWD protocol version 2

To select a new target a host must:

1. Perform a line reset. See Figure 4-9.
2. Write to DP register `0xC`, TARGETSEL, where the data indicates the selected target. The target response must be ignored. See Figure 4-9.
3. Read from the DP register `0x0`, DPIDR, to verify that the target has been successfully selected.

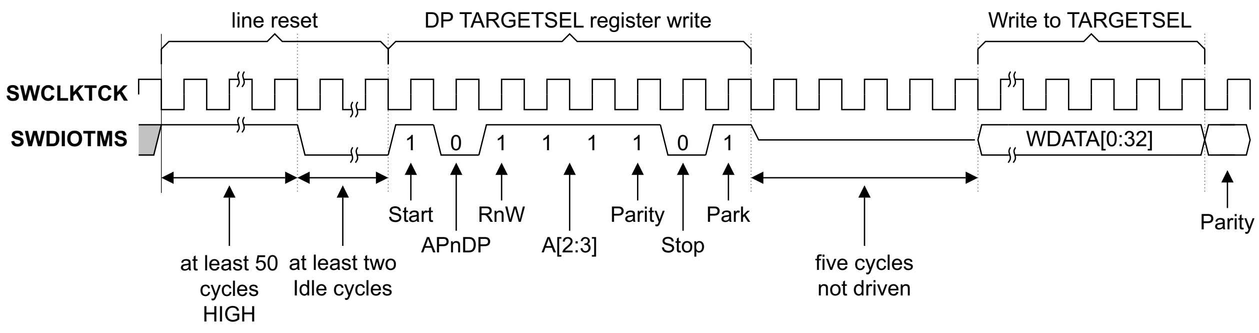


Figure 4-9 Line reset sequence followed by a DP TARGETSEL write

A write to the TARGETSEL register must always be followed by a read of the DPIDR register or a line reset. If the response to the DPIDR read is incorrect, or there is no response, the host must start the sequence again.

The target is selected on receiving a line reset sequence.

Following receipt of a line reset sequence, if the target then receives a write to TARGETSEL that does not select the same target, the target is deselected.

When deselected, the target ignores all accesses and must not drive the line. Only a write to TARGETSEL immediately following a line reset sequence can select or deselect the target. Writes to TARGETSEL at any other time are UNPREDICTABLE.

If the target encounters a protocol error, it becomes deselected. Specifically, it does not respond to a read of the DPIDR register.

For more information, including the required behavior of the target during the response phase of the write to the TARGETSEL register, see *Protocol errors, SW-DP* on page 2-36.

A parity error in the data phase of a write to the TARGETSEL register does not set the CTRL/STAT.WDATAERR bit to 1. A parity error in the data phase of a write to the TARGETSEL register is treated as a protocol error.

Accesses to the TARGETSEL register are not affected by the state of the CTRL/STAT.{WDATAERR, STICKYERR, STICKYCMP, STICKYORUN} bits.

Implementations of Serial Wire Debug protocol version 2 must also support dormant operation. See *Dormant operation* on page 5-113.

Chapter 5
The Serial Wire/JTAG Debug Port (SWJ-DP)

This appendix describes multiple protocol interoperability as implemented in the *Serial Wire/JTAG Debug Port* (SWJ-DP) CoreSight component. It contains the following sections:

- *Serial Wire/JTAG Debug Port (SWJ-DP)* on page 5-108.
- *SWD and JTAG select mechanism* on page 5-110.
- *Dormant operation* on page 5-113.
- *Restriction on switching* on page 5-120.

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

The SWJ-DP interface provides a mechanism to select between *Serial Wire Debug* (SWD) and JTAG Data Link protocols. This enables the JTAG-DP and SW-DP to share pins.

SWJ-DP is a combined JTAG-DP and SW-DP that enables a debug probe to connect to the target using either the SWD protocol or JTAG. To make efficient use of package pins, the SWD interface shares, or overlays, the JTAG pins, and a mechanism is provided to switch between JTAG-DP and SW-DP, depending on which debug probe type is connected. The SWJ-DP behaves like a JTAG-DP device if normal JTAG sequences are sent to it.

5.1.1 SWJ-DP structure

The SWJ-DP logically consists of a wrapper around the JTAG-DP and SW-DP. Its function is to select JTAG or SWD as the Data Link protocol and enable either JTAG-DP or SW-DP as the interface to the DAP.

Figure 5-1 shows the wrapped SW-DP and JTAG-DP.

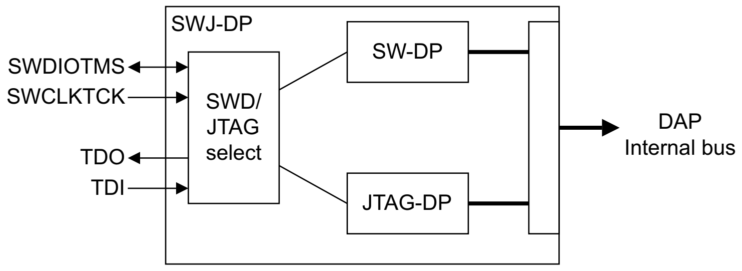


Figure 5-1 SWJ-DP conceptual model

There is no requirement to implement separate SW-DP and JTAG-DP blocks within the SWJ-DP wrapper.

The number, type and location of APs accessed by the SWJ-DP must be the same, and must access the same debug resources, but there is no requirement to implement these as shared APs.

This means that tools must not rely on the state of either DP, or any AP accessed through the DP, persisting when the other DP is selected. On switching DPs, the debugger must re-initialize the DAP, including setting the CTRL/STAT.{CDBGPWRUPREQ, CSYSPWRUPREQ} bits correctly.

The JTAG-DP and SW-DP programmers' models do not have to implement the same Debug Port architecture version. See Chapter 2 *The Debug Port (DP)*.

Note

If the JTAG protocol is never used, a pull-down on TDI is required at the target.

5.1.2 SWJ-DP operation

SWJ-DP enables a SoC to be designed for used in systems that require either a JTAG interface or a Serial Wire Debug interface. There is however a trade-off between the number of pins used and compatibility with existing hardware and test equipment.

There are several scenarios where the use of a JTAG debug interface must be maintained, such as:

- To enable inclusion in an existing scan chain, generally on-chip TAPs used for test or other purposes.
- To enable the device to be cascaded with legacy devices which use JTAG for debug, although this can also be supported using a *JTAG Access Port* (JTAG-AP).
- To enable use of existing debug hardware with the corresponding test TAPs, for example, in *Automatic Test Equipment* (ATE).

A SoC fitted with SWJ-DP support can be connected to legacy JTAG equipment without any requirement to make changes. If a SWD tool is available, then only two pins are required, instead of the usual four used for JTAG. Two pins are therefore released for alternative functions.

These two released pins can only be used when there is no conflict with their use in JTAG operation. In addition, to support use of SWJ-DP in a scan chain with other JTAG devices, the default state after reset must be to use these released pins for their JTAG operation. However, if the direction of the alternative function is compatible with being driven by a JTAG debug device, the transition of the JTAG TAP to the Shift-DR or Shift-IR state can transition these pins from their alternative function to JTAG operation.

The alternate function cannot be used while the SoC is being used in JTAG operation.

The switching scheme is arranged so that, provided there is no conflict on the **TDI** and **TDO** pins, a JTAG debugger is able to connect by sending a specific sequence.

The connection sequence used for SWD is safe when applied to the JTAG interface, even when hot-plugged, enabling the debugger to continually retry its access sequence. A sequence with **TMS** HIGH ensures that all parts of the SWJ-DP are in a known reset state. The pattern that selects SWD has no effect on JTAG devices. SWJ-DP is compatible with a free-running **TCK**, or a gated clock, that is supplied by the external tools.

5.1.3 SWD and JTAG interface

The external JTAG interface has four mandatory pins:

TCK	Test Clock.
TMS	Test Mode State.
TDI	Test Data In.
TDO	Test Data Out.

There is also an optional reset pin, **nTRST**, which, when available, can reset the TAP controller's state machine. Debug ports also require a separate system reset signal that is asserted, for example, at powerup.

The SWD interface has two pins:

SWDIO	Data In/Out.
SWCLK	Clock. The clock can be either an input or an output.

To enable sharing of the connector for either JTAG or SWD, connections must be made external to the SWJ-DP block, see *SWJ-DP conceptual model* on page 5-108. The combined SWJ-DP includes two shared pins:

SWDIOTMS	Shared for **SWDIO** and **TMS** data signals.
SWCLKTCK	Shared for **SWCLK** and **TCK** clock signals.

In addition, **TMS** must be a bidirectional pin to support the bidirectional **SWDIO** pin for Serial Wire Debug protocol.

Note

- When the Serial Wire Debug protocol is being used, the JTAG pins **TDI**, **TDO**, and **nTRST** are expected to be reused.
- An SWJ-DP can be implemented in a package where the JTAG pins **TDI**, **TDO**, and **nTRST** are not connected because an SWJ-DP is designed to allow selection of SWD protocol without using these JTAG pins.

5.2 SWD and JTAG select mechanism

SWJ-DP enables either a SWD or JTAG protocol to be used on the debug port. To do this, it implements a *watcher circuit* that detects a specific 16-bit select sequence on **SWDIOTMS**:

- A 16-bit sequence to switch from JTAG to SWD operation.
- A 16-bit sequence is to switch from SWD to JTAG.

ARM deprecates use of these sequences on devices where the dormant state of operation is implemented, and recommends using a transition through dormant state instead. For more information, see *Dormant operation* on page 5-113.

SWJ-DP defaults to JTAG operation on powerup reset and therefore the JTAG protocol can be used from reset without sending a select sequence.

Switching from one protocol to the other can only occur when the selected interface is in its reset state. The JTAG TAP state machine must be in its Test-Logic-Reset (TLR) state and the SWD interface must be in line-reset. The powerup reset state for a JTAG TAP state machine is the Test-Logic-Reset state.

Having detected a switching sequence, SWJ-DP does not detect more sequences until after a reset condition. If JTAG is selected, the JTAG TAP state machine being in the Test-Logic-Reset state is the reset condition. If SWD is selected, a line reset is the reset condition.

Figure 5-2 is a simplified state diagram that shows how SWJ-DP transitions between selected, detecting, and selection states.

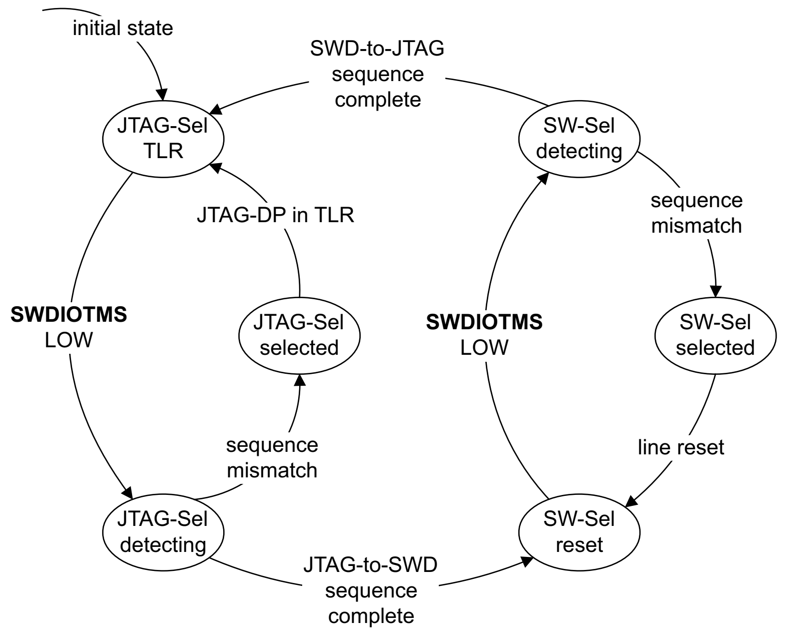


Figure 5-2 SWD and JTAG select state diagram

Note

In Figure 5-2:

- The JTAG-to-SWD sequence terminates in the SW-Sel reset state.
- The SWD-to-JTAG sequence terminates in the JTAG-Sel TLR state.

The recommended sequences end with a reset sequence for the selected state, to ensure the target is in the relevant reset state.

5.2.1 Switching from JTAG to SWD operation

To switch SWJ-DP from JTAG to SWD operation:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that the current interface is in its reset state. The JTAG interface only detects the 16-bit JTAG-to-SWD sequence starting from the Test-Logic-Reset state.
2. Send the 16-bit JTAG-to-SWD select sequence on **SWDIOTMS**.
3. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that if SWJ-DP was already in SWD operation before sending the select sequence, the SWD interface enters line reset state.

The 16-bit JTAG-to-SWD select sequence is `0b0111 1001 1110 0111`, *most-significant-bit* (MSB) first. This can be represented as one of the following:

- `0x79E7`, transmitted MSB first.
- `0xE79E`, transmitted *least-significant-bit* (LSB) first.

Figure 5-3 shows the interface timing.

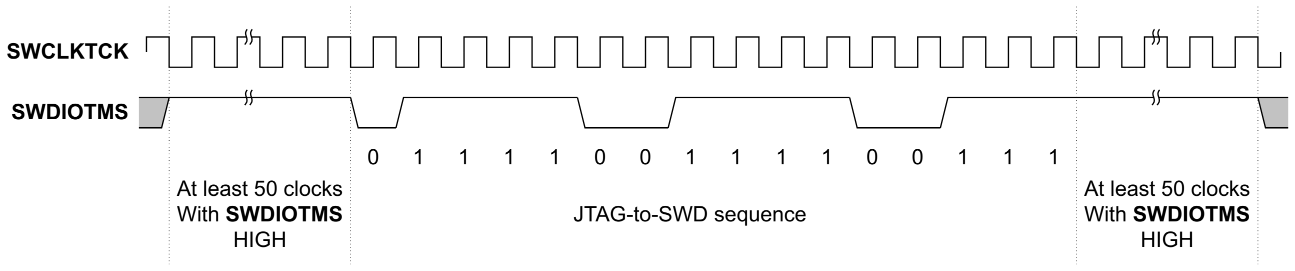

SWCLKTCK
SWDIOTMS
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
At least 50 clocks With SWDIOTMS HIGH
JTAG-to-SWD sequence
At least 50 clocks With SWDIOTMS HIGH

Figure 5-3 JTAG-to-SWD sequence timing

This sequence has been chosen to ensure that the SWJ-DP switches to using SWD whether it was previously expecting JTAG or SWD. As long as the 50 cycles with **SWDIOTMS** HIGH sequence is sent first, the JTAG-to-SWD select sequence does not affect SW-DP, or the SWD and JTAG protocols used in the SWJ-DP, and any other TAP controllers that might be connected to **SWDIOTMS**.

On selecting SWD operation, the SWD interface is in a reset state. See *Connection and line reset sequence* on page 4-104.

5.2.2 Switching from SWD to JTAG operation

To switch SWJ-DP from SWD to JTAG operation:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that the current interface is in its reset state. The SWD interface only detects the 16-bit SWD-to-JTAG sequence when it is in the reset state.
2. Send the 16-bit SWD-to-JTAG select sequence on **SWDIOTMS**.
3. Send at least 5 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that if SWJ-DP was already in JTAG operation before sending the select sequence, the JTAG TAP enters the Test-Logic-Reset state.

The 16-bit SWD-to-JTAG select sequence is `0b0011 1100 1110 0111`, MSB first. This can be represented as either of the following:

- `0x3CE7`, transmitted MSB first
- `0xE73C`, transmitted LSB first.

Figure 5-4 on page 5-112 shows the SWD-to-JTAG sequence timing.

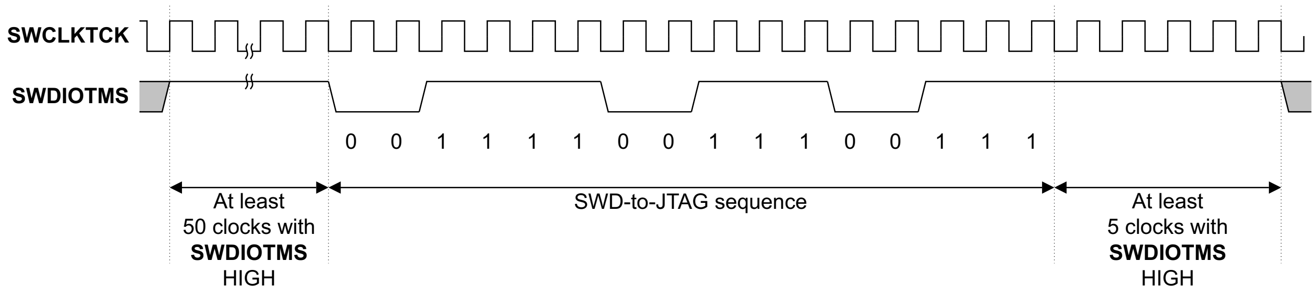

SWCLKTCK
SWDIOTMS
0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1
At least 50 clocks with **SWDIOTMS** HIGH
SWD-to-JTAG sequence
At least 5 clocks with **SWDIOTMS** HIGH

Figure 5-4 SWD-to-JTAG sequence timing

This sequence has been chosen to ensure that the SWJ-DP switches to using JTAG whether it was previously expecting JTAG or SWD. If the **SWDIOTMS** HIGH sequence is sent first, the SWD-to-JTAG select sequence does not affect SW-DP, or the SWD and JTAG protocols used in the SWJ-DP, and any other TAP controllers that might be connected to **SWDIOTMS**.

Figure 5-4 shows that the SWD-to-JTAG sequence begins with two **SWDIOTMS** LOW cycles after the line reset. No additional **SWDIOTMS** LOW cycles are allowed.

5.3 Dormant operation

An alternative to the selection mechanism for switching between JTAG and SWD operation described in *SWD and JTAG select mechanism* on page 5-110 is the *dormant* state of operation.

To switch between JTAG and SWD operation, a debugger must first place the target into dormant state, and then transition to the required operating state.

Using dormant state allows the target to be placed into a quiescent mode, allowing devices to inter-operate with other devices implementing other protocols. Those other protocols must also implement a quiescent state, with a mechanism for entering and leaving that state that is compatible, but not necessarily compliant, with the SWJ-DP and SW-DP protocols.

This third state of operation is required by SWJ-DP and SW-DP implementations that implement Serial Wire Debug protocol version 2. Serial Wire Debug protocol version 2 is described in Chapter 4 *The Serial Wire Debug Port (SW-DP)*. Otherwise, support for dormant state is IMPLEMENTATION DEFINED. In dormant state the target must ignore any stimulus, with any timing, other than a defined Selection Alert sequence.

The Selection Alert sequence must be followed by a protocol-specific Activation code.

Selection of dormant state is possible when either JTAG or SWD operation is selected. Figure 5-5 extends the state diagram of Figure 5-2 on page 5-110 to include selection of dormant state, for an SWJ-DP implementation.

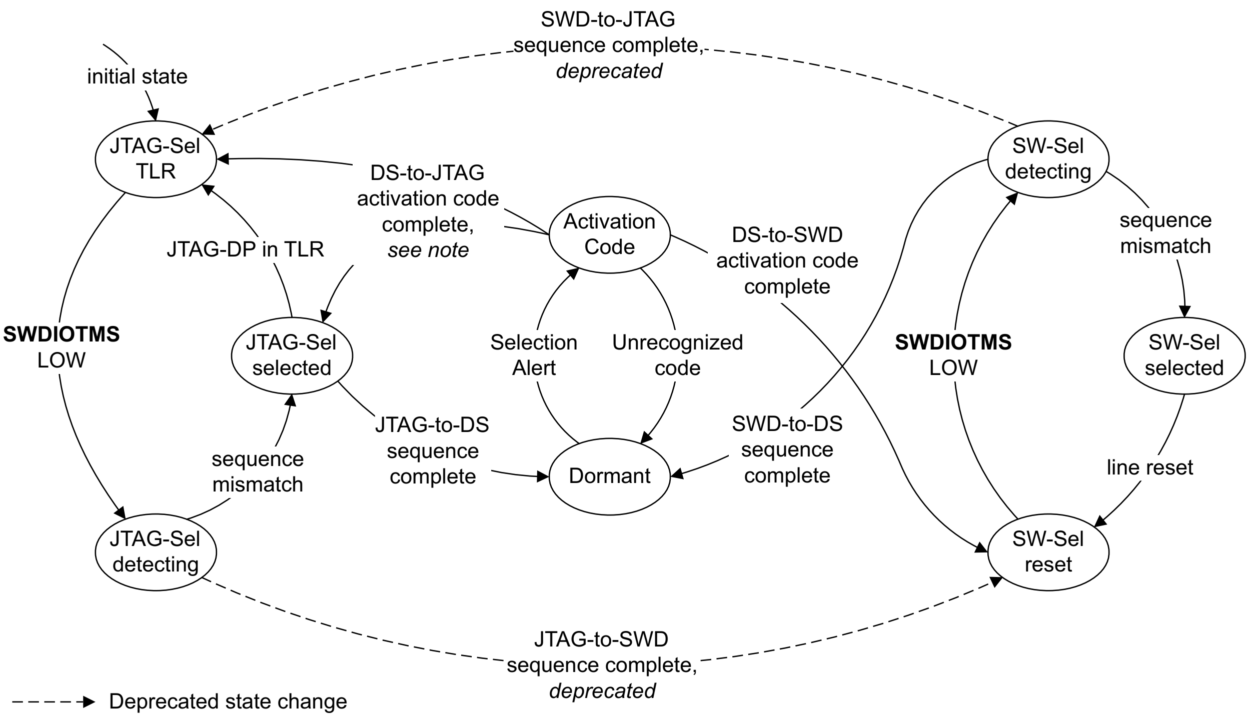


Figure 5-5 SWJ-DP selection of JTAG, SWD, and dormant states

Note

- Following the DS-to-JTAG activation code, the JTAG TAP is in either the Test-Logic-Reset state or Run-Test/Idle state, and therefore this state machine is in either the JTAG-Sel TLR state or the JTAG-Sel selected state. Normally, the TAP state returned to is the TAP state left from. However, it is also possible to reset the JTAG TAP state machine when JTAG is not the selected protocol.

  ARM recommends the DS-to-JTAG sequence is followed by a single clock with **SWDIOTMS** LOW to ensure the TAP is in the Run-Test/Idle state.

- The DS-to-SWD sequence is shown terminating in the SW-Sel reset state. The recommended sequence ends with a line reset to ensure the target is in the reset state.

5.3.1 Use of Dormant state other than in SWJ-DP

A Serial Wire Debug device that does not implement JTAG can nevertheless implement dormant state, and inter-operate with SWJ-DP and other JTAG devices that also implement dormant state. In this case:

- The operating mode selection state machine is simplified.
- The initial state, entered on a powerup reset, is dormant state.

Figure 5-6 shows the state diagram for an SW-DP that implements protocol version 2, meaning it supports dormant state.

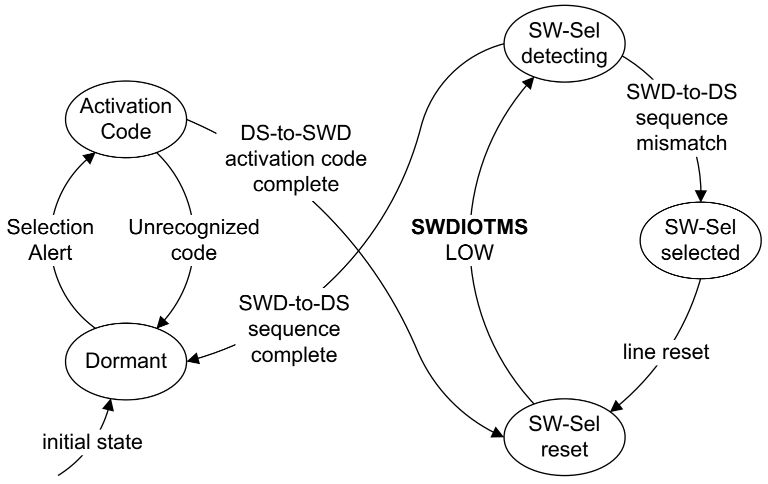


Figure 5-6 SW-DP selection of SWD, and dormant states

This means multi-drop SWJ-DP, SW-DP and JTAG TAPs can share a physical connection to a host, as shown in Figure 5-7. These different devices can be in different physical packages, or on different dies in a single package, or on a single die.

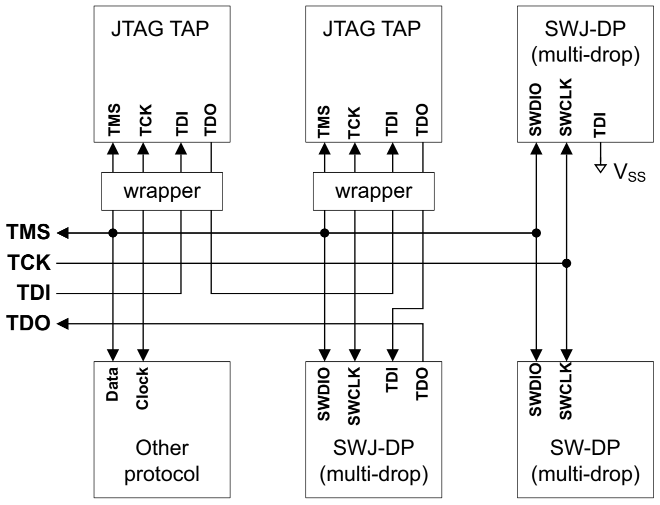


Figure 5-7 Multiple JTAG, SW, SWJ (multi-drop) and other protocol devices on shared connection

5.3.2 JTAG to Dormant Switching

To switch from JTAG to dormant a debugger must:

1. Send at least 5 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This places the JTAG TAP state machine into the Test-Logic-Reset state, and selects the IDCODE instruction.
2. Send the recommended 31-bit JTAG-to-DS select sequence on **SWDIOTMS**.

The recommended 31-bit JTAG-to-DS select sequence is `0b010_1110_1110_1110_1110_1110_1110_0110`, MSB first. This can be represented as either:

- `0x2EEEEEE6` transmitted MSB first, that is, starting from bit 30.
- `0x33BBBBBA` transmitted LSB first.

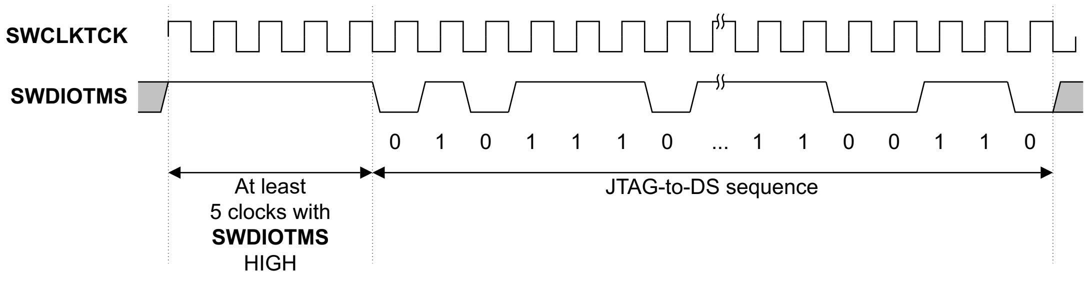


Figure 5-8 Recommended JTAG-to-DS sequence timing

Requirements for implementations

The JTAG-to-DS sequence is the shortest sequence that switches from JTAG-to-DS. For compatibility with other standards, all JTAG devices that implement dormant state must recognize other sequences as valid JTAG-to-DS select sequences.

The full sequence is defined around the concept of a zero-bit-DR-scan (ZBS or ZBS scan) which is in turn defined by transitions of the JTAG TAP state machine. A ZBS is defined as any JTAG TAP state machine sequence that starts at Capture-DR and ends in Update-DR without passing through Shift-DR.

Examples of a ZBS are:

- **Capture-DR → Exit1-DR → Update-DR**
- **Capture-DR → Exit1-DR → Pause-DR → … → Pause-DR → Exit2-DR → Update-DR**

The sequence also uses the ZBS count, which is defined as follows:

- The ZBS count is unlocked and reset to zero if the TAP state machine enters either the Select-IR-Scan or Test-Logic-Reset state. This includes asynchronously entering Test-Logic-Reset following assertion of nTRST. At reset, the ZBS count is unlocked and reset to zero.
- On entering Update-DR at the end of a ZBS scan, if the ZBS count is unlocked and less than seven, it is incremented by one.
- The counter does not increment past seven. On entering Update-DR at the end of a ZBS scan, if the ZBS count is unlocked and equal to seven, it is not incremented. The count does not wrap to zero.
- The ZBS count is locked if the TAP state machine enters the Shift-DR state and the ZBS count is not zero.

The JTAG-to-DS sequence is defined as any sequence of TAP state machine transitions that terminates in the Run-Test/Idle state with a locked ZBS count of six. On entering Run-Test/Idle, the target is placed into dormant state (DS).

The behavior of the target on entering Run-Test/Idle with other locked ZBS counts is IMPLEMENTATION DEFINED.

Although the recommended JTAG-to-DS sequence starts by placing the JTAG TAP state machine in the Test-Logic-Reset state, this is not a requirement for recognizing the JTAG-to-DS sequence. Tools must, however, ensure the Instruction Register (IR) is loaded with either the BYPASS or IDCODE instruction before placing the target into the dormant state. If the IR is not loaded with either of these instructions when the target is put into dormant state, the behavior is unpredictable.

The pseudocode function `EnterDormantState` describes the function of the JTAG-to-DS sequence detector. It is notionally called on every TAP state machine transition. The function's argument is the state being entered, and the function's result is a Boolean indicating whether dormant state must be entered.

For details of the pseudocode language, see Appendix C *Pseudocode Definition*.

```
enumeration TAPState {
    TestLogicReset, RunTestIdle,
    SelectDRScan, CaptureDR, ShiftDR, Exit1DR, PauseDR, Exit2DR, UpdateDR,
    SelectIRScan, CaptureIR, ShiftIR, Exit1IR, PauseIR, Exit2IR, UpdateIR};

boolean shiftDRflag = FALSE;
integer ZBScount = 0;
boolean ZBSlocked = FALSE;

// EnterDormantState()
// ===================

boolean EnterDormantState(TAPState state)
    case state of
        when CaptureDR
            shiftDRflag = FALSE;
        when ShiftDR
            shiftDRflag = TRUE;

            if ZBScount != 0 then ZBSlocked = TRUE;
        when UpdateDR
            if !ZBSlocked && !shiftDRflag && ZBScount < 7 then
                ZBScount = ZBScount + 1;
        when SelectIRScan, TestLogicReset
            ZBScount = 0;  ZBSlocked = FALSE;
        return state == RunTestIdle && ZBSlocked && ZBScount == 6;
```

Note

If the JTAG-to-DS sequence is terminated by a entering the Test-Logic-Reset state, an SWJ-DP can immediately detect a JTAG-to-SWD sequence.

5.3.3 SWD to Dormant switching

To switch from SWD to dormant:

1. Send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures the SWD interface is in the reset state. The target only detects the SWD-to-DS sequence when it is in the reset state.
2. Send the 16-bit SWD-to-DS select sequence on **SWDIOTMS**.

The 16-bit SWD-to-DS select sequence is 0b0011_1101_1100_0111, MSB first. This can be represented as either:

- `0x3DC7` transmitted MSB first.
- `0xE3BC` transmitted LSB first.

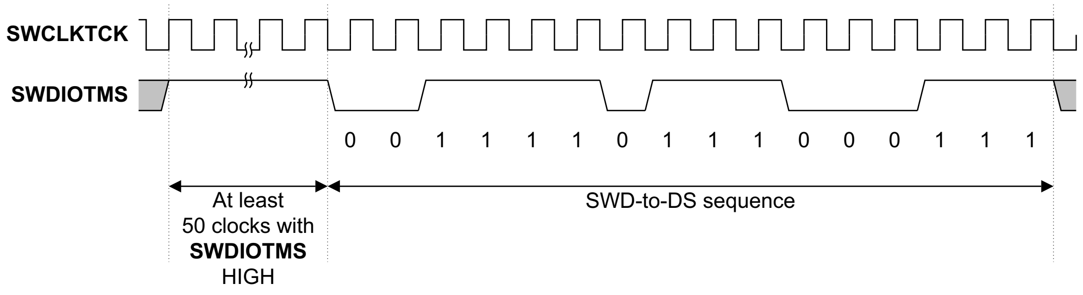


Figure 5-9 SWD-to-DS sequence timing

Figure 5-9 shows that the SWD-to-DS sequence begins with two **SWDIOTMS** LOW cycles after the line reset. No additional **SWDIOTMS** LOW cycles are allowed.

5.3.4 Switching out of Dormant state

The sequence to switch out of dormant state is considerably longer to avoid it being generated accidentally by whichever alternative protocol is in use. The sequence is long enough that it is statistically highly improbable that it is generated any other way.

To switch out of dormant state:

1. Send at least 8 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This is to ensure the target is not in the middle of detecting a Selection Alert sequence. The target is permitted to detect the Selection Alert sequence even if this 8-cycle sequence is not present.
2. Send the 128-bit Selection Alert sequence on **SWDIOTMS**.
3. Send 4 **SWCLKTCK** cycles with **SWDIOTMS** LOW. The target must ignore the value on **SWDIOTMS** during these cycles.
4. Send the required activation code sequence on **SWDIOTMS**.
5. Send a sequence to place the target into a known state
   - If selecting JTAG, the target is in either the Run/Test Idle or Test-Logic-Reset states, see the *Note* that follows Figure 5-5 on page 5-113 for more information. ARM recommends that the debugger sends one **SWCLKTCK** cycle with **SWDIOTMS** LOW. This ensures that the TAP state machine is in the Run-Test/Idle state. Alternatively, send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH to ensure the TAP state machine is in the Test-Logic/Reset state.
   - If selecting SWD, the target is in the protocol error state. The debugger must send at least 50 **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures the SWD interface is in the line reset state.

   Note

   The Activation code selects a protocol, not a target. In a multidrop Serial Wire Debug system with multiple SW-DPs, a target must then be selected. For more information, see *Target selection protocol, SWD protocol version 2* on page 4-105.

The Selection Alert sequence, in binary, is

```
0100_1001_1100_1111_1001_0000_0100_0110_1010_1001_1011_0100_1010_0001_0110_0001_
1001_0111_1111_0101_1011_1011_1100_0111_0100_0101_0111_0000_0011_1101_1001_1000
```

This sequence is sent MSB first. This can be represented as either:

- `0x49CF9046 A9B4A161 97F5BBC7 45703D98` transmitted MSB first.
- `0x19BC0EA2 E3DDAFE9 86852D95 6209F392` transmitted LSB first.

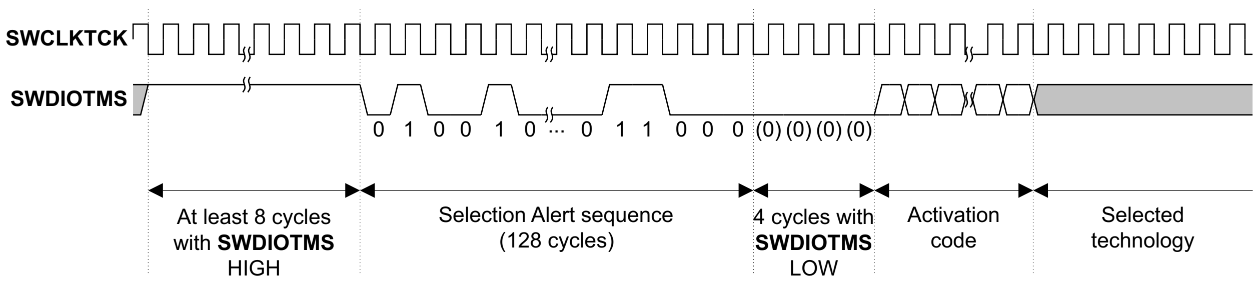

SWCLKTCK
SWDIOTMS
0 1 0 0 1 0 ··· 0 1 1 0 0 0 (0) (0) (0) (0)
At least 8 cycles with SWDIOTMS HIGH
Selection Alert sequence (128 cycles)
4 cycles with SWDIOTMS LOW
Activation code
Selected technology

Figure 5-10 Selection Alert sequence

——— **Note** ———

The Selection Alert sequence can be generated by implementing a Linear Feedback Shift Register (LFSR) implementing feedback on bits 6, 5, 3 and 0, starting in the state 0b1001001 and shifting out one bit from bit 0 each cycle. The sequence starts with a zero start bit and continues with the output of the LFSR.

———————

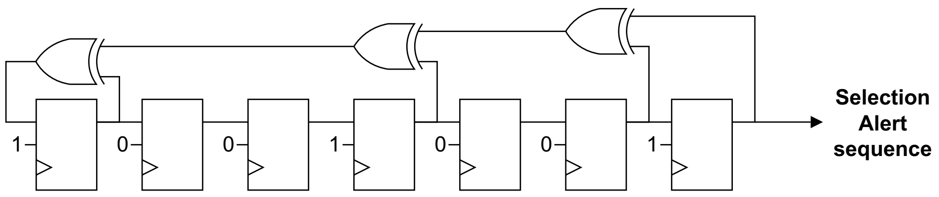


Figure 5-11 LSFR for generating Selection Alert sequence

The value of the activation code depends on whether SWD or JTAG operation is to be requested. Table 5-1 defines the recommended activation codes a debugger must use for JTAG devices, SW-DP devices and SWJ-DP devices. These sequences are sent MSB first.

Table 5-1 Activation codes

Activation code	Value, MSB first	Devices activated				Protocol selected
		Other JTAG	ADIv5 Debug Ports			
			JTAG	SW	SWJ	
JTAG-Serial	`0b0000_0000_0000`	Yes	Yes	No	Yes	JTAG
ARM CoreSight SW-DP	`0b0101_1000`	No	No	Yes	Yes	SWD
ARM CoreSight JTAG-DP	`0b0101_0000`	No	Yes	No	Yes	JTAG

JTAG online activation codes

For compatibility with other standards, all JTAG devices that implement dormant state using the ADIv5 defined selection alert sequence, must recognize other sequences as valid JTAG-Serial activation codes.

Figure 5-12 shows, as a state diagram, the sequence that a JTAG device must recognize.

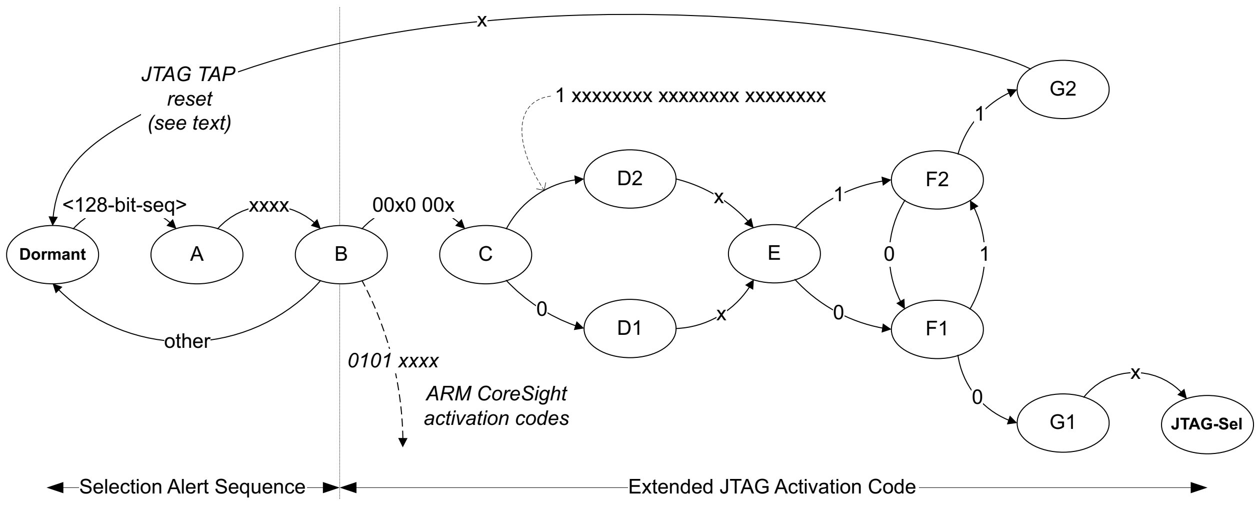

x
JTAG TAP reset (see text)
1 xxxxxxxx xxxxxxxx xxxxxxxx
G2
1
D2
F2
<128-bit-seq>
xxxx
00x0 00x
x
1
Dormant
A
B
C
E
0
1
0
D1
x
0
F1
other
0101 xxxx
ARM CoreSight activation codes
x
0
G1
JTAG-Sel
Selection Alert Sequence
Extended JTAG Activation Code

Figure 5-12 Dormant to JTAG state diagram

Each of the bit-strings shown in Figure 5-12 are received MSB first. The transition out of state G2 requires a reset of the JTAG TAP, but otherwise returns to dormant state. For more information on this sequence, contact ARM.

Note

ADIv5 does not define any other activation codes, but also does not prohibit an implementation from recognizing other activation codes for compatibility with other standards. Implementations can also use alternative selection alert mechanisms. Debuggers can generate multiple selection alert sequences to alert multiple devices, and then use the common activation codes to select which devices to activate.

5.4 Restriction on switching

A debugger must not mix JTAG-DP and SW-DP reads and writes of DAP registers in a single debug session. A single debug session is defined as from when a debugger connection is made with the system in a reset state through to the debugger connection being broken. At the start of a debug session, the state of the target is essentially UNKNOWN.

Attempting to mix JTAG-DP and SW-DP reads and writes of DAP registers while any component of the DAP is active can have unpredictable results.

A powerup reset cycle might be required to reset the DAP before a change in active Data Link protocol. However, this is not required when switching between the active protocol and dormant state.

Chapter 6
The Access Port (AP)

A Debug Access Port can include multiple Access Ports. ARM provides two AP definitions, but other designers might implement additional APs. This chapter gives an overview of Access Ports, and describes the features that must be implemented in every Access Port.

This chapter contains the following sections:

- *Overview of Access Ports (APs)* on page 6-122.
- *Selecting and accessing an AP* on page 6-123.
- *The Programmers' Model for Access Port (AP) registers* on page 6-124.

——— **Note** ———

The following chapters give more information about APs:

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

———————————

6.1 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).

A DAP might have multiple Access Ports, and these can be a mixture of types.

Note

- This Architecture Specification also permits a DAP to include additional Access Port types. A debugger must be able to recognize any AP, and must ignore any AP that it does not understand.
- If a DAP 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 a DAP 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, to any future AP implementations by ARM, and to any Access Ports that might be implemented by any third party.

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 *The Programmers' Model for Access Port (AP) registers* on page 6-124.

It is also required that any Access Port supports accesses from the implemented Debug Port, as described in *Using the Access Port to access debug resources* on page 1-27. A summary of how to access an AP is given in *Selecting and accessing an AP* on page 6-123.

There are no other requirements for APs in the ADIv5 specifications. All features provided by an AP can be IMPLEMENTATION DEFINED.

6.2 Selecting and accessing an AP

A full description of how APs are selected and accessed is given in *Using the Access Port to access debug resources* on page 1-27. 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 SELECT 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 AP access.
- 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 *DPACC and APACC, the JTAG-DP DP Access register and AP Access register* on page 3-81.

For more information about APACC accesses from a SW-DP, see *Serial Wire Debug protocol operation* on page 4-93.

APACC accesses to a MEM-AP are shown in Figure 7-1 on page 7-129.

APACC accesses to a JTAG-AP are shown in Figure 8-1 on page 8-162.

6.2.1 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, the DAP can put an AP access into a pending state, and the access does not have to complete within a fixed number of cycles. This is important because 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 7-134, for stalling accesses to a MEM-AP.
- *Stalling accesses* on page 8-167, for stalling accesses to a JTAG-AP.

6.3 The Programmers' Model for Access Port (AP) registers

An *Access Port* (AP) implements 32-bit registers. These are mapped into a 256-byte address space. *Using the Access Port to access debug resources* on page 1-27 summarizes the programmers' model for accessing these registers.

Additional information about the AP programmers' model is given in the chapters that describe the two types of AP:

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

ADIv5 requires every AP to implement an AP Identification Register, IDR, at offset `0xFC`. This is the last register in the AP register space, and is described in *IDR, Identification Register*.

The IDR is the only register that must be implemented by all Access Ports. This is summarized in Table 6-1.

Table 6-1 Summary of the common Access Port (AP) register

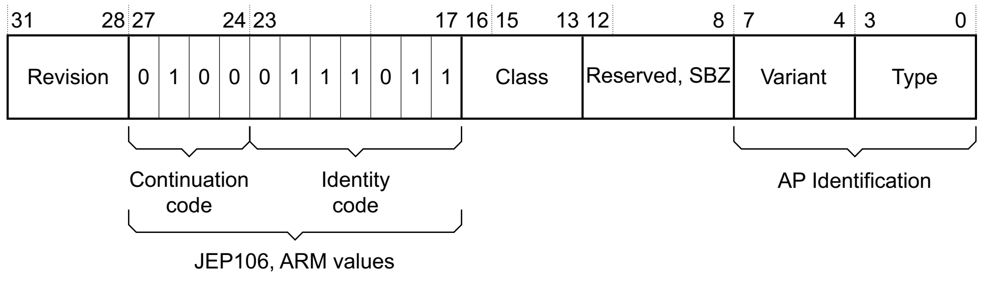
Address	Name	Access	Description	Reset value
`0xFC`	IDR	RO	AP Identification Register	IMPLEMENTATION DEFINED

See *Reserved addresses* on page 6-126 for the requirements for reserved registers.

6.3.1 IDR, Identification Register

DP architecture	The IDR is defined and implemented in DPv0, DPv1, and DPv2.
Purpose	The Identification Register identifies the Access Port. An IDR value of zero indicates that there is no AP present.
Attributes	The IDR is: • A read-only register. • Accessed at offset `0xFC` of the AP register space.

The IDR bit assignments are:



Revision, bits[31:28]

Starts at `0x0` for the first implementation of an AP design, and increments by 1 on each major or minor revision of the design. Major design revisions introduce functionality changes, minor revisions are bug fixes.

JEP106 continuation code, bits[27:24]

Code that identifies the *designer* of the AP. See *JEP106 identity and continuation codes* on page 6-125.

For an AP designed by ARM this field has the value `0x4`.

JEP106 identity code, bits[23:17]

Code that identifies the *designer* of the AP. See *JEP106 identity and continuation codes* on page 6-125.

For an AP designed by ARM this field has the value `0x3B`.

Class, bits[16:13]

Defines the class of AP. An AP belongs to a class if it follows a programmers' model defined as part of the ADIv5 specification or extensions to it. This bit field can have the following values:

`0b0000`	No defined class.
`0b1000`	Memory Access Port. See Chapter 7 *The Memory Access Port (MEM-AP)*.

Bits[12:8] Reserved, SBZ. This field is reserved for future ID register fields. If a debugger reads a non-zero value in this field, it must treat the AP as unidentifiable.

AP identification, bits[7:0]

Access Port Identification. This bit 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 this field is subdivided as:

Variant, bits[7:4]	Identifies different AP implementations of the same Type.
Type, bits[3:0]	Indicates the type of bus, or other connection, that connects to the AP. See *ARM AP Identification types*.

JEP106 identity and continuation codes

The JEP106 codes are assigned by JEDEC and identify the *manufacturer* of a device. However, in the AP Identification register they identify the *designer* of the AP. The following sections describe registers that include JEP106 codes. Each register description clarifies the meaning of its use of the JEP106 codes:

- *IDCODE, the JTAG TAP ID register* on page 3-79.
- *DPIDR, Debug Port Identification Register* on page 2-54.
- *TARGETID, Target Identification register* on page 2-60.
- *The Peripheral ID Registers* on page 9-191.

An implementer of an ARM MEM-AP or JTAG-AP must not change the AP Identification Register values from those given in *IDR, Identification Register* on page 6-124.

Note

- For backwards compatibility, debuggers must treat a JEP106 field of zero as indicating an AP designed by ARM. 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 JEP106 code and include Revision and Class fields.

ARM AP Identification types

Table 6-2 lists the possible values of the Type field for an AP designed by ARM. It also shows the value of the Class bit that corresponds to each Type value.

Table 6-2 ADIv5 AP Identification types for an AP designed by ARM

Type[a]	Connection to AP	Class[b]	Notes
`0x0`	JTAG connection	`0b0000`	Not used by MEM-AP, indicates JTAG-AP. Variant field, bits [7:4] of IDR, must be non-zero.
`0x1`	AMBA AHB bus	`0b1000`	-

Table 6-2 ADIv5 AP Identification types for an AP designed by ARM (continued)

Type[a]	Connection to AP	Class[b]	Notes
`0x2`	AMBA APB2 or APB3 bus	`0b1000`	-
`0x4`	AMBA AXI3 or AXI4 bus, with optional ACE-Lite support	`0b1000`	Not defined in issue A of this document.
Other[c]	Reserved	-	-

a. Bits [3:0] of the IDR.

b. Bits [16:13] of the IDR.

c. Any value other than `0x0`, `0x1`, `0x2`, or `0x4`.

6.3.2 Reserved addresses

For all APs, reserved registers must be RES0.

This applies to all APs, including any implemented by companies other than ARM.

Chapter 7
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.

This chapter contains the following sections:

- *About the function of a Memory Access Port (MEM-AP)* on page 7-128.
- *MEM-AP functions* on page 7-132.
- *Implementing a MEM-AP* on page 7-142.
- *MEM-AP examples of pushed-verify and pushed-compare* on page 7-144.
- *MEM-AP register summary* on page 7-146.
- *MEM-AP register descriptions* on page 7-147.

Note

Chapter 6 *The Access Port (AP)* gives additional information about APs.

7.1 About 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.

7.1.1 The programmers' model for debug register access

The programmers' model for debug registers is a memory map. Although use of a memory bus system is not required, this abstraction enables the same programming model to be used for accessing debug registers and system memory. With this model, the debug registers might be implemented as a peripheral within the system memory space.

The debug registers in a debug component occupy one or more 4KB blocks of address space, and a system might contain several such debug components.

Although the architecture specifications permits a debug component to implement multiple 4KB blocks, most components implement a single block.

Note

Although a component might occupy only 4KB of address space, ARM recommends that the base address of each component is aligned to the largest translation granule supported by any processor that can access the component. For an ARMv8 processor, this might be 64KB.

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.

Chapter 10 *ROM Tables* describes ROM tables.

7.1.2 Selecting and accessing the MEM-AP

Figure 7-1 on page 7-129 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 6-123.

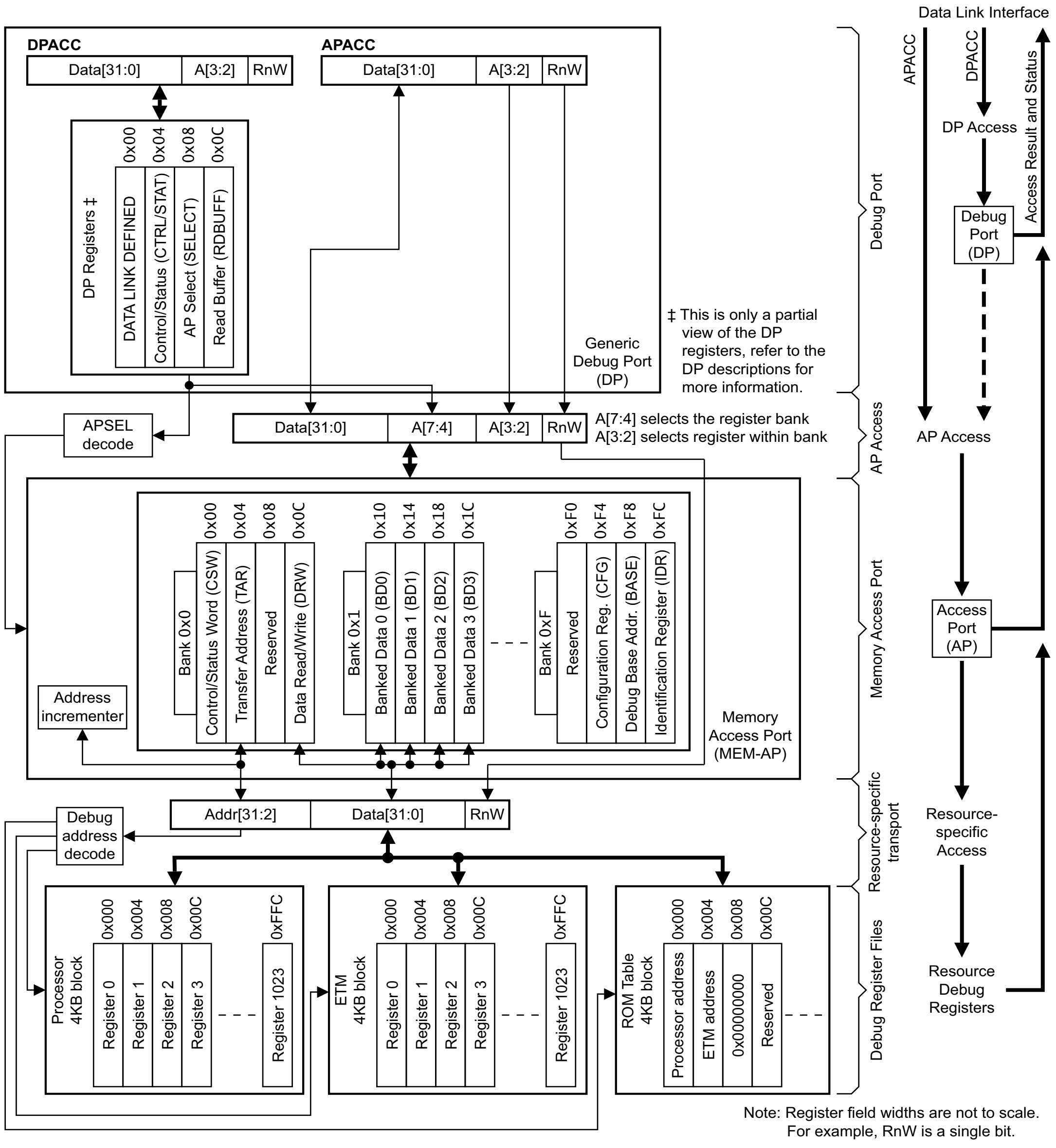


Figure 7-1 MEM-AP connecting the DP to debug components

7.1.3 The MEM-AP registers

The MEM-AP registers, and the memory map of the MEM-AP, are described in detail in *MEM-AP register descriptions* on page 7-147. 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 7-1. To summarize, 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, BD0, 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 (TAR[31:4] << 4) in memory.
- Accessing BD1 accesses ((TAR[31:4] << 4) + `0x4`) in memory.
- Accessing BD2 accesses ((TAR[31:4] << 4) + `0x8`) in memory.
- Accessing BD3 accesses ((TAR[31:4] << 4) + `0xC`) in memory.

The value in TAR[3:0] is ignored in constructing the access address:

- Bits[3:2] of the access address depend solely on which of the four banked data registers is being accessed.
- Bits[1:0] of the access is always zero.

Configuration register, CFG

The CFG register hold information about the configuration of the MEM-AP.

Debug Base Address register, BASE

The BASE register is a pointer into the connected memory system. It points to one of:

- The start of a set of debug registers for the single connected debug component.
- The start of a ROM table that describes the connected debug components.

Identification Register, IDR

The IDR identifies the MEM-AP.

Note

This brief summary of the MEM-AP registers does not include cross-references to the detailed register descriptions. For more information about these registers, see *MEM-AP register descriptions* on page 7-147.

7.1.4 MEM-AP register accesses and memory accesses

Note

In this section, an access to the debug resources is described as a memory access.

This 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 are summarized in the following sections:

- *Accesses that do not initiate a memory access*.
- *Accesses that initiate a memory access* on page 7-131.
- *Accesses that support pushed transactions and the transaction counter* on page 7-131.

Accesses that do not initiate a memory access

APACC accesses to the following MEM-AP registers do not cause a memory access:

- The Control/Status Word register, CSW.
- The Transfer Address Register, TAR.
- The Configuration register, CFG.

- The Debug Base Address register, BASE.
- The Identification Register, IDR.

Accesses that initiate a memory access

This section introduces the APACC accesses to MEM-AP registers that initiate one or more memory accesses. These APACC accesses are:

- Accesses to the DRW register. A memory access is initiated, using the address held in the TAR.
- Accesses to one of the Banked Data Registers, BD0.

  The address used for the memory access depends on which Banked Data Register is accessed.

However, if the MEM-AP implementation includes the Large Data Extension, and CSW.Size specifies a transfer size that is larger than word, some DRW and BD0 accesses do not initiate a memory access, see *Use of the DRW register with the MEM-AP Large Data Extension* on page 7-156 and *Use of the BD registers with the MEM-AP Large Data Extension* on page 7-152.

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 2-38.
- If packed transfers are supported and enabled and the transfer size is smaller than word. See *Packed transfers* on page 7-138.

For more information see *Packed transfers and the transaction counter* on page 7-139.

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.

Accesses that support pushed transactions and the transaction counter

A MEM-AP supports pushed transactions and sequences of transactions to the following registers only:

- DRW, Data Read/Write register.
- Banked Data registers 0-3, see *BD0-BD3, Banked Data registers 0-3* on page 7-150.

For more information see:

- *Pushed-compare and pushed-verify operations* on page 2-36.
- *The transaction counter* on page 2-38.

7.2 MEM-AP functions

This section describes the functions of a MEM-AP. These functions are controlled by the MEM-AP registers, as described in *MEM-AP register descriptions* on page 7-147.

The following sections describe functions that a MEM-AP must support:

- *Enabling access to the connected debug device or memory system*.
- *Auto-incrementing the Transfer Address Register (TAR)*.
- *Stalling accesses* on page 7-134.
- *Response to debug component errors* on page 7-135.

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 7-136.
- *Byte lanes* on page 7-137.
- *Packed transfers* on page 7-138.
- *Slave memory port and software access control* on page 7-140.
- *Implementing a MEM-AP* on page 7-142.

——— **Note** ———

Some of the IMPLEMENTATION DEFINED functions are inter-dependent. These dependencies are summarized in *MEM-AP implementation requirements* on page 7-142.

7.2.1 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). **DEVICEEN** is normally tied HIGH, so that it is asserted even when the Debug Enable signal, **DBGEN**, 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 CSW.DeviceEn bit.

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 CTRL/STAT.STICKYERR bit to be set to 1.
- 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.

7.2.2 Auto-incrementing the Transfer Address Register (TAR)

As indicated in *The MEM-AP registers* on page 7-129, 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 determines 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 7-131.

Software 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 CSW.AddrInc field.

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 7-1.

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 7-1 Summary of AddrInc field values

AddrInc value	Description	Support required?
`0b00`	Auto-increment off	Always.
`0b01`	Increment single	Always.
`0b10`	Increment packed	If Packed transfers are supported. See *Packed transfers* on page 7-138. If Packed transfers are not supported, the value `0b10` selects the *Auto-increment off* mode and reading the AddrInc value returns `0b00`.
`0b11`	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 7-136.

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.

Increment packed

If packed transfers are supported, setting AddrInc to `0b10`, Increment packed, enables packed transfer operation. Packed transfers are described in more detail in *Packed transfers* on page 7-138. In brief, with packed transfers multiple halfword 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 7-138, 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.

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 the TAR is set to `0x14A4`, and the access size is word, successive accesses to the DRW increment TAR to `0x14A8`, `0x14A`, and in steps of 4 bytes 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.

7.2.3 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, 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.

7.2.4 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 2-34.

7.2.5 Variable access size for memory accesses

It is IMPLEMENTATION DEFINED whether an MEM-AP supports memory access sizes other than word (32-bit).

If the MEM-AP Large Data Extension is not supported, then when a MEM-AP implementation supports different sized accesses, it *must* support word, halfword and byte accesses.

Note

The ARM Debug Interface specification does not require a MEM-AP to support accesses other than word. However if a MEM-AP can access other memory, such as system memory, ARM recommends that it supports other access sizes.

For more information see *MEM-AP implementation requirements* on page 7-142.

The access size is controlled by the CSW.Size field. Table 7-2 shows the access size options.

Table 7-2 Size field values when the MEM-AP supports different access sizes

Size value, CSW.Size	Memory access size	Support required?
`0b000`	Byte (8-bits)	No
`0b001`	Halfword (16-bits)	No
`0b010`	Word (32-bits)	Yes[a]
`0b011`[b]	Doubleword (64-bits)	No
`0b100`[b]	128-bits	No
`0b101`[b]	256-bits	No
`0b110` - `0b111`	Reserved	-

a. On a MEM-AP implementation that does not support access sizes other than word, the Size field is read-only, and always returns the value `0b010`.

b. Supported by the MEM-AP Large Data Extension, see *MEM-AP Large Data Extension* on page 7-143. If the extension is not implemented, this value is reserved.

When a Size smaller than word is used, the resulting data access is returned in byte lanes. See *Byte lanes* on page 7-137 for more information.

Caution

Behavior is UNPREDICTABLE if a Banked Data Register is accessed with the Size field set to any size other than word or doubleword.

7.2.6 Byte lanes

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 CSW.Size field, see *Variable access size for memory accesses* on page 7-136.
- The bottom two bits of the TAR, TAR[1:0].

If supported, packed transfers also use byte laning for byte and halfword transfers. This is described in *Packed transfers* on page 7-138.

Table 7-3 shows how the DRW data is byte-laned.

Table 7-3 Byte-laning of memory accesses from the DRW

CSW[2:0], Size	TAR[1:0]	Access data
`0b000`, byte	`0b00`	DRW[7:0]
	`0b01`	DRW[15:8]
	`0b10`	DRW[23:16]
	`0b11`	DRW[31:24]
`0b001`, halfword	`0b00`	DRW[15:0]
	`0b10`	DRW[31:16]
	`0bX1`	IMPLEMENTATION DEFINED
`0b010`, word	`0b00`	DRW[31:0]
	`0b1X`, `0bX1`	IMPLEMENTATION DEFINED

The IMPLEMENTATION DEFINED behavior shown in Table 7-3 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 CTRL/STAT.STICKYERR bit is set to 1.
- 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 7-138. 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.

Big-endian support

The byte-laning described in this section has the least significant byte of DRW or BD0-BD3 containing data from the lowest address, and so might be described as little-endian.

Previous versions of this manual described a variant of the MEM-AP which supported an alternative form of byte-laning where the most significant byte of DRW or BD0-BD3 contained the byte from the lowest address, described as big-endian. Bit[0] of the CFG register described whether the MEM-AP was little-endian or big-endian. ADIv5.2 obsoletes such implementations.

If the target uses a big-endian memory arrangement, the external debugger must manipulate the values it passes through the MEM-AP accordingly.

7.2.7 Packed transfers

Whether a MEM-AP supports packed transfers is IMPLEMENTATION DEFINED. If packed transfers are supported they are enabled by setting the auto address increment field, CSW.AddrInc, to `0b10`, Increment packed. See *Auto-incrementing the Transfer Address Register (TAR)* on page 7-132.

When packed transfers are enabled, each access to the DRW results in one of the following actions, depending on the value of the CSW.Size field, see *Variable access size for memory accesses* on page 7-136:

- When CSW.Size = `0b010`, word, there is a single word (32-bit) access.
- When CSW.Size = `0b001`, halfword, there are two halfword (16-bit) accesses.
- When CSW.Size = `0b000`, byte, there are four byte (8-bit) accesses.

Use of packed transfers with CSW.Size set to a transfer size larger than word is UNPREDICTABLE.

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 7-3 on page 7-137. This is shown in the following examples:

- Example 7-1, *Halfword packed write operation*.
- Example 7-2, *Byte packed write operation*.
- Example 7-3 on page 7-139, *Halfword packed read operation* on page 7-139.

Example 7-1 Halfword packed write operation

This example describes a single word (32-bit) write access to the DRW on a MEM-AP with the following settings:

- Size, CSW[2:0] = `0b001`, to give halfword (16-bit) memory accesses.
- AddrInc, CSW[5:4] = `0b10`, to give packed transfer operation.
- TAR[31:0] = `0x00000000`, to define the base address of the access.

Two write transfers are made. The *halfword* entries in Table 7-3 on page 7-137 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 7-2 Byte packed write operation

This example describes a single word (32-bit) write access to the DRW on a MEM-AP with the following settings:

- Size, CSW[2:0] = `0b000`, to give byte (8-bit) memory accesses
- AddrInc, CSW[5:4] = `0b10`, to give packed transfer operation
- TAR[31:0] = `0x00000002`, to define the base address of the access.

Four write transfers are made. The *byte* entries in Table 7-3 on page 7-137 define the byte-laning for these accesses. The accesses are made in the following order:

- TAR[1:0]=`0b10`, 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`.
- TAR[1:0]=`0b11`, 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`.
- TAR[1:0]=`0b00`, 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`.
- TAR[1:0]=`0b01`, 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 7-3 Halfword packed read operation

This example describes a single word (32-bit) read access to the DRW on a MEM-AP with the following settings:

- Size, CSW[2:0]= `0b001`, to give halfword (16-bit) memory accesses.
- AddrInc, CSW[5:4] = `0b10`, to give packed transfer operation.
- TAR[31:0] = `0x00000002`, to define the base address of the access.

Two read transfers are made. The little-endian *halfword* entries in Table 7-3 on page 7-137 define the byte-laning for these accesses. The accesses are made in the following order:

- 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`.
- 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 2-36 for a description of pushed comparisons.

Packed transfers and the transaction counter

The optional Debug Port transaction counter, described in *The transaction counter* on page 2-38, 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 transfers, if 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.

7.2.8 Slave memory port and software access control

The CSW register can include a debug software access enable flag, CSW.DbgSwEnable. Whether this bit is supported is IMPLEMENTATION DEFINED. The following subsections describe the possible alternative uses of this bit:

- *Using DbgSwEnable to control a slave memory port.*
- *Use of DbgSwEnable to control software access to debug resources.*

If neither of these features is implemented then CSW.DbgSwEnable is RAZ.

If implemented, then CSW.DbgSwEnable must be ignored and treated as one if the MEM-AP device is disabled.

Caution

If CSW.DbgSwEnable is implemented, then setting it to zero can cause software executing on the target to generate unexpected errors, which is likely to cause it to malfunction.

ARM strongly recommends not setting CSW.DbgSwEnable to zero.

A MEM-AP can include a slave memory port, that an external bus master can use 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. A debug component can use the value of this address bit to distinguish between slave memory port accesses and DAP accesses.

The BASEADDR field on page 7-149 gives more information about MEM-AP memory addressing.

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 7-4.

Table 7-4 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.

Use of DbgSwEnable to control software access to debug resources

The DbgSwEnable bit can drive a system-level signal, **DBGSWENABLE**. This signal gates software access to debug resources. For example, in a processor that complies with the ARMv7 Debug Architecture, some CP14 registers are not accessible when **DBGSWENABLE** is LOW. For more information see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Table 7-5 shows the behavior of the DbgSwEnable bit when **DBGSWENABLE** is implemented.

Table 7-5 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.

7.3 Implementing a MEM-AP

This section gives information about the implementation of a MEM-AP and contains the following:

- *IMPLEMENTATION DEFINED features of an MEM-AP implementation*.
- *MEM-AP implementation requirements*.
- *MEM-AP Extensions* on page 7-143.

7.3.1 IMPLEMENTATION DEFINED features of an MEM-AP implementation

There are a number of IMPLEMENTATION DEFINED features of an MEM-AP implementation:

- Whether the MEM-AP supports data bus access sizes other than word, including whether the implementation includes the extension summarized in *MEM-AP Large Data Extension* on page 7-143.
- Whether the MEM-AP supports packed transfers.
- Whether the MEM-AP supports addresses larger than 32-bit, meaning whether the implementation includes the extension summarized in *MEM-AP Large Physical Address Extension* on page 7-143.
- Whether the MEM-AP supports barrier operations, meaning whether the implementation includes the extension summarized in *MEM-AP Barrier Operation Extension* on page 7-143.

Also, it is IMPLEMENTATION DEFINED whether the features described in *Slave memory port and software access control* on page 7-140 are supported.

These implementation choices affect the following register fields:

- CSW.{DbgSwEnable, Mode, AddrInc, Size}.
- CFG.{LD, LA, BE}.

In addition, the CSW register can include the following optional fields that are not described elsewhere in this chapter:

CSW.{Prot, Type}, bits[30:24, 15:12]

These fields can be implemented to provide a bus access control mechanism. If implemented, it enables a debugger to specify flags for a memory access. The permitted values and their significance are IMPLEMENTATION DEFINED, because they relate to the underlying bus architecture. ARM strongly 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 must select privileged accesses.
- If the bus supports code and data accesses, the reset value must select data accesses.
- If the bus supports both Secure and Non-secure address spaces, CSW.{Prot,Type} must reset to select Non-secure addresses.

CSW.SPIDEN, bit[23]

This field can be implemented to indicate whether the MEM-AP can generate secure accesses.

7.3.2 MEM-AP implementation requirements

The descriptions given in the section *MEM-AP functions* on page 7-132 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), halfword (16-bit), and byte (8-bit) accesses, and optionally support larger access sizes.

No other combinations of supported access sizes are permitted. For more information see *Variable access size for memory accesses* on page 7-136.

- If access sizes smaller than word are not supported then packed transfers are not supported. Otherwise, it is IMPLEMENTATION DEFINED whether packed transfers are supported. For more information see *Packed transfers* on page 7-138.
- It is IMPLEMENTATION DEFINED whether access sizes larger than 32-bit are supported. If larger access sizes are not supported, CFG.LD is RAZ. For more information, see *MEM-AP Large Data Extension*.
- It is IMPLEMENTATION DEFINED whether addresses larger than 32-bit are supported. If larger addresses are not supported, CFG.LA is RAZ. For more information, see *MEM-AP Large Physical Address Extension*.
- It is IMPLEMENTATION DEFINED whether barrier operations are supported. If barrier operations are not supported, CSW.Mode is RAZ. For more information, see *MEM-AP Barrier Operation Extension*.

7.3.3 MEM-AP Extensions

The following subsections summarize the effects of the optional MEM-AP Extensions.

MEM-AP Large Physical Address Extension

The MEM-AP Large Physical Address Extension provides address spaces of up to 64-bits.

Implementing this extension changes the format of the following MEM-AP registers:

- *BASE, Debug Base Address register* on page 7-147.
- *BD0-BD3, Banked Data registers 0-3* on page 7-150.
- *CFG, Configuration register* on page 7-153.
- *CSW, Control/Status Word register* on page 7-154.
- *DRW, Data Read/Write register* on page 7-156.
- *TAR, Transfer Address Register* on page 7-158.

MEM-AP Large Data Extension

The MEM-AP Large Data Extension can support 32-bit, 64-bit, 128-bit, or 256-bit accesses, in addition to optional 8-bit and 16-bit accesses.

The following registers have different formats to support this extension:

- *CSW, Control/Status Word register* on page 7-154.
- *DRW, Data Read/Write register* on page 7-156.
- *BD0-BD3, Banked Data registers 0-3* on page 7-150.
- *CFG, Configuration register* on page 7-153.

Although the extension can support 64-bit, 128-bit, and 256-bit accesses, it does not require an implementation to support all of these access sizes. If the CSW.Size field is written with a value corresponding to a size that is not supported, or with a reserved value:

- A read of the field returns a value corresponding to a supported size.
- MEM-AP behavior corresponds to the value returned by the read of the CSW.Size field.

MEM-AP Barrier Operation Extension

The MEM-AP Barrier Operation Extension provides support for barrier operations. If the bus supports a weak memory ordering model, then barrier operations must create order.

The following registers are new or have different formats to support this extension:

- *CSW, Control/Status Word register* on page 7-154.
- *MBT, Memory Barrier Transfer register* on page 7-158.

7.4 MEM-AP examples of pushed-verify and pushed-compare

A Debug Port (DP) might support pushed operations, as described in *Pushed-compare and pushed-verify operations* on page 2-36. 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 a DP that is connected to a MEM-AP.

7.4.1 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) register is set up to increment the *Transfer Address Register* (TAR) after each access.
- Write to the *Transfer Address Register* (TAR) to indicate the start address of the memory region that is to be verified.
- 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 CTRL/STAT.STICKYCMP bit if the values do not match.

  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.

7.4.2 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) register is set up to increment the TAR after each access.
- Write to the *Transfer Address Register* (TAR) to indicate the start address of the debug register region that is to be searched.
- Repeatedly write the value to be searched for as an AP write transaction to the *Data Read/Write register* (DRW). On each transaction the MEM-AP *reads* the location indicated by the TAR.
  — The value returned is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to 1.
  — Otherwise, 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 7-145 describes how the transaction counter can 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.

7.4.3 Example using the transaction counter for a pushed-find operation on a MEM-AP

The transaction counter can refine the pushed-find search operation described in *Example use of pushed-find operation on a MEM-AP* on page 7-144. 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) register is set up to increment the TAR after each access.
- Write to the *Transfer Address Register* (TAR) to indicate the start address of the debug register region that is to be searched.
- Write to the transaction counter field, CTRL/STAT.TRNCNT to indicate the required number of repeat accesses. This value defines the size of the region to be searched.
- Write the value you are searching for as an AP write to the Data Read/Write register (DRW). The MEM-AP repeatedly *reads* the location indicated by the TAR. The value returned by each read is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set to 1 and the operation completes. Otherwise:
  - — The TAR is incremented.
  - — If the transaction counter is nonzero, it is decremented.

  The operation completes when either the STICKYCMP flag is set to 1 or after the final read when the transaction counter was zero.

7.5 MEM-AP register summary

Table 7-6 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 6 *The Access Port (AP)*.

All of the registers listed in Table 7-6 are required in every MEM-AP implementation.

Table 7-6 Summary of Memory Access Port (MEM-AP) registers

Address[a]	Access	Reset value	Description
`0x00`	RW	See *CSW, Control/Status Word register* on page 7-154.	See *CSW, Control/Status Word register* on page 7-154.
`0x04`	RW	UNKNOWN	See *TAR, Transfer Address Register* on page 7-158.
`0x08`	RW	`0x00000000`	See *TAR, Transfer Address Register* on page 7-158. The entries in this row only apply if the implementation includes the Large Physical Address Extension.[b] Otherwise, this register is reserved, RES0.
`0x0C`	RW	Not applicable.[c]	See *DRW, Data Read/Write register* on page 7-156.
`0x10`	RW	Not applicable.[c]	See *BD0-BD3, Banked Data registers 0-3* on page 7-150.
`0x14`	RW		
`0x18`	RW		
`0x1C`	RW		
`0x20`	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	See *MBT, Memory Barrier Transfer register* on page 7-158. The entries in this row only apply if the implementation includes the Barrier Operation Extension. Otherwise, this register is reserved, RES0.
`0x24` - `0xEC`	RES0	-	Reserved
`0xF0`	RO	IMPLEMENTATION DEFINED	See *BASE, Debug Base Address register* on page 7-147. The entries in this row only apply if the implementation includes the Large Physical Address Extension.[b] Otherwise, this register is reserved, RES0.
`0xF4`	RO	IMPLEMENTATION DEFINED	See *CFG, Configuration register* on page 7-153.
`0xF8`	RO	IMPLEMENTATION DEFINED[d]	See *BASE, Debug Base Address register* on page 7-147.
`0xFC`	RO	IMPLEMENTATION DEFINED[d]	See *IDR, Identification Register* on page 6-124.

a. Bits [1:0] of the register address are always `0b00`.
b. When the MEM-AP implementation includes the Large Physical Address Extension, the location holds the most-significant word of the 64-bit register.
c. These registers cannot be read until the memory access has completed. Therefore they do not have reset values.
d. The register descriptions give details of the reset values of *some* bits of these registers.

Reserved addresses in the register memory map are RES0.

Using the Debug Port to access Access Ports on page 1-24 explains how to access AP registers.

7.6 MEM-AP register descriptions

This section describes each of the required MEM-AP registers. Table 7-6 on page 7-146 shows these registers, and indexes the full register descriptions in this section. The following subsections describe the registers:

- *BASE, Debug Base Address register*.
- *BD0-BD3, Banked Data registers 0-3* on page 7-150.
- *CFG, Configuration register* on page 7-153.
- *CSW, Control/Status Word register* on page 7-154.
- *DRW, Data Read/Write register* on page 7-156.
- *MBT, Memory Barrier Transfer register* on page 7-158.
- *TAR, Transfer Address Register* on page 7-158.

7.6.1 BASE, Debug Base Address register

The BASE register attributes are:

Purpose The BASE register provides an index into the connected memory-mapped resource. This index value points to one of the following:

- The start of a set of debug registers.
- A ROM table that describes the connected debug components.

Configurations

A MEM-AP register. The BASE register is:

- A 64-bit register in a MEM-AP implementation that includes the extension described in *MEM-AP Large Physical Address Extension* on page 7-143.
- Otherwise, a 32-bit register.

If the bus supports both Secure and Non-secure address spaces, the BASE register is defined to be a Non-secure address. Whether the ROM tables are also accessible in the Secure address space is IMPLEMENTATION DEFINED.

CSW.{Prot,Type} must reset to an access type that is suitable for accessing ROM tables.

Attributes The BASE register is:

- A read-only register.
- In a MEM-AP implementation that does not include the Large Physical Address Extension, a 32-bit register at offset `0xF8` in the MEM-AP register space. This means it is the third register in the last register bank of the MEM-AP register space, bank `0xF`.
- In a MEM-AP implementation that includes the Large Physical Address Extension, a 64-bit register in the MEM-AP register space, with:
  — The least-significant word of the register at offset `0xF8`.
  — The most-significant word of the register at offset `0xF0`.

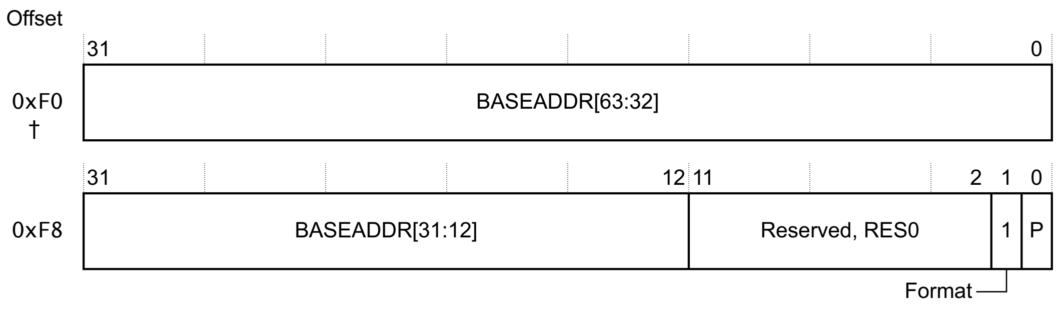
  Caution

  In the 64-bit register implementation, the two words of the register are not contiguous in memory.

Note

Early implementations of Debug Access Ports had different implementations of the Debug Base Address register, described in *Legacy format of the Debug Base Address Register* on page 7-150. The legacy format is a 32-bit register at offset `0xF8`.

The BASE register bit assignments are:



† 64-bit register implementation only. When BASE is a 32-bit register, location is reserved, RES0.

BASEADDR[63:32], bits[31:0] of word at offset 0xF0, 64-bit register only

When the BASE register is implemented as a 64-bit register, bits[63:32] of the address offset, in the memory-mapped resource, of the start of the debug register space or a ROM table address.

For more information, see *The BASEADDR field* on page 7-149.

When the BASE register is implemented as a 32-bit register, this memory location is reserved, RES0.

BASEADDR[31:12], bits[31:12] of word at offset 0xF8

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 *The BASEADDR field* on page 7-149. Bits [11:0] of the address offset are `0x000`.

Bits[11:2] of word at offset 0xF8

Reserved, RES0.

Format, bit[1] of word at offset 0xF8

Base address register format.

This bit is RAO, indicating the ARM Debug Interface v5 format.

Note

This bit is RAZ in one of the legacy Debug Base Address register formats, see *Legacy format of the Debug Base Address Register* on page 7-150.

Entry present, bit[0] of word at offset 0xF8

This field indicates whether a debug entry is present for this MEM-AP:

0	No debug entry present.
1	Debug entry present.

Note

Legacy format of the Debug Base Address Register on page 7-150 includes a description of the legacy format of the BASE register when there is no debug entry present.

When BASE is implemented as a 64-bit register, it can specify any address in a 64-bit *physical address* (PA) space. However:

- ROM tables support only 32-bit signed offset values.
- ARMv7-A processors with the MMU disabled, and ARMv7-R, ARMv6-M and ARMv7-M processors can access only a 32-bit PA space.

Therefore, ARM recommends that all debug components:

- Are located in the bottom 4GB of the PA space.
- Are located in one 2GB half of that address space.

The BASEADDR field

The BASEADDR field defines the *BaseAddress,* in the debug component memory map, of the description of the debug component or components to which the ADIv5 DAP is connected, and:

- Bits[11:0] of the address are zero, and are not held in the BASE register.
- Bits[31:12] of the BASE register word at offset `0xF8` hold bits[31:12] of the address.
- When BASE is implemented as a 64-bit register, bits[31:0] of the BASE register word at offset `0xF0` hold bits[63:32] of this address.

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 1-28, *BaseAddress* 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, *BaseAddress* 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 1-6 on page 1-30, then *BaseAddress* 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 10 *ROM Tables*.

  A simple system, that contains only a single debug component, might be implemented with a separate ROM table, as shown in Figure 1-5 on page 1-29. In this case *BaseAddress* 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 (*BaseAddress* + `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 (*BaseAddress* + `0xFF0`) in the TAR returns the value of this register. For more information about these registers see *The Component ID Registers* on page 9-187.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 specify requirements about the type of component pointed to by the BASE register.

A debugger must handle the following situations as non-fatal errors:

- *BaseAddress* is a faulting location.
- The four words starting at (*BaseAddress* + `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 ARM recommends it continues operating. An example of an implementation that might cause errors of this type is a system with static *BaseAddress* 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

When using 32-bit addressing, meaning BASE is implemented as a 32-bit register, 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 TAR[31]=1.
- System-initiated accesses, if permitted, address the logical segment with TAR[31]=0.

With such an implementation, BASEADDR[31] must be set to 1.

Even where system-initiated accesses are not permitted, ARM recommends that the debug component address space is segmented in this way, and that debugger-initiated accesses have TAR[31]=1.

Other systems might include IMPLEMENTATION DEFINED methods for signaling debugger accesses to system components.

The section *Legacy format of the Debug Base Address Register* is also important for debugger implementation.

Legacy format of the Debug Base Address Register

The legacy format for the Debug Base Address Register is described in the following subsections:

- *Legacy format when no debug entries are present*.
- *Legacy format for specifying BASEADDR*.

The legacy format is defined only for 32-bit addresses and so is not permitted for a MEM-AP that implements the Large Physical Address Extension.

Note

This format must not be used for new ARM Debug Interface designs.

Legacy format when no debug entries are present

The legacy format of the BASE register when there are no debug entries present is:

NOTPRESENT, bits[31:0]

This field has the value `0xFFFFFFFF`, indicating that there are no debug entries.

Legacy format for specifying BASEADDR

When bit[1] of the BASE register is 0, the legacy format of the register holds the base address value. This format is:

BASEADDR, bits[31:12]

Bits[31:12] of the base address. Bits[11:0] of the base address are zero.

For more information, see *The BASEADDR field* on page 7-149.

Bits[11:2] Reserved, RAZ.

FORMAT, bit[1]

RAZ, indicating that this is the legacy 32-bit BASE register format.

Bit[0] Reserved, RAZ.

7.6.2 BD0-BD3, Banked Data registers 0-3

The BD0-BD3 register attributes are:

Purpose 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.

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.

Configurations

MEM-AP registers. The MEM-AP Large Data Extension, described in *MEM-AP Large Data Extension* on page 7-143, extends the behavior of these registers for assesses with CSW.Size set to doubleword.

Attributes The Banked Data registers are:

- At offsets `0x10`, `0x14`, `0x18`, and `0x1C` in the MEM-AP register space.
- Read/write registers.

The bit assignments of a BD*n* register are:

Banked data, bits[31:0]

Data value of the current transfer.

See *Use of the BD registers with the MEM-AP Large Data Extension* on page 7-152 for more information about BD register accesses in an MEM-AP that includes the Large Data Extension.

Auto address incrementing is not performed when a Banked Data register is accessed. The value of CSW.AddrInc has no effect on Banked Data register accesses.

Behavior is UNPREDICTABLE if a Banked Data register is accessed with the CSW.Size field set to any transfer size other value other than word, or doubleword if the MEM-AP Large Data Extension is implemented.

Table 7-7 shows how the four Banked Data registers map onto the address space when CSW.Size specifies word accesses. Bits [1:0] of the memory address are always `0b00`.

Table 7-7 Mapping of Banked Data registers onto memory addresses for word accesses

Register	Offset	Memory address accessed if Large Physical Address Extension is not implemented	Memory address accessed if Large Physical Address Extension is implemented
BD0	`0x10`	TAR[31:4] << 4	TAR[63:4] << 4
BD1	`0x14`	(TAR[31:4] << 4) + `0x4`	(TAR[63:4] << 4) + `0x4`
BD2	`0x18`	(TAR[31:4] << 4) + `0x8`	(TAR[63:4] << 4) + `0x8`
BD3	`0x1C`	(TAR[31:4] << 4) + `0xC`	(TAR[63:4] << 4) + `0xC`

An access to a Banked Data register initiates an access to the memory address shown in Table 7-7. The AP access does not complete until the memory access has completed.

Use of the BD registers with the MEM-AP Large Data Extension

For an MEM-AP implementation that includes the Large Data Extension, Table 7-8 shows how the four Banked Data registers map onto the address space when CSW.Size specifies doubleword accesses. Bits [2:0] of the memory address are always `0b000`.

Table 7-8 Mapping of Banked Data registers onto memory addresses for doubleword accesses

Register	Offset	Memory address accessed if Large Physical Address Extension is not implemented	Memory address accessed if Large Physical Address Extension is implemented	Access Order
BD0	`0x10`	TAR[31:4] << 4	TAR[63:4] << 4	First
BD1	`0x14`	TAR[31:4] << 4	TAR[63:4] << 4	Second
BD2	`0x18`	(TAR[31:4] << 4) + `0x8`	(TAR[63:4] << 4) + `0x8`	First
BD3	`0x1C`	(TAR[31:4] << 4) + `0x8`	(TAR[63:4] << 4) + `0x8`	Second

A debugger must:

- Access both registers of the pair.
- Access the lower-numbered register first.

The lower-numbered register holds the least-significant word. The higher-numbered register holds the most-significant word.

For a read, the first AP access initiates the memory access to the memory address shown in Table 7-8. The AP access does not complete until the memory access completes.

For a write, the second AP access initiates the memory access to the memory address shown in Table 7-8. The AP access does not complete until the memory access completes.

For example, if the Large Physical Address Extension is implemented, to read the doubleword value at (TAR[63:4] << 4), a debugger must:

1. Read BD0, to obtain bits [31:0] of the doubleword.
2. Read BD1, to obtain bits [63:32] of the doubleword.

When CSW.Size specifies doubleword accesses, for a sequence of two accesses to the BD registers:

- The effect of mixing reads and writes in the sequence is UNPREDICTABLE.
- An access to CSW in the middle of the sequence terminates that sequence. The next access to a BD register is the first access of a new sequence.

  If a write sequence is terminated, no memory write is initiated.
- The effect of not accessing the correct register first is UNPREDICTABLE.
- After accessing the first BD register, the effect of accessing any MEM-AP register other than the CSW or the correct second BD register is UNPREDICTABLE. This means the UNPREDICTABLE sequences include:
  - — Accessing BD1 and then accessing BD2.
  - — Two accesses to the same BD register.

7.6.3 CFG, Configuration register

The CFG register attributes are:

Purpose The CFG register provides information about how the MEM-AP implementation is configured. That is, it gives information about the implemented MEM-AP features.

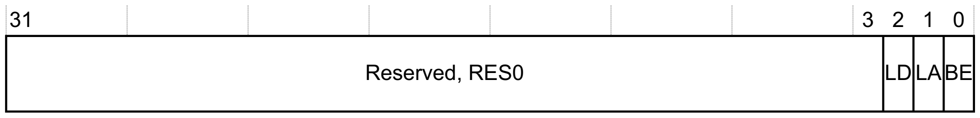
Configurations

A MEM-AP register.

Attributes The CFG register is:

- At offset `0xF4` in the MEM-AP register space.
- A read-only register.

The CFG register bit assignments are:



Bits[31:3] Reserved, RES0.

LD, bit[2] Large data. This bit indicates whether the MEM-AP implementation includes the Large Data Extension, that provides support for data items larger than 32-bits:

0 The implementation does not support data items larger than 32-bits.

1 The implementation includes the Large Data Extension, and therefore supports data items larger than 32-bits.

For more information, see *MEM-AP Large Data Extension* on page 7-143.

Note

Regardless of the value of this bit, the MEM-AP must support word data items, and might support smaller data items. See the description of the CSW.Size field for more information about possible support for smaller data items.

LA, bit[1] Long address. This bit indicates whether the MEM-AP implementation includes the Large Physical Address Extension, that supports physical addresses of more than 32-bits:

0 The implementation support only physical addresses of 32-bits, or fewer bits.

Offsets `0x08` and `0xF0` in the MEM-AP register map are reserved.

1 The implementation supports physical addresses of more than 32-bits. This means:

- The TAR is a 64-bit register, at offsets `0x04` and `0x08` in the MEM-AP register map.
- The BASE register is a 64-bit register, at offsets `0xF8` and `0xF0` in the MEM-AP register map.

For more information, see *MEM-AP Large Physical Address Extension* on page 7-143.

BE, bit[0] Big-endian. ADIv5.2 obsoletes support for big-endian MEM-AP, and this bit must RAZ. For more information, see *Big-endian support* on page 7-137.

7.6.4 CSW, Control/Status Word register

The CSW register attributes are:

Purpose

The CSW register configures and controls accesses through the MEM-AP to or from a connected memory system.

Configurations

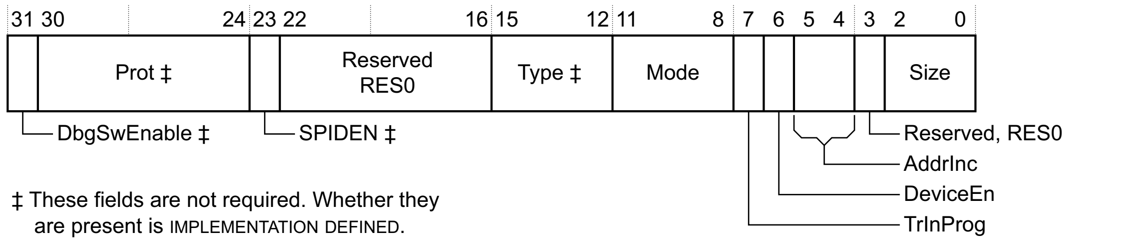
A MEM-AP register.

Attributes

The CSW register is:

- At offset `0x00` in the MEM-AP register space.
- A read/write register, although some bits of the register are read-only.

The CSW register bit assignments are:



DbgSwEnable, bit[31]

Debug software access enable.

DbgSwEnable must be ignored and treated as one if DeviceEn is set to 0.

The use of this flag is IMPLEMENTATION DEFINED, see *Slave memory port and software access control* on page 7-140.

This bit is optional. If not implemented, the bit is RES0.

{Prot, Type}, bits[30:24, 15:12]

Bus access protection control. A debugger can use these fields to specify flags for a debug access. The permitted values and their significance are IMPLEMENTATION DEFINED, because they relate to the underlying bus architecture. For more information, see *Implementing a MEM-AP* on page 7-142.

These fields are optional. If not implemented, the fields are RES0.

SPIDEN, bit[23]

Secure Privileged Debug Enabled. If this bit is implemented, the possible values are:

0	Secure accesses disabled.
1	Secure accesses enabled.

This bit is optional, and read-only. If not implemented, the bit is RES0.

Bits[22:16] Reserved, RES0.

Type, bits[15:12]

See the entry for **{Prot, Type}**.

If not implemented, the field is RES0.

Mode, bits[11:8]

Mode of operation. The possible values of this bit are:

0000	Basic mode.
0001	Barrier support enabled. For more information, see *MEM-AP Barrier Operation Extension* on page 7-143.

All other values are reserved.

The set of modes supported is IMPLEMENTATION DEFINED. If the implementation supports only one mode, this field can be RO.

If this field is RW, the reset value of this field is UNKNOWN.

TrInProg, bit[7]

Transfer in progress. This bit is set to 1 while a transfer is in progress on the connection to the memory system, and is set to 0 while the connection is idle.

After an ABORT operation, debug software can read this bit to check whether the aborted transaction completed.

This bit is read-only.

DeviceEn, bit[6]

Device enabled. This bit is set to 1 when transactions can be issued through the MEM-AP.

This bit shows the value of the **DEVICEEN** signal, that is a control input to the DAP. If **DEVICEEN** is not implemented this bit is RAO.

For more information, see *Enabling access to the connected debug device or memory system* on page 7-132.

This bit is read-only.

AddrInc, bits[5:4]

Address auto-increment and packing mode. This field controls whether the access address increments automatically on read and write data accesses through the DRW register. For more information see *Auto-incrementing the Transfer Address Register (TAR)* on page 7-132.

The reset value of this field is UNKNOWN.

Bit[3] Reserved, RES0.

Size, bits[2:0] Size of access. This field indicates the size of access to perform. It is IMPLEMENTATION DEFINED whether a MEM-AP supports access sizes other than 32-bits:

- If it does then the Size field is RW, and the field indicates the size of the access to perform. See *Variable access size for memory accesses* on page 7-136.

  When this field is RW, its reset value is UNKNOWN.
- If it does not then this field is RO and it reads as `0b010` to indicate that only 32-bit accesses are supported.

If a MEM-AP implementation includes the Large Data Extension then:

- This field must be implemented as RW.
- If a reserved value, or a value corresponding to an unsupported access size, is written to this field, then:
  - — A read of the field returns the value corresponding to a supported size.
  - — MEM-AP behavior corresponds to the value returned by a read of this field.

For more information about the extension, see *MEM-AP Large Data Extension* on page 7-143.

7.6.5 DRW, Data Read/Write register

The DRW register attributes are:

Purpose 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.

Configurations

A MEM-AP register. The MEM-AP Large Data Extension, described in *MEM-AP Large Data Extension* on page 7-143, modifies the behavior of this registers for accesses with CSW.Size set to a value larger than `0b010`.

Attributes The DRW register is:

- At offset `0x0C` in the MEM-AP register space.
- A read/write register.

The DRW register is a 32-bit register:

- In write mode, the DRW holds the value to write for the current transfer to the address specified in the TAR.
- In read mode, the DRW holds the value read in the current transfer from the address specified in the TAR.

The DRW register bit assignments are:

Data, bits[31:0]

Data value of the current transfer.

The size of memory access is controlled by the CSW.Size field. See *Variable access size for memory accesses* on page 7-136.

See *Use of the DRW register with the MEM-AP Large Data Extension* for more information about DRW register accesses in an MEM-AP that includes the Large Data Extension.

If CSW.Size specifies an access size of less than 32-bits then a single access to the DRW might result in multiple memory accesses, depending on the values of CSW.AddrInc. See *Packed transfers* on page 7-138.

After each access the TAR might be incremented depending on the value of CSW.AddrInc. See *Auto-incrementing the Transfer Address Register (TAR)* on page 7-132.

Use of the DRW register with the MEM-AP Large Data Extension

In an MEM-AP implementation that includes the Large Data Extension, the behavior of accesses to the DRW register depends on the memory access size defined by the CSW.Size field, as follows:

8-bit, 16-bit, or 32-bit accesses

Behavior is identical to the behavior described in this section for a MEM-AP implementation that does not include the Large Data Extension.

64-bit, 128-bit, and 256-bit accesses

Software must make multiple accesses to the DRW to perform a single memory access, as Table 7-9 shows. The *Total number* column shows the number of DRW accesses in the required DRW access sequence:

Table 7-9 Number of DRW accesses required for different CSW.Size values

CSW.Size	Memory access size	Total number of DRW accesses required
`0b011`	64-bit	2
`0b100`	128-bit	4
`0b101`	256-bit	8

The behavior depends on whether the access is a read or a write, as follows:

Read The first read of the DRW in a sequence initiates the memory access, and returns the least significant 32-bit word of the required data value. The AP access does not complete until the memory access completes.

Each subsequent DRW read in the sequence, up to the total shown in Table 7-9, returns the next 32-bit word of the required data, but does not initiate another memory access.

Write The first writes of the DRW in a sequence, except the last, specify 32-bit words of the data value to be written, starting with the least-significant word. These writes do not initiate a memory access.

The last write of the DRW specifies the most-significant 32-bit word of the data value, and initiates the memory access. The AP access does not complete until the memory access completes.

As this description indicates, each DRW access sequence:

- Consists of the number of DRW accesses specified in Table 7-9.
- Corresponds to a single memory access.

Within such a DRW access sequence:

- The effect of mixing reads and writes in the sequence is UNPREDICTABLE.
- An access to CSW in the middle of the sequence terminates that sequence. The next access to the DRW register is the first access of a new sequence.

  If a write sequence is terminated, no memory write is initiated.
- After the first DRW access of the sequence, the effect of accessing any MEM-AP register other than the CSW or the DRW is UNPREDICTABLE.

7.6.6 MBT, Memory Barrier Transfer register

The MBT register attributes are:

Purpose The MBT register generates a barrier operation on the bus. If the Barrier Operation Extension is not implemented, this register is reserved, RES0.

Configurations

A MEM-AP register, implemented only if the MEM-AP implementation includes the extension described in *MEM-AP Barrier Operation Extension* on page 7-143.

Usage constraints

If CSW.Mode is set to a value other than `0b0001`, writes to this register are ignored.

Attributes The MBT register is at offset `0x20` in the MEM-AP register space.

Other properties of the register are IMPLEMENTATION DEFINED.

The definition of this register is IMPLEMENTATION DEFINED.

7.6.7 TAR, Transfer Address Register

The TAR attributes are:

Purpose The TAR holds the memory address to be accessed.

Configurations

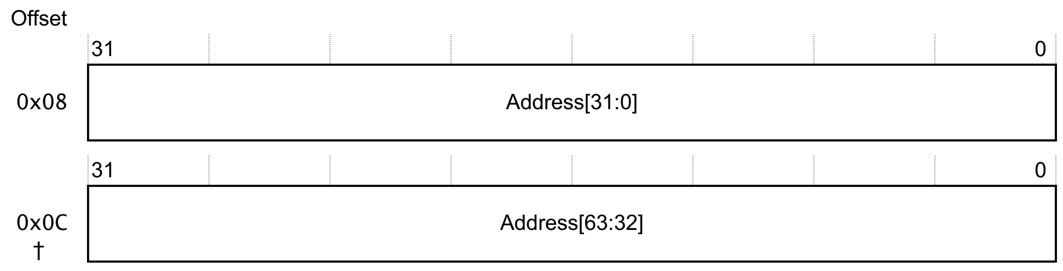
A MEM-AP register. The TAR is:

- A 64-bit register in a MEM-AP implementation that includes the extension described in *MEM-AP Large Physical Address Extension* on page 7-143.
- Otherwise, a 32-bit register.

Attributes The TAR is:

- A RW register.
- In a MEM-AP implementation that does not include the Large Physical Address Extension, a 32-bit register at offset `0x04` in the MEM-AP register space.
  The reset value of this register is UNKNOWN.
- In a MEM-AP implementation that includes the Large Physical Address Extension, a 64-bit register in the MEM-AP register space, with:
  — The least-significant word of the register at offset `0x04`.
  The reset value of this word is UNKNOWN.
  — The most-significant word of the register at offset `0x08`.
  The reset value of this word is `0x00000000`.

The TAR bit assignments are:



† 64-bit register implementation only. When the TAR is a 32-bit register, location is reserved, RES0.

Address[31:0], bits[31:0] of the register word at offset `0x08`

Bits[31:0] of the memory address for the current transfer.

Address[63:32], bits[31:0] of the register word at offset `0x0C`, Large Physical Address Extension

Bits[63:32] of the memory address for the current transfer.

If the MEM-AP implementation does not include the Large Physical Address Extension, this register word is RES0.

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.

Depending on whether the MEM-AP implementation includes the Large Physical Address Extension, TAR is either:

- A 64-bit register, TAR[63:0].
- A 32-bit register, TAR[31:0].

When using the DRW, the TAR 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 RAZ/WI.
- The contents of the TAR can be incremented automatically on a successful DRW access, see *Auto-incrementing the Transfer Address Register (TAR)* on page 7-132.

When accessing memory through *BD0-BD3, Banked Data registers 0-3* on page 7-150, bits [3:0] of the TAR are ignored, and TAR[63:4] or TAR[31:4] specifies the base address of the 16-byte block of memory that can be accessed.

Chapter 8
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.

This chapter contains the following sections:

- *Overview of the JTAG Access Port (JTAG-AP)* on page 8-162.
- *Operation of the JTAG-AP* on page 8-166.
- *The JTAG Engine Byte Command Protocol* on page 8-169.
- *JTAG-AP register summary* on page 8-176.
- *JTAG-AP register descriptions* on page 8-177.

Note

Chapter 6 *The Access Port (AP)* gives additional information about APs.

8.1 Overview of the 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 recommends that only one TAP is connected to each 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.

8.1.1 Selecting and accessing the JTAG-AP

Figure 8-1 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 6-123.

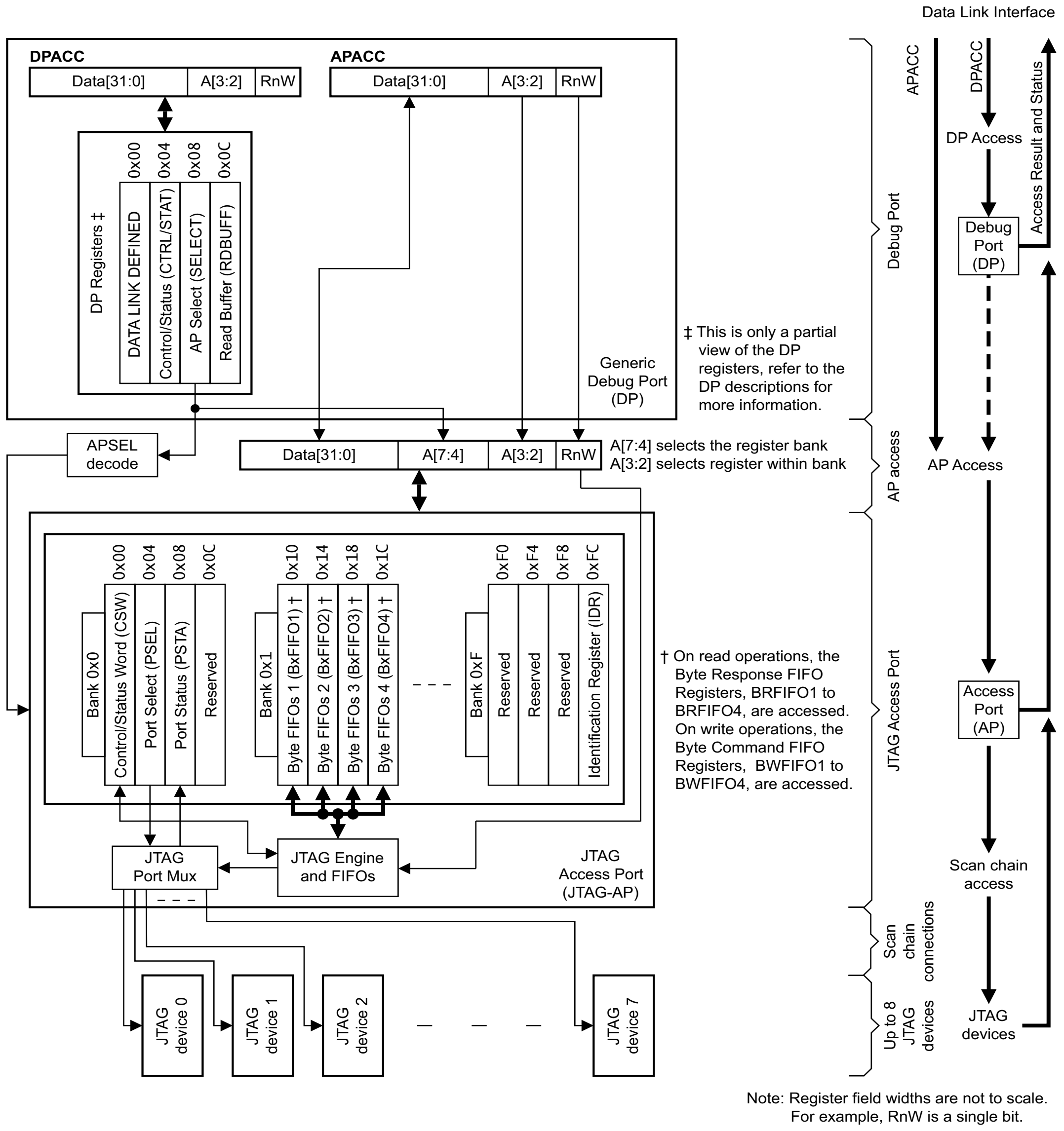

Data Link Interface
DPACC
APACC
Data[31:0]
A[3:2]
RnW
DP Registers ‡
DATA LINK DEFINED 0x00
Control/Status (CTRL/STAT) 0x04
AP Select (SELECT) 0x08
Read Buffer (RDBUFF) 0x0C
Generic Debug Port (DP)
‡ This is only a partial view of the DP registers, refer to the DP descriptions for more information.
APSEL decode
Data[31:0]
A[7:4]
A[3:2]
RnW
A[7:4] selects the register bank
A[3:2] selects register within bank
Bank 0x0
Control/Status Word (CSW) 0x00
Port Select (PSEL) 0x04
Port Status (PSTA) 0x08
Reserved 0x0C
Bank 0x1
Byte FIFOs 1 (BxFIFO1) † 0x10
Byte FIFOs 2 (BxFIFO2) † 0x14
Byte FIFOs 3 (BxFIFO3) † 0x18
Byte FIFOs 4 (BxFIFO4) † 0x1C
Bank 0xF
Reserved 0xF0
Reserved 0xF4
Reserved 0xF8
Identification Register (IDR) 0xFC
† On read operations, the Byte Response FIFO Registers, BRFIFO1 to BRFIFO4, are accessed. On write operations, the Byte Command FIFO Registers, BWFIFO1 to BWFIFO4, are accessed.
JTAG Port Mux
JTAG Engine and FIFOs
JTAG Access Port (JTAG-AP)
JTAG device 0
JTAG device 1
JTAG device 2
JTAG device 7
Debug Port
AP access
JTAG Access Port
Scan chain connections
Up to 8 JTAG devices
APACC
DPACC
DP Access
Debug Port (DP)
Access Result and Status
AP Access
Access Port (AP)
Scan chain access
JTAG devices
Note: Register field widths are not to scale. For example, RnW is a single bit.

Figure 8-1 JTAG-AP connecting the DP to JTAG devices

8.1.2 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 8-2 on page 8-164 shows this JTAG-AP structure.

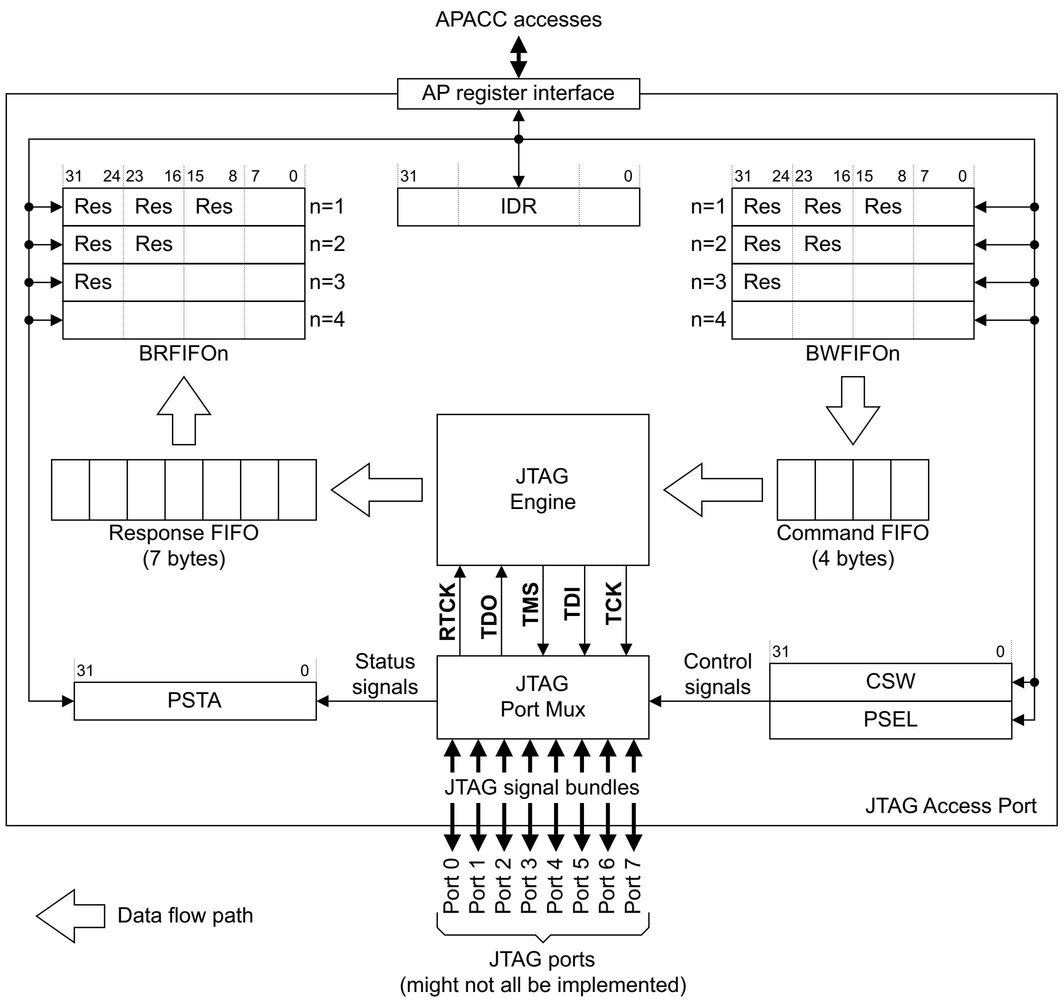


Figure 8-2 Structure of the JTAG Access Port (JTAG-AP)

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.

8.1.3 JTAG port signals

The signal bundle between the JTAG Port Multiplexer and each implemented JTAG port includes:

- The standard IEEE 1149.1 JTAG signals.
- The non-IEEE 1149.1 extension **RTCK** signal.
- Additional port control and status signals.

Table 8-1 gives the full signal list. This list applies to each implemented port.

Table 8-1 JTAG Access Port JTAG port signals

Signal	Direction[a]	Description	Notes
TCK	Out	Test Clock	JTAG IEEE 1149.1 standard signals.
TMS	Out	Test Mode Select	
TDI	Out	Test Data In	
TDO	In	Test Data Out	
TRST*	Out	Test Reset	Active LOW JTAG IEEE 1149.1 standard signal.
RTCK	In	Return Clock	JTAG extension signal, not specified by IEEE 1149.1. If the connected JTAG device has no Return Clock, **RTCK** must be synthesized for the port. Implementation of the **RTCK** signal is IMPLEMENTATION DEFINED.
nSRSTOUT	Out	Subsystem Reset	Active LOW.
SRSTCONNECTED	In	Subsystem Reset Connected	Tie-off configuration signals to the JTAG Port Multiplexer.
PORTCONNECTED	In	Port Connected	
PORTENABLED	In	Port Enabled	Can be deasserted by the JTAG subsystem, for example when the connected TAP powers down.

a. Signal directions are given relative to the JTAG Port Multiplexer in the JTAG-AP.

8.2 Operation of the JTAG-AP

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 8-2 on page 8-164.

The external debugger:

1. Encodes JTAG commands and data into the JTAG Engine Byte Command Protocol, described in *The JTAG Engine Byte Command Protocol* on page 8-169.
2. Writes to the BWFIFOn registers, to transfer the encoded JTAG Engine commands and data to the JTAG Command FIFO.
3. Reads from the BWFIFOn registers. These reads return JTAG TDO data that is collected in response to the encoded JTAG Engine commands.
4. Decodes the actual TDO data from the response data.

The JTAG Engine provides the connection between stages 2 and 3 of this process.

Note

The JTAG-AP can connect to up to 8 JTAG devices. The debugger must write to the PSEL register to select the JTAG port or ports that are connected through the JTAG Port Multiplexer to the JTAG Engine.

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.

For example:

1. The debugger writes two bytes to BWFIFO2, to specify:
   a. A TDI_TDO scan, with 64 bits of TDI data.
   b. The TDO data is to be returned to the debugger.
2. The debugger writes a word to BWFIFO4, containing the first 32-bits of TDI data.
3. The debugger reads a word from BRFIFO4, to obtain the first 32-bits of TDO data.
4. The debugger writes another word to BWFIFO4, with the next 32-bits of TDI data.
5. The debugger reads another word from BRFIFO4, to obtain the next 32-bits of TDO data.

This provides a very efficient encapsulation of the JTAG scan chain. However, as described in *Stalling accesses* on page 8-167, a read of BRFIFOn 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.

Operation of the JTAG-AP is described in more detail in *The JTAG Engine Byte Command Protocol* on page 8-169.

8.2.1 JTAG-AP register accesses and accesses to the JTAG scan chain

APACC accesses to JTAG-AP registers 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 8-2 on page 8-164:
  - — CSW.
  - — PSEL.
  - — PSTA.
  - — IDR.

  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 8-2 on page 8-164:
  - — The Byte Response FIFO Registers, BRFIFO1 to BRFIFO4.
  - — The Byte Command FIFO Registers, BWFIFO1 to BWFIFO4.

  These registers are described in BxFIFO1-BxFIFO4. Using these registers to access the JTAG state machine and JTAG scan chains is described in *The JTAG Engine Byte Command Protocol* on page 8-169.

  Accesses to these registers can be stalled, see *Stalling accesses*.

Stalling accesses

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*.

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 BRFIFO4 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 CSW.RFIFOCNT field. 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 CSW.WFIFOCNT field, and can calculate the number of free bytes in the FIFO. A read of the CSW always completes immediately.

8.2.2 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 8-180.

8.2.3 Pushed transaction and transaction counter support

A JTAG-AP supports pushed transactions and sequences of transactions to the following registers only:

- PSTA.
- BxFIFO1-BxFIFO4.

For more information see:

- *Pushed-compare and pushed-verify operations* on page 2-36.
- *The transaction counter* on page 2-38.

8.3 The JTAG Engine Byte Command Protocol

All JTAG commands, including TMS and TDI data, are written to the JTAG-AP Command FIFO through the interface 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 8-177.

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 8-177.

In the JTAG Engine Byte Command Protocol, all commands are one byte (8-bits). Table 8-2 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 8-2 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	*The encoding of the TMS packet*.
1	0	0	Opcode payload					TDI_TDO	*The encoding of the TDI_TDO packet* on page 8-170.
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	-

8.3.1 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 3-70.

Table 8-3 shows the possible encodings of a TMS packet.

Table 8-3 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 8-3 on page 8-169, 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 8-3. As the diagram shows, this sequence of **TMS** signals takes four **TCK** cycles.

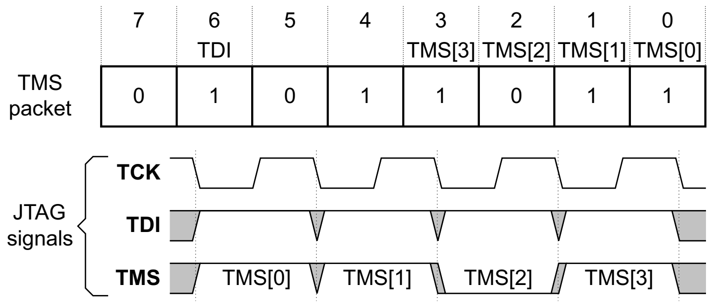


Figure 8-3 TMS packet example with TDI held at 1

To send the sequence of **TMS** bits [1, followed by 0] while keeping **TDI** at 0, the TMS packet is as shown in Figure 8-4. As shown in the diagram, this sequence of **TMS** signals takes two **TCK** cycles.

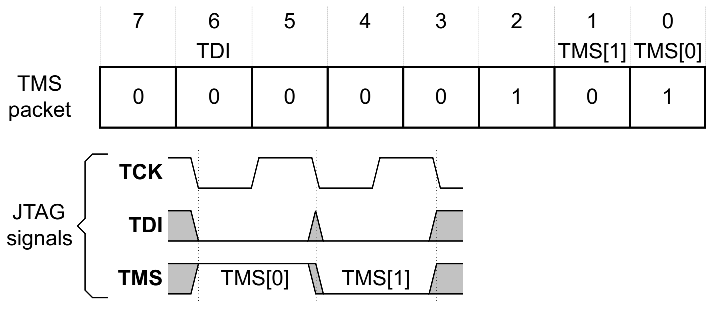


Figure 8-4 TMS packet example with TDI held at 0

8.3.2 The encoding of the TDI_TDO packet

A TDI_TDO packet is a multi-byte packet that is at least two bytes long. It comprises:

- The TDI_TDO opcode byte.
- A second byte, that contains:
  - — For short packets, of fewer than 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 8-5 shows the format of this byte.

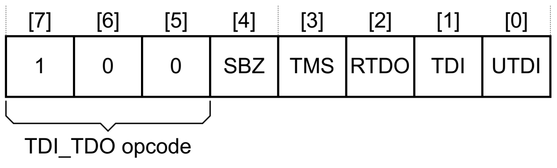


Figure 8-5 TDI_TDO first byte (opcode) format

Bits [3:0] are control bits that define the TDI_TDO sub-type. Table 8-4 describes all the bits of the TDI_TDO packet first byte.

Table 8-4 TDI_TDO first byte (opcode) format

Bit	Name	Value[a]	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	-	**TMS** value to use on the last cycle of the scan: 0 = **TMS** LOW on last cycle 1 = **TMS** HIGH on last cycle. **TMS** is always LOW on all the earlier cycles of the scan.
[2]	RTDO	-	Read **TDO**. This bit determines whether **TDO** values returned during the scan are captured and placed in the Response FIFO: 0 = Do not capture **TDO** 1 = Capture **TDO**. **Caution** Do not set this bit to 1 if more than one JTAG port is selected and enabled. If you do, the **TDO** values captured are UNKNOWN.
[1]	TDI	-	**TDI** value to use throughout the scan if UTDI bit is set to 1: 0 = hold **TDI** signal LOW throughout the scan 1 = hold **TDI** signal HIGH throughout the scan The value of the TDI bit is ignored if UTDI is set to 0.
[0]	UTDI	-	Use TDI bit. This bit determines whether the TDI bits to be used during the scan are supplied in the other bytes of the TDI_TDO packet, or whether the TDI bit, bit [1], specifies the **TDI** signal to use throughout the scan: 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 **TDI** signal to use throughout the scan. When this bit is set to 1, no TDI data is included in the TDI_TDO packet[b].

a. Given for bits that have a fixed value for the TDI_TDO first byte.

b. When the Packed format is used for the second byte of the packet certain bits of that byte are designated as TDI data bits, however the value of these bits is ignored if UTI = 1, see *The TDI_TDO length byte, the second byte of the packet*. 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* on page 8-172.

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 8-172.

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 8-6.

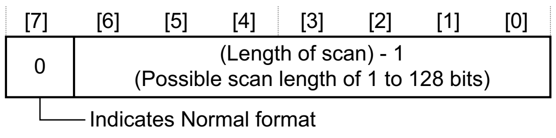


Figure 8-6 TDI_TDO second byte (length byte), normal format

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 8-173 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 8-170 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 8-7 on page 8-173 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 8-7 TDI_TDO second byte (length byte), packed format

The packed format for the TDI_TDO second byte is summarized in Table 8-5.

Table 8-5 TDI_TDO second byte (length byte), packed format

Scan length	Must be zero bits	Data start flag	TDI data for scan[a]
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] = `0b00`	Bit [4] = 1	Bits [3:0]
3 bits	Bits [6:4] = `0b000`	Bit [3] = 1	Bits [2:0]
2 bits	Bits [6:3] = `0b0000`	Bit [2] = 1	Bits [1:0]
1 bit	Bits [6:2] = `0b00000`	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 `0b10001XXX`, referring to Figure 8-7 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 8-3 on page 8-169.

See *The TDI_TDO opcode byte, the first byte of the packet* on page 8-170 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 8-172.
- 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 8-170.

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 8-8 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 8-8 TDI_TDO formatting example. Complete packet for a scan of 21 TCK cycles

In Figure 8-8:

Byte 1, the opcode byte

- Bits[7:5] are the TDI_TDO opcode, `0b100`.
- 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 `0b0010100`, which is 20, meaning the scan length is 21 bits.

Bytes 3 and 4, the first and second data bytes

These bytes contain TDI[15:0], the TDI data for the first 16 cycles of the scan.

Byte 5, the third data byte

This byte contains TDI[20:16], the TDI data for the final five cycles of the scan. Any bits that are not required for TDI data must be set to 0, so bits [7:5] = `0b000`.

8.3.3 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 8-8 shows a TDI_TDO packet with RTDO = 1.

Caution

If more than one JTAG port is selected and enabled, the returned TDO values are UNKNOWN.

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 8-9 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 8-9 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 8-170 for details of the Read TDO (RTDO) bit.

8.4 JTAG-AP register summary

Table 8-6 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 6 *The Access Port (AP)*.

All of the registers listed in Table 8-6 are required in every JTAG-AP implementation.

Table 8-6 Summary of JTAG Access Port (JTAG-AP) registers

Address	Access	Reset value	Description, see	
`0x00`	RW	[a]	CSW	
`0x04`	RW	UNKNOWN	PSEL	
`0x08`	RW	`0x00000000`	PSTA	
`0x0C`	RES0	-	Reserved	
`0x10`	RO	[b]	Read, single entry	BxFIFO1-BxFIFO4
	WO	-	Write, single entry	
`0x14`	RO	[b]	Read, two entries	
	WO	-	Write, two entries	
`0x18`	RO	[b]	Read, three entries	
	WO	-	Write, three entries	
`0x1C`	RO	[b]	Read, four entries	
	WO	-	Write, four entries	
`0x20 - 0xF8`	RES0	-	Reserved	
`0xFC`	RO	[a]	IDR	

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 are RES0.

Using the Debug Port to access Access Ports on page 1-24 explains how to access AP registers.

8.5 JTAG-AP register descriptions

This section describes each of the JTAG-AP registers. Table 8-6 on page 8-176 shows these registers, and indexes the full register descriptions in this section. The following subsections describe these registers:

- *The Byte FIFO registers, BRFIFO1 to BRFIFO4 and BWFIFO1 to BWFIFO4*.
- *CSW, Control/Status Word Register* on page 8-179.
- *PSEL, Port Select register* on page 8-181.
- *PSTA, Port Status Register* on page 8-183.

8.5.1 The Byte FIFO registers, BRFIFO1 to BRFIFO4 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 BRFIFOn register.

The BRFIFOn and BWFIFOn registers are mapped to the same JTAG-AP register addresses, with the BRFIFOn 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 BRFIFO1 and BWFIFO1 registers are at offset `0x10` in the JTAG-AP register space, and transfer a single data byte from or to a FIFO.
- The BRFIFO2 and BWFIFO2 registers are at offset `0x14` in the JTAG-AP register space, and 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 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 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 8-169.

The Byte Read FIFO Registers, BRFIFO1 to BRFIFO4

Figure 8-10 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 8-10 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 CSW.RFIFOCNT field to check the number of response bytes available.

The Byte Write FIFO Registers, BWFIFO1 to BWFIFO4

Figure 8-11 on page 8-178 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 8-11 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 CSW.WFIFOCNT field 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.

8.5.2 CSW, Control/Status Word Register

The CSW register attributes are:

Purpose

The CSW register configures and controls transfers through the JTAG interface.

Configurations

A JTAG-AP register.

Attributes

The CSW register is:

- At offset `0x00` in the JTAG-AP register space.
- A read/write register, although some bits of the register are read-only.
- See the field descriptions for information about the reset value.

The CSW register bit assignments are:

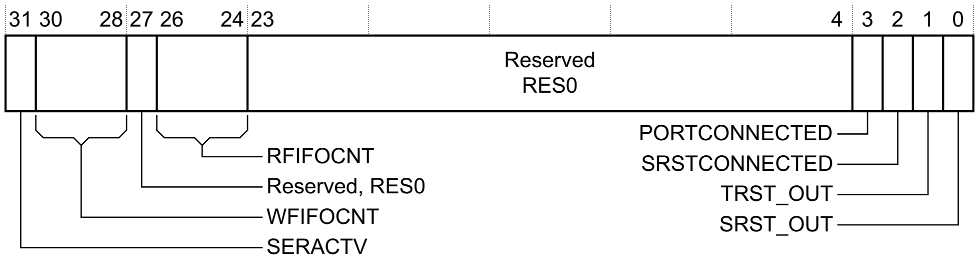


Figure 8-12 JTAG-AP Control/Status Word Register, CSW, bit assignments

SERACTV, bit[31]

JTAG Engine active. This bit is read-only. The possible values of this flag are:

0	JTAG Engine is inactive if WFIFOCNT is 0.
1	JTAG Engine is processing commands from the Command FIFO.

The JTAG Engine is only guaranteed to be inactive if both SERACTV and WFIFOCNT are zero. The reset value of this bit is 0.

WFIFOCNT, bits[30:28]

Command FIFO outstanding byte count. This field is read-only. Gives the number of command bytes held in the Command FIFO that have yet to be processed by the JTAG Engine. The reset value is of this field `0b000`.

Bit[27] Reserved, RES0.

RFIFOCNT, bits[26:24]

Response FIFO outstanding byte count. This field is read-only. Gives the number of bytes of response data available in the Response FIFO. The reset value of this field is `0b000`.

Bits[23:4] Reserved, RES0.

PORTCONNECTED, bit[3]

Selected ports connected. The value of this bit is the logical AND of the **PORTCONNECTED** signals from all currently-selected ports. This bit is read-only. The reset value always depends on the state of the connected signals when the register is read.

SRSTCONNECTED, bit[2]

Selected ports reset connected. The value of this bit is the logical AND of the **SRSTCONNECTED** signals from all currently-selected ports. This bit is read-only. The reset value always depends on the state of the connected signals when the register is read.

TRST_OUT, bit[1]

This bit specifies the signal to drive out on the **TRST*** signal for the currently-selected port or ports. The **TRST*** signal is active LOW, when this bit is set to 1 the **TRST*** output is LOW. This bit is read-only. This bit does not self-reset, it must be cleared to 0 by a software write to this register. The reset value is 0.

0	Deassert **TRST*** HIGH.
1	Assert **TRST*** LOW.

For more information about the use of this bit see *Resetting JTAG devices*.

SRST_OUT, bit[0]

This bit specifies the signal to drive out on the **nSRSTOUT** signal for the currently-selected port or ports. The **nSRSTOUT** signal is active LOW, when this bit is set to 1 the **nSRSTOUT** output is LOW. This bit does not self-reset, it must be cleared to 0 by a software write to this register. The reset value is 0.

0	Deassert **nSRSTOUT** HIGH.
1	Assert **nSRSTOUT** LOW.

Resetting JTAG devices

Although the TRST_OUT bit 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 CSW.TRST_OUT bit, to specify that **TRST*** must be asserted LOW.
2. Drive a sequence of at least five **TMS** = 1 clocks from the JTAG Engine. You can do this by issuing the command `0b00111111` 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 CSW.TRST_OUT bit, 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.

8.5.3 PSEL, Port Select register

The PSEL register attributes are:

Purpose

The PSEL register selects one or more JTAG ports.

Configurations

A JTAG-AP register.

Attributes

The PSEL register is:

- At offset `0x04` in the JTAG-AP register space.
- A read/write register.
- Reset value is UNKNOWN.

The PSEL register bit assignments are:

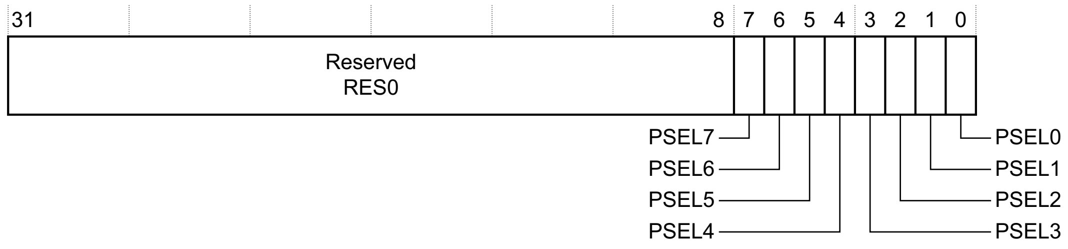


Figure 8-13 JTAG-AP Port Select Register, PSEL, bit assignments

Bits[31:8]

Reserved, RES0.

PSEL*n*, bit[*n*], for *n* = 0 to 7

Select control for JTAG port *n*. If JTAG port *n* is not connected to the JTAG-AP, it is IMPLEMENTATION DEFINED whether PSEL*n* is read/write or RES0. The possible values of this bit are:

0	JTAG port *n* is not selected.
1	JTAG port *n* is selected.

A JTAG port is enabled if all of the following apply:

- The port is connected to the JTAG-AP.
- The PSEL*n* bit for the port is set to 1.
- The **PORTENABLED** signal from the port to the JTAG-AP is asserted HIGH.

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.

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** are UNKNOWN.

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.

8.5.4 PSTA, Port Status Register

The PSTA register attributes are:.

Purpose

The PSTA register indicates whenever a JTAG port that is connected and selected has been disabled, even if the disable is only transient.

Configurations

A JTAG-AP register.

Attributes

The PSTA register is:

- At offset `0x08` in the JTAG-AP register space.
- A read/write register.
- Reset value is 0.

The PSTA register bit assignments are:

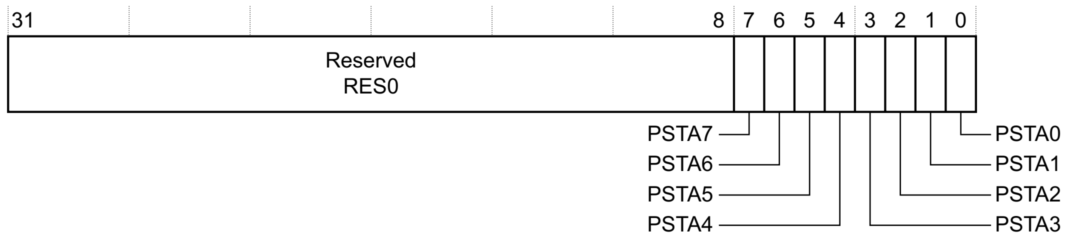


Figure 8-14 JTAG-AP Port Status Register, PSTA, bit assignments

Bits[31:8]

Reserved, RES0.

PSTAn, bit[*n*], for *n* = 0 to 7

Sticky status flag for JTAG port *n*.

This bit is R/W1C.

If JTAG port *n* is connected to the JTAG-AP and PSEL.PSEL*n* is set to 1, then PSTA*n* is set to 1 if the port is disabled. The flag captures and holds the fact that a connected and selected port has been disabled or powered down, even if this is only transient.

Table 8-7 shows the behavior on reads and writes:

Table 8-7 Behavior on reads and writes of PSTA*n*

Value	Meaning on reads	Action on writes
0	Port has not been disabled	No action, write is ignored
1	Port has been disabled	Clear bit to 0

Note

If a port is not connected to the JTAG-AP, its PSTA*n* bit is RAZ.

Chapter 9
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 1-29, and in other illustrations of debug systems.

This chapter contains the following sections:

- *Component and Peripheral ID registers* on page 9-186.
- *The Component ID Registers* on page 9-187.
- *The Peripheral ID Registers* on page 9-191.

Note

Contact ARM if you require more details of the ARM Peripheral Identification specification.

9.1 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 9-187.
- *The Peripheral ID Registers* on page 9-191.

9.1.1 Summary of Component and Peripheral ID Registers

Table 9-1 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 9-1 are required in every debug component implementation.

Table 9-1 Summary of Component and Peripheral ID Registers

Address	Name	Access	Value
`0xFD0`	PIDR4	RO	IMPLEMENTATION DEFINED[a]
`0xFD4`	PIDR5, see PID5-PID7	RO	`0x00000000`
`0xFD8`	PIDR6, see PID5-PID7	RO	`0x00000000`
`0xFDC`	PIDR7, see PID5-PID7	RO	`0x00000000`
`0xFE0`	PIDR0	RO	IMPLEMENTATION DEFINED[a]
`0xFE4`	PIDR1	RO	IMPLEMENTATION DEFINED[a]
`0xFE8`	PIDR2	RO	IMPLEMENTATION DEFINED[a]
`0xFEC`	PIDR3	RO	IMPLEMENTATION DEFINED[a]
`0xFF0`	CIDR0	RO	`0x0000000D`
`0xFF4`	CIDR1	RO	IMPLEMENTATION DEFINED[a]
`0xFF8`	CIDR2	RO	`0x00000005`
`0xFFC`	CIDR3	RO	`0x000000B1`

a. See the register description for more information.

9.2 The Component ID Registers

There are four read-only Component Identification Registers, Component ID register 3 to Component ID register 0, CIDR3-CIDR0. Table 9-2 lists these registers:

Table 9-2 Summary of the Component Identification Registers

Register	Address
CIDR0	`0xFF0`
CIDR1	`0xFF4`
CIDR2	`0xFF8`
CIDR3	`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 9-1 shows the concept of a single 32-bit component ID, obtained from the four Component Identification Registers.

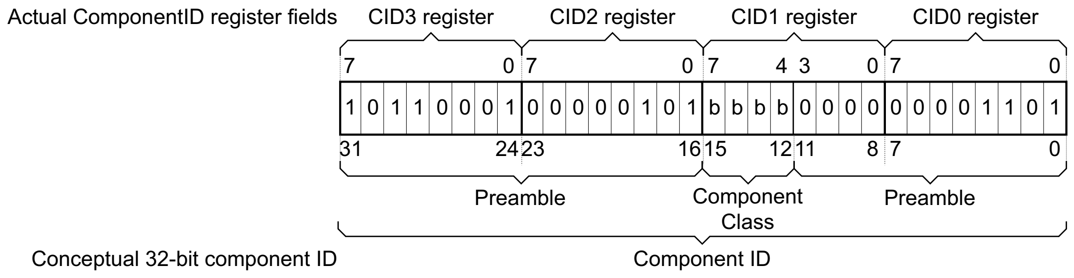


Figure 9-1 Mapping between the Component ID Registers and the Component ID value

The Component Class field identifies the type of the component. Table 9-3 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 9-3 Component Class values in the Component ID

Component class	Class description	Standardized layout
`0x0`	Generic verification component	None.
`0x1`	ROM Table	See Chapter 10 *ROM Tables*.
`0x2` - `0x8`	Reserved	-
`0x9`	Debug component	See the *CoreSight Architecture Specification*.
`0xA`	Reserved	-
`0xB`	Peripheral Test Block (PTB)	None.
`0xC`	Reserved	-

Table 9-3 Component Class values in the Component ID (continued)

Component class	Class description	Standardized layout
`0xD`	OptimoDE Data Engine SubSystem (DESS) component	None.
`0xE`	Generic IP component	None.
`0xF`	PrimeCell peripheral	None.

As shown in Table 9-3 on page 9-187, ROM table components have a standard register space layout. This is defined in Chapter 10 *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 9-191.

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, ARMv7-A and ARMv7-R edition* and the *ETM Architecture Specification*.

The following sections describe each of the Component ID Registers.

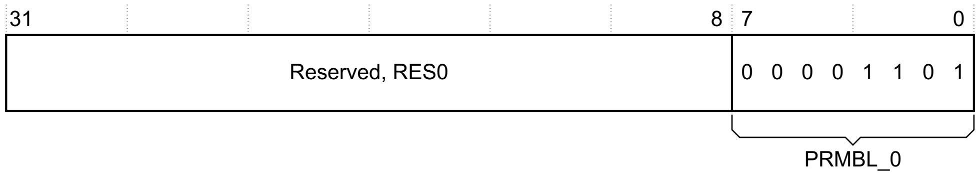
9.2.1 CIDR0, Component ID Register 0

The CIDR0 characteristics are:

Purpose Provides bits[7:0] of the 32-bit conceptual Component ID, see Figure 9-1 on page 9-187.

Attributes A read-only register at address offset `0xFF0`.

The CIDR0 register bit assignments are:



Bits[31:8] Reserved, RES0.

PRMBL_0, bits[7:0]

Preamble byte 0. This byte has the value `0x0D`.

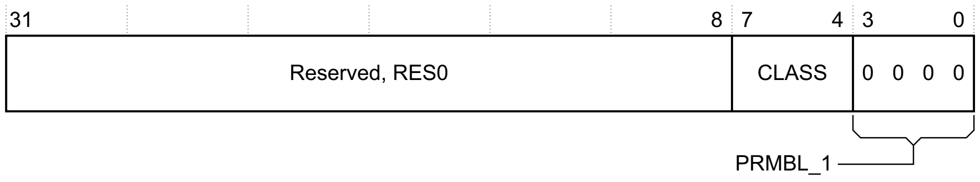
9.2.2 CIDR1, Component ID Register 1

The CIDR1 characteristics are:

Purpose Provides bits[15:8] of the 32-bit conceptual Component ID, see Figure 9-1 on page 9-187.

Attributes A read-only register at address offset `0xFF4`.

The CIDR1 register bit assignments are:



Bits[31:8] Reserved, RES0.

CLASS, bits[7:4]

Component class. Table 9-3 on page 9-187 shows the possible values of this field.

PRMBL_1, bits[3:0]

Preamble bits[11:8]. This field has the value `0x0`.

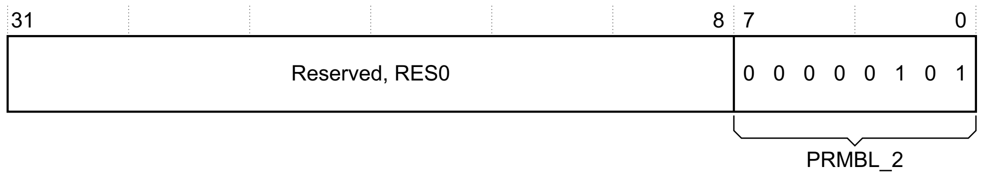
9.2.3 CIDR2, Component ID Register 2

The CIDR2 characteristics are:

Purpose Provides bits[23:16] of the 32-bit conceptual Component ID, see Figure 9-1 on page 9-187.

Attributes A read-only register at address offset `0xFF8`.

The CIDR2 register bit assignments are:



Bits[31:8] Reserved, RES0.

PRMBL_2, bits[7:0]

Preamble byte 2. This byte has the value `0x05`.

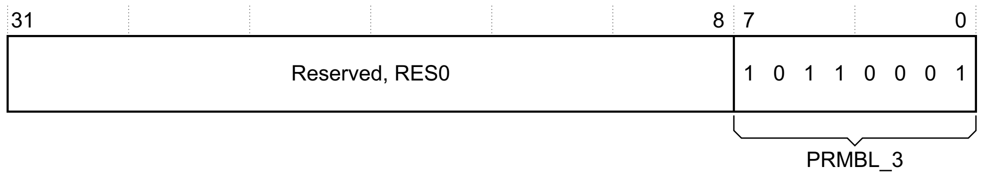
9.2.4 CIDR3, Component ID Register 3

The CIDR3 characteristics are:

Purpose Provides bits[31:24] of the 32-bit conceptual Component ID, see Figure 9-1 on page 9-187.

Attributes A read-only register at address offset `0xFFC`.

The CIDR3 Register bit assignments are:



Bits[31:8] Reserved, RES0.

PRMBL_3, bits[7:0]

Preamble byte 3. This byte has the value `0xB1`.

9.3 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, Peripheral ID register 7 to Peripheral ID register 0, PIDR7-PIDR0. Table 9-4 lists the Peripheral ID registers in order of their address offsets, and gives their offsets within a 4KB register space.

Table 9-4 Summary of the peripheral identification registers

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

Note

Table 9-4 lists the Peripheral ID registers in address order. This does not match the numerical order of the registers, PIDR0-PIDR7.

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 9-2.

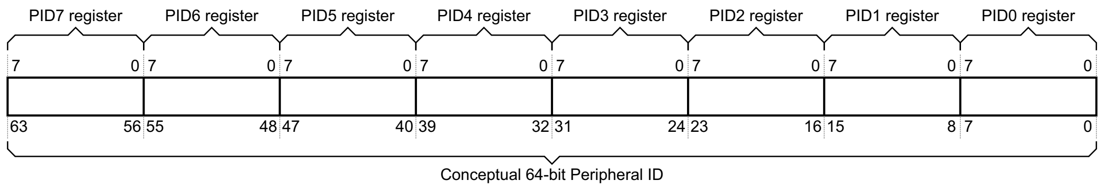


Figure 9-2 Mapping between the Peripheral ID Registers and the Peripheral ID value

Figure 9-3 shows the standard Peripheral ID fields in the single conceptual Peripheral ID, for a peripheral designed by ARM.

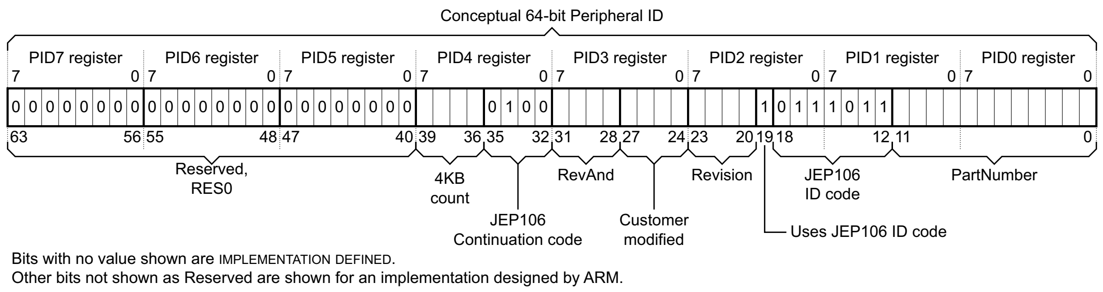


Figure 9-3 Peripheral ID field descriptions

Note

Figure 9-3 on page 9-191 shows the short description of each of the Peripheral ID fields, not the actual PIDR register field names.

Table 9-5 lists the standard Peripheral ID fields, and shows where this information is held in the Peripheral ID Registers.

Table 9-5 Register fields for the peripheral identification registers

Name	Size	Description	Register fields
SIZE	4 bits	4KB count. Log_2 of the number of 4KB blocks occupied by the component. The number of blocks occupied by a component must be a power of two, and this field gives that power. The first four possible values of this field are: `0b0000` 1 block. `0b0001` 2 blocks. `0b0010` 4 blocks. `0b0011` 8 blocks.	PIDR4.SIZE
DES_0, DES_1, DES_2	4+7 bits	JEP106 code. Identifies the designer of the component. This field consists of both: • A 7-bit identity code. DES_ defines bits[3:0]. and DES_1 defines bits[6:4]. • A 4-bit continuation code, defined by DES_2. For an implementation designed by ARM, the continuation code is `0x4` and the identity code is `0x3B`.	PIDR1.DES_0, PIDR2.DES_1, PIDR4.DES_2
REVAND	4 bits	RevAnd. Manufacturer Revision Number. Indicates a late modification to the component, 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.	PIDR3.REVAND
CMOD	4 bits	Customer modified. Indicates an endorsed modification to the component. If the system designer cannot modify the implementation supplied by the designer of the ADI then this field is RAZ.	PIDR3.CMOD
REVISION	4 bits	Revision of the peripheral. Starts at `0x0` and increments by 1 at both major and minor revisions.	PIDR2.REVISION
JEDEC	1 bit	Uses JEP106 ID code. This bit is set to 1 when a JEP106 ID code is used. This bit must be 1 on new implementations.	PIDR2.JEDEC
PART_0, PART_1	12 bits	PartNumber. PART_1 defines PartNumber[11:8], and PART_0 defines PartNumber[7:0]. The part number for the component. Each organization designing components to the ARM Peripheral Identification specifications has its own part number list. For components with a component class 0x9, multiple components are permitted to share the same Part number so long as they have different DEVTYPEs. However, ARM recommends that each component has a unique Part number.	PIDR0.PART_0, PIDR1.PART_1

For more information about these fields, see the *CoreSight Architecture Specification*.

Note

In legacy components, an ASCII Identity Code, allocated by ARM, is used rather than the JEP106 ID code. In such a component:

- The 7-bit ASCII Identity Code replaces the 7-bit *JEP106 ID code* shown in Figure 9-3 on page 9-191. This field is split between the Peripheral ID1 and Peripheral ID2 Registers.

  On legacy components designed by ARM this field has the value `0x41`.

- The *Uses JEP106 ID code* bit, bit [3] of the Peripheral ID2 Register, is zero.
- The Peripheral ID4 Register is not implemented.

Legacy Peripheral ID layout on page 9-197 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 9-4 on page 9-191 lists the register numbers and addresses of these registers, in register address order, which does not match the register name order.

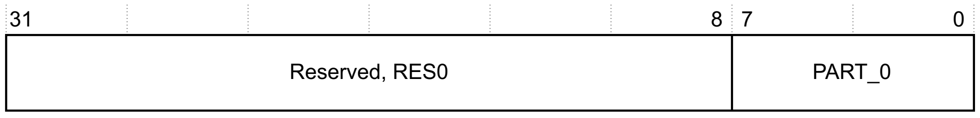
9.3.1 PIDR0, Peripheral ID Register 0

The PIDR0 characteristics are:

Purpose Provides bits[7:0] of the 64-bit conceptual Peripheral ID, see Figure 9-3 on page 9-191.

Attributes A read-only register at address offset `0xFE0`.

The PIDR0 register bit assignments are:



Bits[31:8] Reserved, RES0.

PART_0, bits[7:0]

PartNumber[7:0]. Bits[7:0] of the IMPLEMENTATION DEFINED part number.

See Table 9-5 on page 9-192 for more information about the register fields.

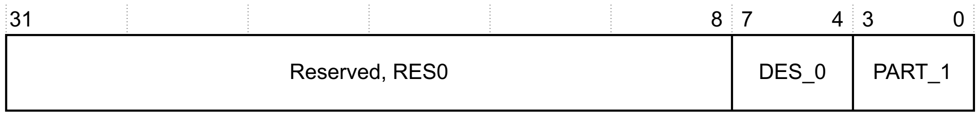
9.3.2 PIDR1, Peripheral ID Register 1

The PIDR1 characteristics are:

Purpose Provides bits[15:8] of the 64-bit conceptual Peripheral ID, see Figure 9-3 on page 9-191.

Attributes A read-only register at address offset `0xFE4`.

The PIDR1 register bit assignments are:



Bits[31:8] Reserved, RES0.

DES_0, bits[7:4]

JEP106 ID code[3:0]. When PIDR2.JEDEC is RAO, bits[3:0] of the IMPLEMENTATION DEFINED JEP106 ID code.

For an implementation designed by ARM, the JEP106 ID code is `0x3B` and therefore this field is `0xB`.

On a legacy component, when PIDR2.JEDEC is RAZ, this field is ASCII Identity Code[3:0]. See also the description of the PIDR2[2:0] field.

PART_1, bits[3:0]

PartNumber[11:8]. Bits[11:8] of the IMPLEMENTATION DEFINED part number.

See Table 9-5 on page 9-192 for more information about the register fields.

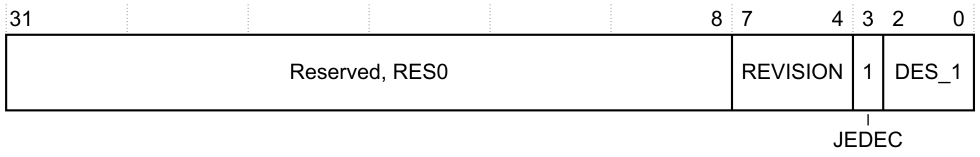
9.3.3 PIDR2, Peripheral ID Register 2

The PIDR2 characteristics are:

Purpose Provides bits[23:16] of the 64-bit conceptual Peripheral ID, see Figure 9-3 on page 9-191.

Attributes A read-only register at address offset `0xFE8`.

The PIDR2 Register bit assignments are:



Bits[31:8] Reserved, RES0.

REVISION, bits[7:4]

The IMPLEMENTATION DEFINED revision number for the implementation.

JEDEC, bit[3]

Uses JEP106 ID code. Indicates whether the Peripheral ID uses a JEP106 ID code.

For new implementations, including all ARMv7 and later implementations, this bit must be RAO, indicating that the Peripheral ID uses a JEP106 ID code.

On a legacy component, this bit might be RAZ, indicating that the Peripheral ID uses an ASCII Identity Code, and that PIDR2[2:0] are ASCII Identity Code[6:4].

DES_1, bits[2:0]

JEP106 ID code[6:4]. When the JEDEC field is RAO, bits[6:4] of the IMPLEMENTATION DEFINED JEP106 ID code.

For an implementation designed by ARM, the JEP106 ID code is `0x3B` and therefore this field is `0b011`.

On a legacy component, when the JEDEC field is RAZ, this field is ASCII Identity Code[6:4].

See Table 9-5 on page 9-192 for more information about the register fields.

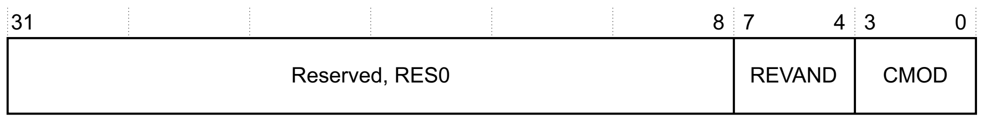
9.3.4 PIDR3, Peripheral ID Register 3

The PIDR3 characteristics are:

Purpose Provides bits[31:24] of the 64-bit conceptual Peripheral ID, see Figure 9-3 on page 9-191.

Attributes A read-only register at address offset `0xFEC`.

The PIDR3 register bit assignments are:



Bits[31:8] Reserved, RES0.

REVAND, bits[7:4]

RevAnd. The IMPLEMENTATION DEFINED manufacturing revision number for the implementation.

CMOD, bits[3:0]

Customer modified. An IMPLEMENTATION DEFINED value that indicates an endorsed modification to the implementation.

If the system designer cannot modify the implementation supplied by the designer of the ADI then this field is RAZ.

See Table 9-5 on page 9-192 for more information about the register fields.

9.3.5 PIDR4, Peripheral ID Register 4

The PIDR4 characteristics are:

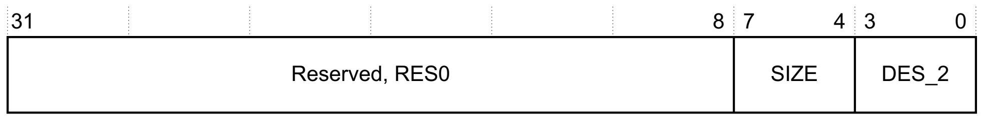
Purpose Provides bits[39:32] of the 64-bit conceptual Peripheral ID, see Figure 9-3 on page 9-191.

Attributes A read-only register at address offset `0xFD0`.

Note

On a legacy component, when bit PIDR2[3] is RAZ, PIDR4 is not implemented. However, the register at `0xFD0` might be used by the component.

The PIDR4 register bit assignments are:



Bits[31:8] Reserved, RES0.

SIZE, bits[7:4]

4KB count. Log_2 of the number of 4KB blocks occupied by the component.

DES_2, bits[3:0]

JEP106 Continuation code. The IMPLEMENTATION DEFINED JEP106 Continuation code.

For an implementation designed by ARM this field is `0x4`.

See Table 9-5 on page 9-192 for more information about the register fields.

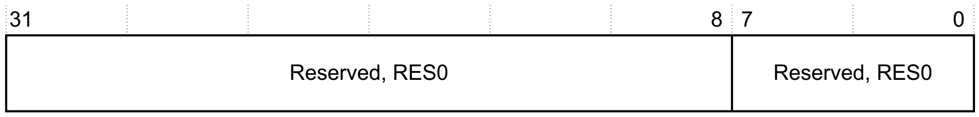
9.3.6 PIDR5-PIDR7, Peripheral ID Registers 5-7

Registers PIDR5-PIDR7 do not hold any information. Table 9-4 on page 9-191 shows the address offsets of these registers.

Note

On legacy components, when bit PIDR2[3] is RAZ, the Peripheral ID5 to Peripheral ID7 registers PIDR5-PIDR7 are not implemented. However, the registers at `0xFD4`, `0xFD8`, and `0xFDC` might be used by the component.

The bit assignments for registers PIDR5-PIDR7 are:



Bits[31:8] Reserved, RES0.

Reserved, bits[7:0]

Reserved, RES0.

9.3.7 Legacy Peripheral ID layout

Table 9-6 shows the format of the peripheral identification registers in a legacy component

Table 9-6 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 JEP106 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 10
ROM Tables

The chapter describes ROM Tables. It includes the following sections:

- *ROM Table overview* on page 10-200.
- *ROM Table entries* on page 10-202.
- *The MEMTYPE register* on page 10-205.
- *Component and Peripheral ID Registers* on page 10-206.
- *ROM Table hierarchies* on page 10-207.

10.1 ROM Table overview

ROM Tables 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 1-5 on page 1-29.
- 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.

A ROM Table is divided into a number of regions, as shown in Table 10-1 and Figure 10-1 on page 10-201.

Table 10-1 ROM Table regions

Region	Start address	End address	Notes
ROM Table entries	`0x000`	IMPLEMENTATION DEFINED	See *ROM Table entries* on page 10-202.
Reserved region	IMPLEMENTATION DEFINED	`0xFCB`	Unused area of ROM Table. See *The unused area of the ROM Table* on page 10-204
MEMTYPE register	`0xFCC`	`0xFCF`	-
Peripheral ID registers	`0xFD0`	`0xFEF`	See *Component and Peripheral ID Registers* on page 10-206.
Component ID registers	`0xFF0`	`0xFFF`	

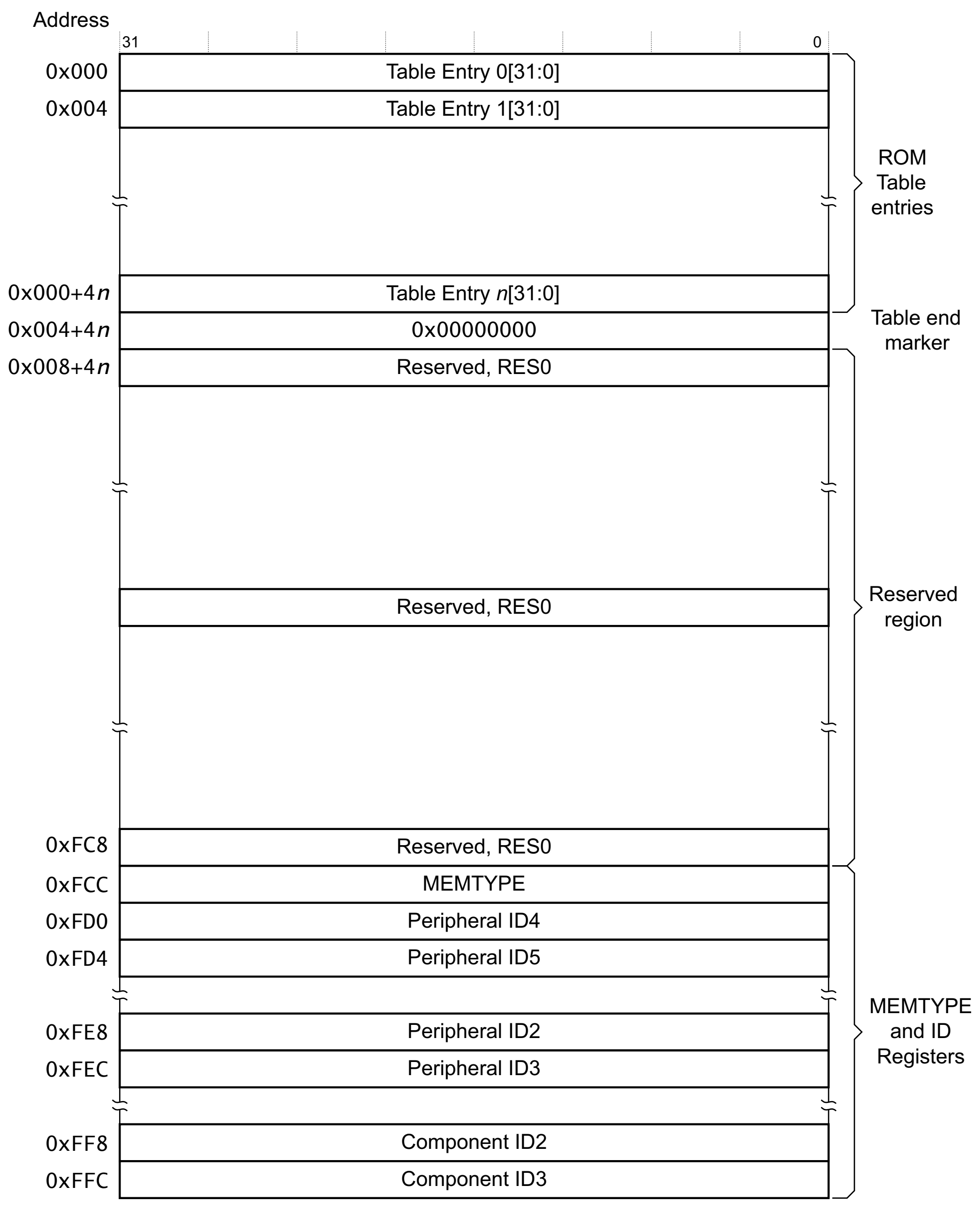


Figure 10-1 ROM Table format

Note

The ROM Table area from `0xF00` to `0xFCB` is reserved, RES0. The last possible ROM Table entry is at `0xEFC`.

10.2 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.
- Is a single 32-bit word.

Table 10-2 shows the format of a single entry in the ROM Table.

Table 10-2 Format of a ROM Table entry

Bits	Field name	Description
[31:12]	Address offset	The base address of the component, relative to the base address of this ROM Table. Negative values are permitted, using two's complement. **Note** 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. See *Component descriptions and the component base address* on page 10-203 for more information.
[11:9]	-	Reserved. RES0.
[8:4]	Power domain ID[a]	Indicates the power domain of the component. This field: • Is only valid if bit[2]==1., otherwise this field must be RAZ. • Supports up to 32 power domains using values `0x00` to `0x1F`.
[3]	-	Reserved. RES0.
[2]	Power domain ID valid[a]	Indicates if the Power domain ID field contains a Power domain ID: **0** Power domain ID not provided. **1** Power domain ID provided in bits[8:4].
[1]	Format	This bit indicates the format of the ROM Table. It is RAO, indicating a 32-bit ROM Table format.
[0]	Entry present	This bit indicates whether an entry is present at this location in the ROM Table. Its possible values are: **0** Entry not present. **1** Entry present. See *Empty entries and the end of the ROM Table* on page 10-204 for more information.

a. See *CoreSight Architecture Specification* for more information on Power domains.

10.2.1 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 1-3 on page 1-28, and other figures in Chapter 1 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 PIDR4[7:4], 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:

   Component_Base_Address = ROM_Base_Address + (Address_Offset << 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:

   Peripheral_ID4_address = 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 `0b0000`, 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 DAP to the system being debugged. For more information about accesses to addresses that are not pointed to by the ROM Table see MEMTYPE.

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.

ARM recommends that the debug component base address is aligned to the largest translation granule supported by any processor that can access the component. For an ARMv8 processor, this might be 64KB.

For more information about the Component and Peripheral ID Registers see Chapter 9 *Component and Peripheral ID Registers*.

10.2.2 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 10-207 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 with the value `0x00000000`.

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, RES0. Similarly, the reserved area starting at offset `0xF00` and ending at `0xFCB`, immediately before the MEMTYPE register, is RES0.

Note

- A ROM Table can hold up to 960 entries.
- 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`.
- 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.

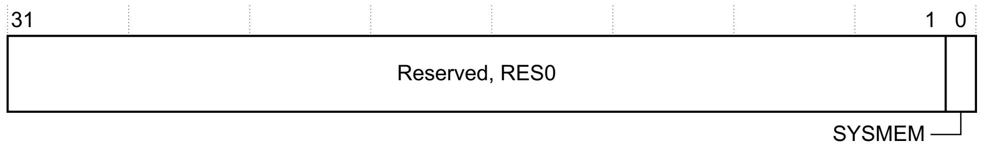
  An implementation is unlikely to require the maximum number of entries in a ROM Table. However, *ROM Table hierarchies* on page 10-207 describes how larger ROM Tables can be constructed.

10.3 The MEMTYPE register

The MEMTYPE register characteristics are:

Purpose	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.
Configurations	Every ROM Table must implement a MEMTYPE register.
Attributes	A read-only register, at offset `0xFCC` from the base address of the ROM Table.

The MEMTYPE register bit assignments are:



Bits[31:1] Reserved, RES0.

SYSMEM, bit[0]

System memory present. Indicates whether system memory is present on the bus that connects to the ROM Table. The possible values are:

0	System memory not present on bus. This is a dedicated debug bus.
1	System memory is also present on this bus.

MEMTYPE.SYSMEM indicates the memory accesses that the DAP can make:

When SYSMEM is RAZ

The ROM Table indicates all the valid addresses in the memory system that the DAP is connected to, and the result of accessing any other address is UNPREDICTABLE. For more information, see *Component descriptions and the component base address* on page 10-203.

When SYSMEM is RAO

There might be other valid addresses in the memory system that the DAP is connected to. The result of accessing these addresses is IMPLEMENTATION DEFINED, and:

- The ADIv5 specification does not include any mechanism that the DAP can use to discover what addresses it can access, other than those indicated by the ROM Table.
- If the DAP accesses addresses other than those indicated by the ROM Table this might have side effects on the system that the DAP is connected to.

10.4 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. Chapter 9 *Component and Peripheral ID Registers* describes these registers. This section only describes particular features of the registers when they relate to a ROM Table.

In a ROM Table implementation:

- The Component class field, CIDR1.CLASS is `0x1`, identifying the component as a ROM Table.
- The PIDR4.SIZE field must be 0. This is because a ROM Table must occupy a single 4KB block of memory.

10.4.1 Identifying the debug SoC or platform

The top-level ROM Table Peripheral ID Registers uniquely identify the SoC or platform. If a system has more than one Memory Access Port with a connected ROM Table then the information from the Peripheral ID Registers of all of the top-level ROM Tables, considered collectively, is required to uniquely identify the SoC or platform. An example of a system with multiple MEM-APs is shown in Figure 1-6 on page 1-30.

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 DAP, it can save its description of the system with the Peripheral ID. If that system is connected to the debugger again, the debugger can retrieve that saved description, avoiding any requirement for topology detection.

If two different systems have the same Peripheral ID a debugger might retrieve an incorrect description. If this situation occurs, you must force the debugger to perform topology detection again.

If the DAP implements DPv2, the DP TARGETID register also uniquely identifies the SoC or platform, and ARM deprecates use of the top-level ROM Table Peripheral ID registers as a unique identifier by tools.

Note

- If SWJ-DP is implemented, it is not required that both the JTAG-DP and SW-DP implement the same DP architecture version, and therefore TARGETID. Tools might be using a DP that does not implement DPv2.
- Deprecation of the use of the top-level ROM Table peripheral ID registers by tools does not remove the requirement on implementations to provide a unique identifier in the top-level ROM Table peripheral ID registers. Future releases of this manual might remove this requirement.

10.5 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 *Component descriptions and the component base address* on page 10-203 describes. 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 9-187.

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.

Figure 10-2 shows an example of a ROM Table hierarchy.

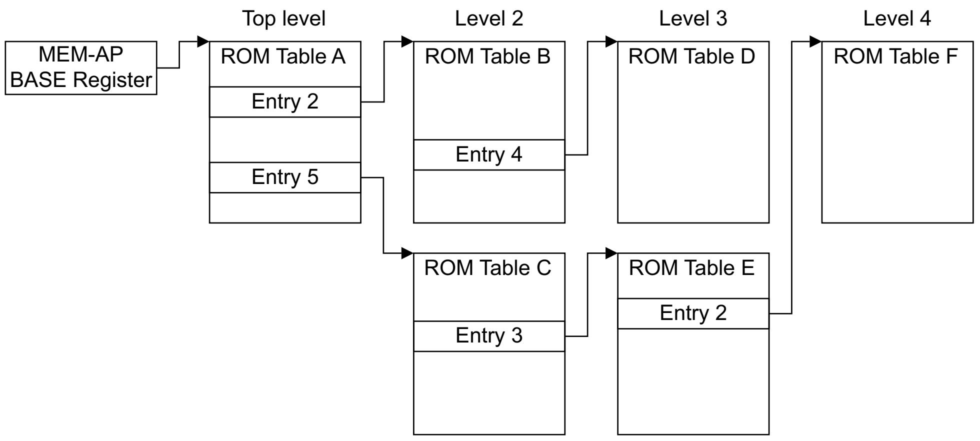


Figure 10-2 ROM Table hierarchy example

A hierarchy of ROM Tables might increase the total number of ROM Table entries in the system.

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.

10.5.1 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 9-187. It is particularly important that the Component class field of the Component ID1 Register correctly identifies the component as a ROM Table.

10.5.2 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 10-3 shows a prohibited case, where entries in ROM Tables B and C both point to ROM Table D.

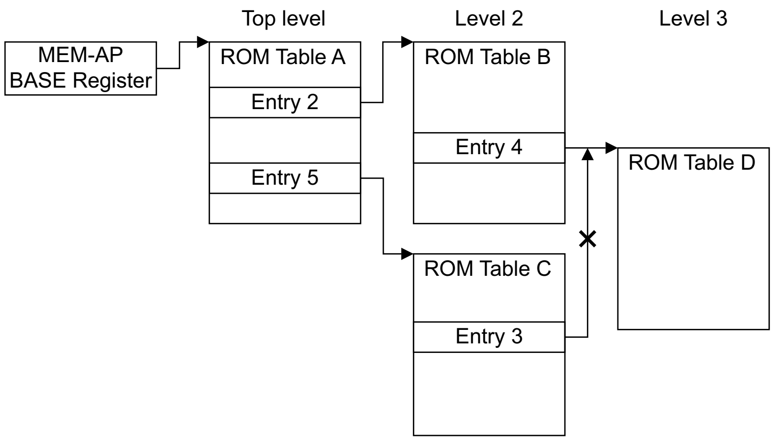


Figure 10-3 Prohibited duplicate ROM Table reference from ROM Table

Figure 10-4 shows a similar prohibited case, where entries in ROM Table A and MEM-AP 2 both point to ROM Table B.

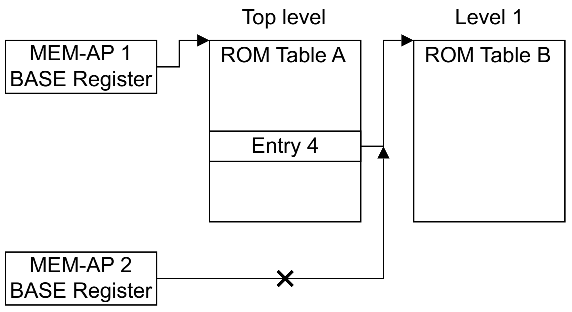


Figure 10-4 Prohibited duplicate ROM Table reference from MEM-AP

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 10-5 on page 10-209, where both ROM Table B and ROM Table C have prohibited links back to ROM Table A.

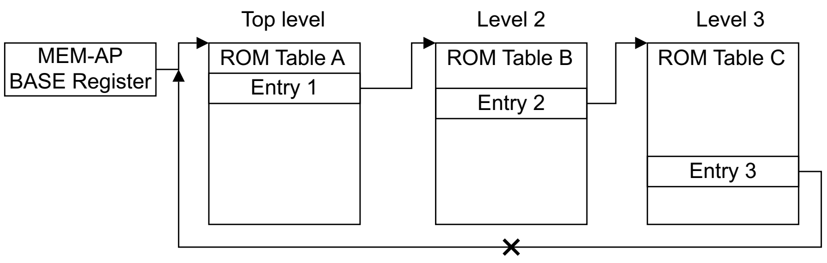


Figure 10-5 Prohibited circular ROM Table references

——— **Note** ———

There is no requirement that components in separate ROM Table hierarchies must be in separate systems. This includes multiple APs in a single DAP, and multiple DAPs. For example, if MEM-AP 1 in DAP 1 points to a hierarchy of ROM Tables which includes a pointer to trace macrocell A, and MEM-AP 2 in DAP 2 points to a hierarchy of ROM Tables which includes a pointer to trace sink B, then trace from trace macrocell A can be collected by trace sink B, as shown in Figure 10-6.

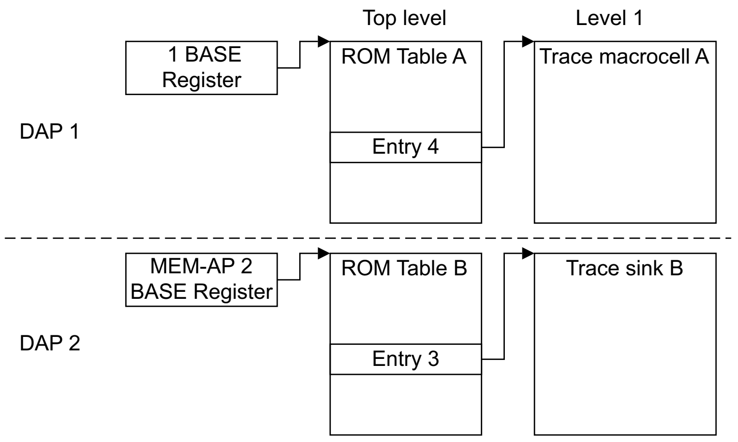


Figure 10-6 Components in separate ROM Table hierarchies

———————————

Appendix A
Standard Memory Access Port Definitions

This appendix provides information on implementing the Memory Access Port (MEM-AP). It contains the following sections:

- *Introduction* on page A-212.
- *AMBA AXI3 and AXI4* on page A-213.
- *AMBA AXI4 with ACE-Lite* on page A-215.
- *AMBA AHB* on page A-217.
- *AMBA APB2 and APB3* on page A-218.

A.1 Introduction

The Memory Access Port (MEM-AP) programmers' model includes IMPLEMENTATION DEFINED features. This appendix provides reference implementation options for implementers and users of MEM-APs when connecting to standard memory interfaces. In particular, it provides the recommended interpretations of the CSW.{Prot, SPIDEN, Type, AddrInc, Size} fields.

A.2 AMBA AXI3 and AXI4

For more information, see *AMBA® AXI™ and ACE™ Protocol Specification AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™*. For AMBA AXI3 and AXI4 implementations, the CSW register is implemented as follows:

Prot, bits [30:24]

For reads, the CSW.Prot field drives the AXI **ARCACHE** and **ARPROT** signals.

For writes, the CSW.Prot field drives the AXI **AWCACHE** and **AWPROT** signals.

The settings for the CSW.Prot field are:

PROT[2:0], bits [30:28]

Drives **AxPROT[2:0]**, where **x** is **R** for reads and **W** for writes, see Table A-1.

Table A-1 CSW.Prot mapping to ARPROT or AWPROT

Bit	ARPROT signal	AWPROT signal	Description
30	**ARPROT[2]**	**AWPROT[2]**	Instruction
29	**ARPROT[1]**	**AWPROT[1]**	Non-secure
28	**ARPROT[0]**	**AWPROT[0]**	Privileged

CSW.Prot[29], Non-secure, specifies a non-secure transfer. Its behavior depends on the vale of CSW.SPIDEN. For values in CSW.Prot[29]:

1 Non-secure transfer requested. **ARPROT[1]** or **AWPROT[1]** is HIGH.

0 Secure transfer requested. If CSW.SPIDEN is 1, **ARPROT[1]** or **AWPROT[1]** is LOW. If CSW.SPIDEN is 0, no transfer is initiated.

CACHE[3:0], bits [27:24]

Drives **AxCACHE[3:0]**, where **x** is **R** for reads and **W** for writes, see Table A-2.

Table A-2 CSW.Prot mapping to ARCACHE or AWCACHE

Bit	ARCACHE signal	AWCACHE signal
27	**ARCACHE[3]**	**AWCACHE[3]**
26	**ARCACHE[2]**	**AWCACHE[2]**
25	**ARCACHE[1]**	**AWCACHE[1]**
24	**ARCACHE[0]**	**AWCACHE[0]**

Note

AMBA AXI4 requires asymmetrical usage of **ARCACHE** and **AWCACHE**.

The reset value of CSW.Prot is `0b0110000`.

SPIDEN, bit [23] The CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**.

Type, bits [15:12] RES0.

Mode, bits [11:8] RES0.

AddrInc, bits [5:4] CSW.AddrInc supports the *Increment Packed* mode of transfer. See *Packed transfers* on page 7-138.

Size, bits [3:0]	CSW.Size must support word, half-word, and byte size accesses. It is IMPLEMENTATION DEFINED whether larger access sizes are supported.

A.3 AMBA AXI4 with ACE-Lite

For more information, see the *AMBA® AXI™ and ACE™ Protocol Specification AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™*. For AMBA AXI4 implementations with ACE-Lite the following registers are implemented as shown:

- *CSW register implementation*.
- *MBT register implementation* on page A-216.

A.3.1 CSW register implementation

The CSW register is implemented as follows:

Prot, bits[30:24]

For reads, the CSW.Prot field drives the AXI **ARCACHE** and **ARPROT** signals.

For writes, the CSW.Prot field drives the AXI **AWCACHE** and **AWPROT** signals.

The settings for the CSW.Prot field are:

PROT[2:0], bits[30:28]

Drives **AxPROT[2:0]**, where **x** is **R** for reads and **W** for writes, see Table A-3.

Table A-3 CSW.Prot mapping to ARPROT or AWPROT

Bit	ARPROT signal	AWPROT signal	Description
30	**ARPROT[2]**	**AWPROT[2]**	Instruction
29	**ARPROT[1]**	**AWPROT[1]**	Non-secure
28	**ARPROT[0]**	**AWPROT[0]**	Privileged

CSW.Prot[29], Non-secure, specifies a non-secure transfer. Its behavior depends on the value of CSW.SPIDEN. For values in CSW.Prot[29]:

1 Non-secure transfer requested. **ARPROT[1]** or **AWPROT[1]** is HIGH.

0 Secure transfer requested. If CSW.SPIDEN is 1, **ARPROT[1]** or **AWPROT[1]** is LOW. If CSW.SPIDEN is 0, no transfer is initiated.

CACHE[3:0], bits[27:24]

Drives **AxCACHE[3:0]**, where **x** is **R** for reads and **W** for writes, see Table A-4.

Table A-4 CSW.Prot mapping to ARCACHE or AWCACHE

Bit	ARCACHE signal	AWCACHE signal
27	**ARCACHE[3]**	**AWCACHE[3]**
26	**ARCACHE[2]**	**AWCACHE[2]**
25	**ARCACHE[1]**	**AWCACHE[1]**
24	**ARCACHE[0]**	**AWCACHE[0]**

Note

AMBA AXI4 requires asymmetrical usage of **ARCACHE** and **AWCACHE**.

The reset value of CSW.Prot is `0b0110000`.

SPIDEN, bit[23] The CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**.

Type, bits[15:12]

The CSW.Type field drives the AXI **AxDOMAIN** signals, where **x** is **R** for reads and **W** for writes.

The settings for the CSW.Type bit field are:

Bit[15] . Reserved, RES0.

DOMAIN[1:0], bits [14:13]

Possible values are:

`0b00`	Non-shareable.
`0b01`	Inner shareable, includes additional masters.
`0b10`	Outer shareable, contains all masters in the Inner shareable domain and can include additional masters.
`0b11`	System, includes all masters.

EnMBT, bit [12]

Enable MBT accesses.

It is IMPLEMENTATION DEFINED whether this field is RW or RAO. If it is RW, the reset value is 0, and must be set to 1 before writing to the MBT register.

Mode, bits [11:8]

It is IMPLEMENTATION DEFINED whether this field is RW or RO. If it is RW, the reset value is `0b0000`, and must be set to `0b0001` before writing to the MBT register. If it is RO, then it has the fixed value `0b0001`.

AddrInc, bits [5:4]

CSW.AddrInc supports the *Increment Packed* mode of transfer. See *Packed transfers* on page 7-138.

Size, bits [3:0]

CSW.Size must support word, half-word, and byte size accesses. It is IMPLEMENTATION DEFINED whether larger access sizes are supported.

A.3.2 MBT register implementation

The MBT register is implemented as follows:

Attributes

MBT register is a read/write register.

Bits[31:3]

Reserved, RES0.

BarTran, bits[2:1]

Possible values are:

`0b00`	Reserved
`0b01`	Memory barrier
`0b10`	Reserved
`0b11`	Synchronization barrier.

Bit[0]

On reads:

0	Barrier transaction in progress.
1	No barrier transaction in progress.

SBO on writes.

A.4 AMBA AHB

For more information, see the *AMBA® Specification (Rev 2.0)* and the *AMBA® 3 AHB-Lite™ Protocol Specification*. For AMBA AHB implementations, the CSW register is implemented as follows:

Prot, bits[30:24] The CSW.Prot field drives the AHB **HPROT** signals. The settings for the CSW.Prot field are:

Bit[30] Reserved, SBO. If this bit is written as 0 the behavior of an AHB-AP transaction is UNPREDICTABLE.

MasterType, bit[29] Master Type bit. MasterType permits the AHB-AP to mimic a second AHB master by driving a different value on **HMASTER[3:0]**. Support for this function is IMPLEMENTATION DEFINED. Valid values for this bit are:

1 Drive **HMASTER[3:0]** with the bus master ID for the AHB-AP.

0 Drive **HMASTER[3:0]** with the bus master ID for the second bus master.

If this function is not implemented the bit is RAZ/WI.

HPROT[4], Allocate, bit[28]

Drives **HPROT[4]**, Allocate. **HPROT[4]** is an ARMv6 extension to AHB. For more information, see the *ARM1136JF-S™ and ARM1136J-S™ Technical Reference Manual*.

If the AHB master interface does not support the ARMv6 extension to AHB, this bit is RAZ/WI.

HPROT[3:0], bits[27:24]

Drives **HPROT[3:0]**. See Table A-5. Support for each **HPROT** signal in the AHB master interface is IMPLEMENTATION DEFINED.

Table A-5 CSW.Prot mapping

Bit	HPROT signal	Description	Description when not implemented at the AHB master interface
27	**HPROT[3]**	Cacheable	RAZ/WI
26	**HPROT[2]**	Bufferable	RAZ/WI
25	**HPROT[1]**	Privileged	RAO/WI
24	**HPROT[0]**	Data	RAO/WI

The reset value of CSW.Prot is `0b1000011`.

SPIDEN, bit[23] It is IMPLEMENTATION DEFINED whether the CSW.SPIDEN bit reflects the state of the CoreSight authentication signal, **SPIDEN**. Otherwise, the CSW.SPIDEN bit is RAZ.

This bit is always read-only.

Note

AMBA AHB does not support Security Extensions.

Type, bits[15:12] RES0.

Mode, bits[11:8] RES0.

AddrInc, bits[5:4] Support for the *Increment Packed* mode of transfer is IMPLEMENTATION DEFINED. See *Packed transfers* on page 7-138.

Size, bits[3:0] CSW.Size must support word, half-word, and byte size accesses.

A.5 AMBA APB2 and APB3

For more information see the *AMBA® Specification (Rev 2.0)*, and the *AMBA® 4 APB™ Protocol Specification* For AMBA APB2 and APB3 implementations, the CSW register is implemented as follows:

Prot, bits[30:24]	RES0.
SPIDEN, bit[23]	RES0.
Type, bits[15:12]	RES0.
Mode, bits[11:8]	RES0.
AddrInc, bits[5:4]	CSW.AddrInc does not support the *Increment Packed* mode of transfer. See *Packed transfers* on page 7-138.
Size, bits[3:0]	CSW.Size only supports word accesses, and reads as `0b010`. Writes to CSW.Size are ignored.

Appendix B
Cross-over with the ARM Architecture

This appendix describes the required or recommended options for the ARM Debug Interface for the ARMv6-M and all ARMv7 and ARMv8 architecture profiles. It contains the following sections:

- *Introduction* on page B-220.
- *ARMv6-M architecture* on page B-221.
- *ARMv7-M architecture profile* on page B-222.
- *ARMv7-A without Large Physical Address Extension and ARMv7-R architecture profiles* on page B-223.
- *ARMv7-A with Large Physical Address Extension and ARMv8-A architecture profiles* on page B-224.
- *Summary of the required ADIv5 implementations* on page B-225.

B.1 Introduction

The ARM Debug Interface v5 is the recommended external debug interface for ARMv6-M, all ARMv7, and all ARMv8 architecture profiles.

When designing with ARM Cortex™ processor cores and ARM CoreSight technology, the choice of Debug Access Port (DAP) features might be at the discretion of the system designer. ARM recommends that system designers choose a DAP that implements all the recommended features for all ARM architecture processor cores contained in the design.

ADIv5 might also be used with other architecture variants. For example, an ADIv5 JTAG Access Port (JTAG-AP) might access a Debug Test Access Port (DBGTAP), as defined by ARM Debug Interface v4 (ADIv4) for ARMv6 architecture processors.

B.2 ARMv6-M architecture

The ARMv6-M architecture requires an ADIv5-compliant DAP.

ARM recommends that the Debug Port (DP) implements the SWD interface, either through a SW-DP or SWJ-DP. A JTAG-DP is permitted. ARM recommends that the DP implements the MINDP model. See *MINDP, Minimal Debug Port extension* on page 2-36.

There must be one MEM-AP for each processor. Each MEM-AP must be capable of addressing the complete memory space visible to the processor, including all debug peripherals and the NVIC. The MEM-AP must support byte, half-word and word size accesses to memory. The MEM-AP is not required to support the packed increment transfer mode.

Other Access Ports (APs) can be connected to the DAP.

B.3 ARMv7-M architecture profile

The ARMv7-M architecture profile requires an ADIv5-compliant DAP.

ARM recommends that the DP implements the SWD interface, either through a SW-DP or SWJ-DP. A JTAG-DP is permitted. ARM recommends that the DP does not implement the MINDP model.

There must be one MEM-AP for each processor. Each MEM-AP must be capable of addressing the complete memory space visible to the processor, including all debug peripherals and the NVIC. The MEM-AP must support byte, half-word and word size accesses to memory. ARM recommends that the MEM-AP supports the packed increment transfer mode.

Other APs can be connected to the DAP.

B.4 ARMv7-A without Large Physical Address Extension and ARMv7-R architecture profiles

The ARMv7-A and ARMv7-R architecture profiles do not require an ADIv5-compliant DAP. Although the ADIv5 interface is not required for compliance with ARMv7, the ARM development tools require this interface to be implemented.

Where an ADIv5-compliant DAP is implemented, ARM recommends that the DP implements the JTAG and Serial Wire Debug interfaces through an SWJ-DP. ARM recommends that the DP does not implement the MINDP model.

Many processors can be connected to a single MEM-AP. The MEM-AP must only be capable of addressing the debug peripherals of the connected processors. If the MEM-AP can only address the debug peripherals, it is only required to support word size accesses to memory, and therefore is not required to support the packed increment transfer mode.

ARM recommends that debug implementations include a MEM-AP that can address the complete memory space visible to the processor or processors. This might be a second MEM-AP in the DAP. ARM recommends that a MEM-AP that can access the complete memory space supports byte, half-word and word size accesses to memory. ARM recommends that this MEM-AP supports the packed increment transfer mode.

Other APs can also be connected to the DAP.

Note

Do not confuse the ARMv7-A Large Physical Address Extension with the ADIv5 MEM-AP Large Physical Address Extension.

B.5 ARMv7-A with Large Physical Address Extension and ARMv8-A architecture profiles

The requirements for ARMv7-A with Large Physical Address Extension and ARMv8-A architecture profiles are the same as for the ARMv7-A without Large Physical Address Extension and ARMv7-R architecture profiles, with the following additions for any MEM-AP with system access:

- MEM-AP Large Physical Address Extension, up to at least the size supported by the processor.
- For ARMv7-A systems with the Large Physical Address Extension, ARM recommends the MEM-AP implements the Large Data Extension providing at least doubleword accesses, to allow for atomic update of page table entries.
- For ARMv8-A systems, the MEM-AP must implement the Large Data Extension providing at least doubleword accesses.

B.6 Summary of the required ADIv5 implementations

Table B-1 summarizes the required and recommended components of an ADIv5 implementation for each of the ARM architecture variants for which ADIv5 is the required or recommended DAP.

Table B-1 Recommended ADIv5 implementations for ARM Architecture variants

Component		ARMv6-M	ARMv7-M	ARMv7-A without Large Physical Address Extension and ARMv7-R	ARMv7-A with Large Physical Address Extension and ARMv8-A
ADIv5 DAP		Required	Required	Recommended	Recommended
DP	JTAG-DP	Permitted	Permitted	Permitted	Permitted
	SW-DP	-	-	Permitted	Permitted
	SWJ-DP	-	-	Recommended	Recommended
	SWJ-DP or SW-DP	Recommended	Recommended	-	-
	Not MINDP	Permitted	Recommended	Recommended	Recommended
MEM-AP	One per processor	Required	Required	Permitted	Permitted
	Access to system memory	Required	Required	Permitted	Permitted
	Support for 8-bit and 16-bit accesses	Required	Required	Required only if system access is supported	Required only if system access is supported
	Support for 32-bit accesses	Required	Required	Required	Required
	Support for 64-bit accesses	Permitted	Permitted	Permitted	Required
	Support for large physical addresses	Not permitted	Not permitted	Permitted	Required if system access is supported
	Support for packed increment transfers	Permitted	Recommended	Recommended	Recommended

Appendix C
Pseudocode Definition

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

- *About ARM pseudocode* on page C-228.
- *Data types* on page C-229.
- *Expressions* on page C-233.
- *Operators and built-in functions* on page C-235.
- *Statements and program structure* on page C-240.

C.1 About ARM pseudocode

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

- *Data types* on page C-229.
- *Expressions* on page C-233.
- *Operators and built-in functions* on page C-235.
- *Statements and program structure* on page C-240.

C.1.1 General limitations of ARM pseudocode

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

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

For more information, see *Simple statements* on page C-240.

C.2 Data types

This section describes:
- *General data type rules*.
- *Bitstrings*.
- *Integers* on page C-230.
- *Reals* on page C-230.
- *Booleans* on page C-230.
- *Enumerations* on page C-230.
- *Lists* on page C-231.
- *Arrays* on page C-232.

C.2.1 General data type rules

ARM architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:
- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- List.
- Array.

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

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

The remaining subsections describe each data type in more detail.

C.2.2 Bitstrings

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

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

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

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons, see *Equality and non-equality testing* on page C-235.

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

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

C.2.3 Integers

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

The type name for integers is `integer`.

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

C.2.4 Reals

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

The type name for reals is `real`.

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

C.2.5 Booleans

A Boolean is a logical true or false value.

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

C.2.6 Enumerations

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

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

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

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

Note

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

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

C.2.7 Lists

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

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

A list always contains at least one data item.

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

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

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

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

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

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

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

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

After this definition and the declaration:

```
ShiftSpec abc;
```

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

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

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

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

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

C.2.8 Arrays

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

For example:

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

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

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

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

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

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

C.3 Expressions

This section describes:

- *General expression syntax*.
- *Operators and functions - polymorphism and prototypes* on page C-234.
- *Precedence rules* on page C-234.

C.3.1 General expression syntax

An expression is one of the following:

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

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

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

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

Note

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

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

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

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

Every expression has a data type:

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

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

C.3.2 Operators and functions - polymorphism and prototypes

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

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

C.3.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables, and function invocations are evaluated with higher priority than any operators using their results.
2. Expressions on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but this is not necessary if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j`, and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

C.4 Operators and built-in functions

This section describes:

- *Operations on generic types*.
- *Operations on Booleans*.
- *Bitstring manipulation*.
- *Arithmetic* on page C-238.

C.4.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

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

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

Note

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

Conditional selection

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

C.4.2 Operations on Booleans

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

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

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

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

C.4.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If `x` is a bitstring:

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

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

Bitstring concatenation and replication

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

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

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

Bitstring extraction

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

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

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

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

  `x<i> : x<j> : x<k> : … : x<n>`.

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

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

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

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

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the ABORT register shows its bit<3> as WDERRCLR. In such cases, the syntax `ABORT.WDERRCLR` is used as a more readable synonym for `ABORT<3>`.

Logical operations on bitstrings

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

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

Bitstring count

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

Testing a bitstring for being all zero or all ones

If x is a bitstring:

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

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

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

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

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

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

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

Zero-extension and sign-extension of bitstrings

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

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

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

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

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

Converting bitstrings to integers

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

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

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

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

```
// UInt()

// ======


integer UInt(bits(N) x)

    result = 0;

    for i = 0 to N-1

        if x<i> == '1' then result = result + 2^i;

    return result;
```

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

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

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

C.4.4 Arithmetic

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

Unary plus, minus, and absolute value

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

Addition and subtraction

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

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

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

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

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

Comparisons

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

Multiplication

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

Division and modulo

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

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

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

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

Square root

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

Rounding and aligning

If x is a real:

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

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

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

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

Scaling

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

If x and n are of type integer, then:

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

Maximum and minimum

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

C.5 Statements and program structure

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

- *Simple statements*.
- *Compound statements* on page C-241.
- *Comments* on page C-244.

C.5.1 Simple statements

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

Assignments

An assignment statement takes the form:

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

Procedure calls

A procedure call takes the form:

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

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

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

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

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

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

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

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

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

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

IMPLEMENTATION_DEFINED

This subsection describes the statement:

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

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

SUBARCHITECTURE_DEFINED

This subsection describes the statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

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

C.5.2 Compound statements

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

Indentation is normally done by four spaces for each level.

if … then … else …

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

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

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

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

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

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

Note

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

repeat … until …

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

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

while … do

A `while … do` structure takes the form:

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

for …

A `for …` structure takes the form:

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

case … of …

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

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

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

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. For details see *Equality and non-equality testing* on page C-235.

Procedure and function definitions

A procedure definition takes the form:

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

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

Note

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

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

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

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

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

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

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

C.5.3 Comments

Two styles of pseudocode comment exist:

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

Glossary

This glossary describes some of the terms used in technical documents from ARM.

Abort A mechanism that indicates that the value associated with a memory access is invalid.

ADI *See* ARM Debug Interface (ADI).

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 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 (AMBA).

Advanced Microcontroller Bus Architecture (AMBA)

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM 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.

AHB *See* Advanced High-performance Bus (AHB).

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 (AMBA).

APB *See* Advanced Peripheral Bus (APB).

Architecture The organization of either or both of hardware and 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 1-18 for a summary of earlier versions of the ARM Debug Interface.

AXI *See* Advanced eXtensible Interface (AXI).

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.

DAP *See* Debug Access Port (DAP).

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

DATA LINK DEFINED

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

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

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

Debug Access Port (DAP)
A TAP block that acts as an AMBA, AHB or AHB-Lite, master for access to a system bus. The DAP is the term that encompasses 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 system that finds faults in the execution of software.

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 resets 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 (DMA).

Doubleword A 64-bit data item.

Embedded Trace Macrocell (ETM)
A hardware component that, when connected to a processor core, generates instruction and data trace information.

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 (ETM)*.

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.

IMPLEMENTATION DEFINED
The behavior is not architecturally defined, but is defined and documented by individual implementations.

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

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* JTAG.

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-AP).

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 (JTAG-DP).

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.

Powerup reset A reset asserted when the device is first powered up.

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

RAO/WI Read-as-One, Writes Ignored. Hardware must implement the field as Read-as-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, and all writes being ignored. This applies to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

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

RAZ/WI Read-as-Zero, Writes ignored. Hardware must implement the field as Read-as-Zero, and must ignore writes to the field. Software can rely on the field reading as all 0s, and all writes being ignored. This applies to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-As-One (RAO)

Read-as-One. Hardware must implement the field as Read-as-One. Software can rely on the field reading as all 1s. This applies to a single bit that reads as 1, or to a field that reads as all 1s.

Read-As-Zero (RAZ)

Read-as-Zero. Hardware must implement the field as Read-as-Zero. Software can rely on the field reading as all 0s. This applies to a single bit that reads as 0, or to a field that reads as all 0s.

RES0 In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the bit or field must be ignored. Software must not rely on the bit reading as 0, or all 0s for a bit field, and except for writing back to a register must treat the value as if it is UNKNOWN. When writing to the register, software must write the bit as 0, or the bit field as all 0s, if it is writing the field without having previously read the register, or when the register has not been initialized.

If the software that is writing to the register has read the register since the register was last reset, it must preserve the value of the field by writing the value that it previously read from the field. Hardware must ignore writes to these fields. If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field in the same register, software must expect an UNPREDICTABLE result.

A RES0 field that is read-only must read-as-zero, but software must not rely on the field reading as zero, and must treat the value as UNKNOWN. A res0 field that is write-only must be written as zero. The effect of writing an other value to the field is UNPREDICTABLE.

In some ARM Architecture Reference Manuals, the definition of RES0 is extended to cover cases where a bit or field is RES0 in some states, but behaves differently in other states.

When RES0 appears in body text, it is always in SMALL CAPITALS.

RES1 In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the bit or field must be ignored. Software must not rely on the bit reading as 1, or all 1s for a bit field, and except for writing back to a register must treat the value as if it is UNKNOWN. When writing to the register, software must write the bit as 1, or the bit field as all 1s, if it is writing the field without having previously read the register, or when the register has not been initialized.

If the software that is writing to the register has read the register since the register was last reset, it must preserve the value of the field by writing the value that it previously read from the field. Hardware must ignore writes to these fields. If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field in the same register, software must expect an UNPREDICTABLE result.

Each bit of a RES1 field that is read-only must read-as-one, but software must not rely on the bits reading as one, and must treat their values as UNKNOWN. Each bit of a RES1 field that is write-only must be written as one. The effect of writing an other value to the field is UNPREDICTABLE.

In some ARM Architecture Reference Manuals, the definition of RES1 is extended to cover cases where a bit or field is RES1 in some states, but behaves differently in other states.

When RES1 appears in body text, it is always in SMALL CAPITALS.

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.

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

Serial-Wire Debug 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 (SW-DP).

TAP *See* Test Access Port (TAP).

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST***.

Trace port A port on a device, such as a processor or SoC, 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.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not:

- Be a security hazard.
- Be documented or promoted as having a defined value or effect.

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

UNP *See* UNPREDICTABLE.

UNPREDICTABLE

For 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.

In issue A of this document, UNPREDICTABLE also meant an UNKNOWN value.

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

WI Writes ignored.

W1C Writing a 1 to the bit clears the bit to 0. Writing a 0 to the bit has no effect.

Word A 32-bit data item.